# Progetto di Dama Italiana

## Facoltà di Informatica (Pisa)
## Esperienze di Programmazione
## Corso A
## a.a.2020/2021

Andrea Del Corto
matricola: 561446

Sommario

# 1 Descrizione del problema

La dama è un gioco da tavolo tradizionale per due giocatori. La parola "dama" proviene dal latino "domina" ed indica il "pezzo sovrano" e, per estensione, l'intero gioco.

Esistono regole di gioco diverse, prevalentemente nazionali, dato che in quasi tutti i paesi la dama, col tempo, ha assunto regole proprie, benché simili. Le caratteristiche che accomunano tutti i tipi di dama sono l'essere un gioco da tavolo di strategia, durante il quale due giocatori muovono i rispettivi pezzi (pedine e dame) su di un supporto, chiamato damiera, che consta di 64 caselle o di 100 o 144 caselle, metà scure e metà chiare, e catturano i pezzi avversari mediante lo "scavalcamento" degli stessi.

Questo progetto si pone l'obiettivo di creare un programma che permetta di giocare a Dama secondo le regole della dama italiana [1] in una delle seguenti modalità:

1. Umano conto umano.
2. Umano contro AI.
3. AI contro AI.

## 1.1 Proprietà del gioco

Il gioco della dama italiana, dal punto di vista della teoria dei giochi [2], ha le seguenti caratteristiche:

1. Two-player: due giocatori giocano l'uno contro l'altro.
2. Zero-sum: I giochi a somma zero modellano tutte quelle situazioni conflittuali in cui la contrapposizione dei due giocatori è totale: la vincita di un giocatore coincide esattamente con la perdita dell'altro. La somma delle vincite dei due contendenti in funzione delle strategie utilizzate è cioè sempre zero. Nella dama ad esempio significa che i soli tre risultati possibili (rappresentando la vittoria con 1, la sconfitta con -1 e il pareggio con 0) possono essere: 1,-1 se vince il bianco; -1,1 se vince il nero; 0,0 se pareggiano. Non esiste ad esempio il caso in cui vincono entrambi o perdono entrambi.
3. Perfect information: ogni giocatore, quando prende una decisione, lo fa essendo perfettamente informato di tutti gli eventi che si sono verificati in precedenza, quindi in questo caso: stato di partenza del gioco e mosse precedenti dell'avversario.

# 2 Design del programma

## 2.1 Rappresentazione dello stato di gioco

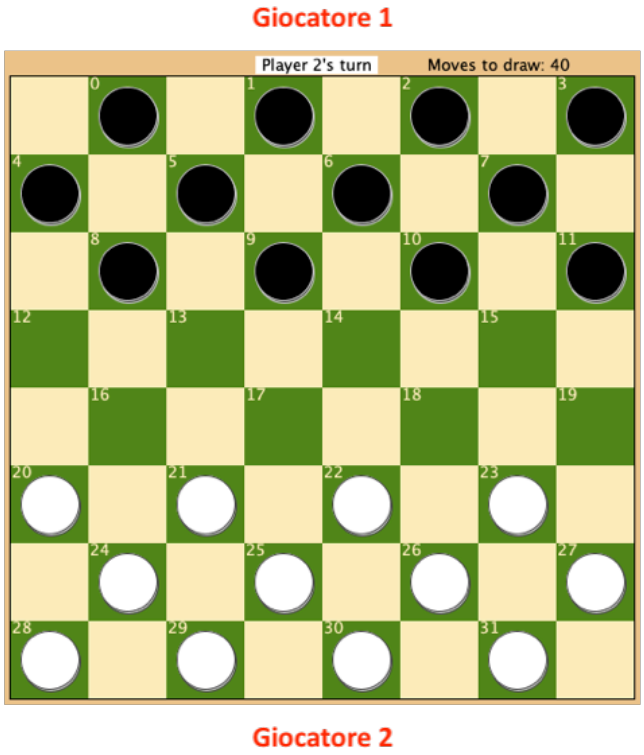



*Figure 1- Damiera, stato iniziale*

Forse uno degli aspetti più importanti della progettazione del programma riguarda come rappresentare uno stato di gioco che nel caso del gioco della dama è costituito da:

a. Posizione e tipologia dei pezzi sulla scacchiera.
b. Giocatore che deve effettuare la prossima mossa.
c. Numero delle mosse mancanti per un pareggio secondo la regola del "conteggio delle mosse" [1].

Le tecniche che si utilizzano per rappresentare giochi come la dama si dividono in tre categorie [3]:

a. "pezzo centriche": tiene traccia di tutti i pezzi che si trovano sulla tavola di gioco, memorizzando per ognun di essi la cella che occupa.
b. "cella centriche": per ogni cella di gioco si memorizza un suo stato che deve essere in grado di indicare se essa contiene un pezzo oppure no ed in caso affermativo anche il tipo del pezzo.
c. "soluzioni ibride": un misto fra le due categorie precedenti.

La dama italiana si gioca su una scacchiera di 8x8 celle dove metà delle celle (32) sono scure mentre l'altra metà sono chiare. Perciò, un modo per rappresentare lo stato della scacchiera, consiste nello sfruttare il fatto che nel gioco della dama italiana i pezzi possono muoversi solamente sulle celle scure (non è così in altre varianti del gioco, tipo la Dama Turca). In questo modo si può adottare una rappresentazione del tipo "cella centrica" nella quale si rappresenta solo lo stato di ognuna delle 32 celle scure della damiera. Le celle scure saranno numerate dall'alto verso il basso e da sinistra verso destra con numeri che vanno da 0 a 31 (come in Figure 1).
Ciascuna cella scura può assumere uno e uno soltanto dei 5 seguenti stati durante il gioco:

| Stato cella | Codifica binaria | Descrizione |
|---|---|---|
| BC | 110 | Sta per "Black Checker", indica che la cella contiene una pedina nera |
| BK | 111 | Sta per "Black King", indica che la cella contiene una dama nera. |
| WC | 100 | Sta per "White Checker", indica che la cella contiene una pedina bianca |
| WK | 101 | Sta per "White King", indica che la cella contiene una dama bianca. |
| E | 000 | Sta per "Empty", indica che la cella non contiene nessun pezzo. |

*Table 1 - Codifica stato celle*

| b2 | b1 | b0 |
|---|---|---|

*Table 2 - significato dei bit di stato*

La codifica binaria degli stati soprariportati segue la seguente logica:

- b2: indica se la cella contiene un pezzo (b2 = 1) oppure no (b2 = 0).
- b1: indica se la cella contiene un pezzo nero (b1 = 1) oppure no (b1 = 0).
- b0: indica se il pezzo contenuto nella cella è una dama (b0 = 1) oppure no (b0 = 0).

Considerando che per rappresentare una cella di gioco si devono poter codificare 5 stati diversi, per ogni cella occorrono 3 bit (che permettono di rappresentare $2^3 = 8$ stati diversi (perciò 3 in più del necessario). In totale occorrono quindi $3 * 32 = 96\ bit$ per rappresentare i pezzi sulla damiera. Considerando che il progetto sarà sviluppato in Java, un intero (int) occupa 4 byte in memoria (32 bit) perciò i 96 bit necessari possono essere memorizzati utilizzando un array di 3 int nel modo seguente:

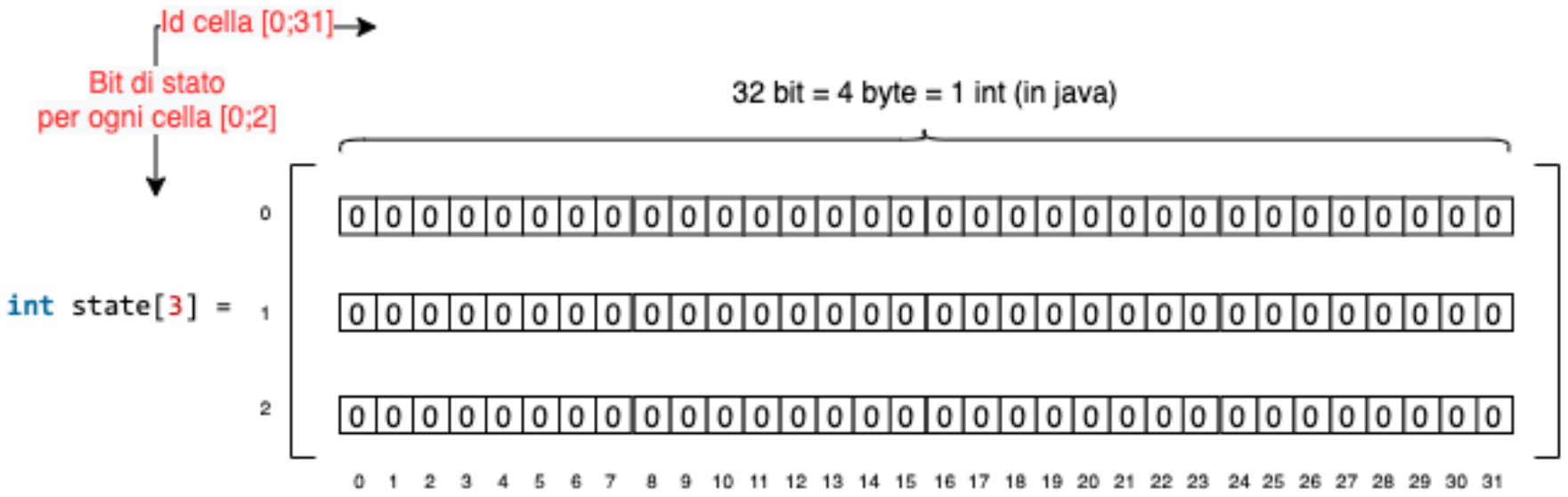



*Figure 2 - Rappresentazione Damiera vuota*

L'array "state" sopraripportato rappresenta una damiera vuota, infatti ogni colonna di bit, che rappresenta lo stato di ciascuna cella scura, è configurato come: 000, ovvero con lo stato"E".

Per dare un'idea più completa di come funziona questa rappresentazione, di seguito viene riportato l'array "state" configurato per rappresentare una damiera con tutti i pezzi nella loro posizione di partenza (come in Figure 1) per dare inizio ad una nuova partita.

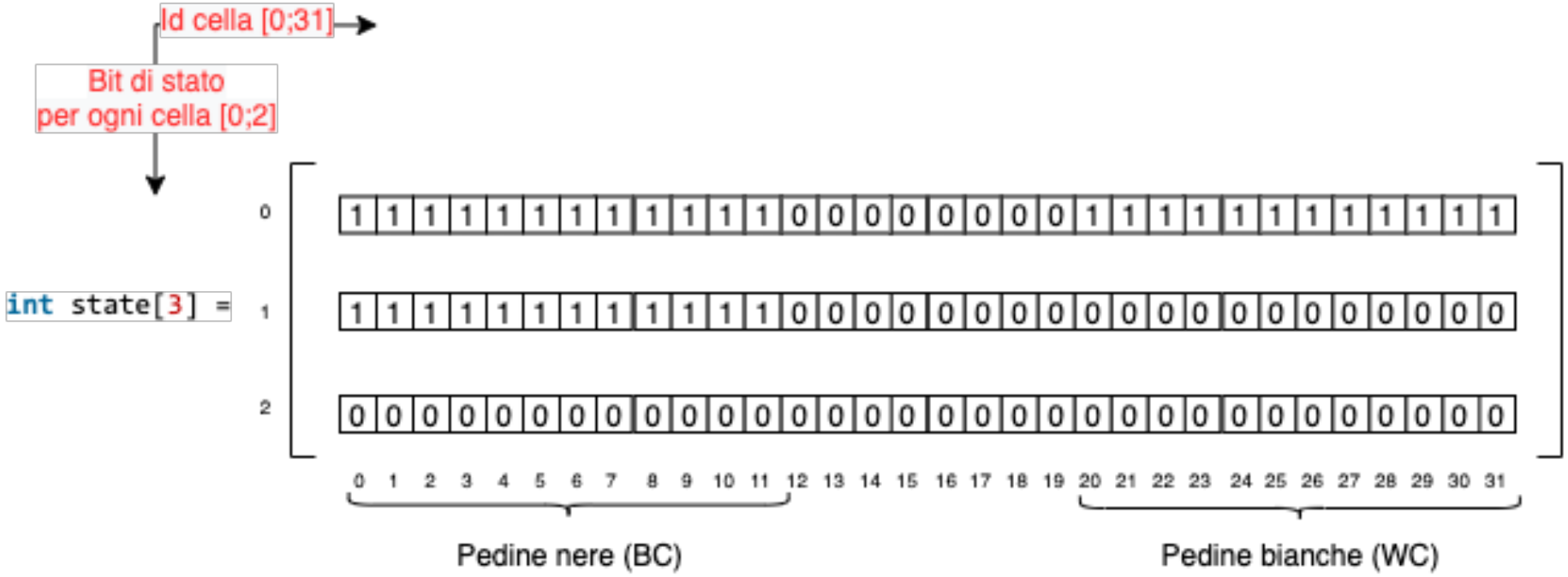



*Figure 3 - Rappresentazione Damiera pronta per una nuova partita*

## 2.2 Struttura del programma

In questo capitolo sarà descritta la struttura generale del programma e le idee chiave che hanno guidato la sua scrittura.

Il metodo main della classe Main è il punto di inizio per l'esecuzione del programma e il suo unico scopo è quello di creare un'istanza di CheckersWindow.
L'unica istanza di CheckersWindow rappresenta l'unica finestra del programma, nonché l'unico punto di interazione programma/utente.
Durante l'inizializzazione di CheckersWindow, vengono create le sue componenti grafiche (CheckerBoard e OptionPanel) e un'istanza della classe GameManager.
La classe GameManager gestisce tutta la logica di gioco, come: turni, pausa, cambio delle modalità di gioco, etc. e l'idea è quella di separare nettamente l'interfaccia grafica dalla logica di gioco.
CheckersWindow svolge il ruolo di punto di comunicazione fra GameManager e gli eventi generati dall'interfaccia grafica.

Per maggiori dettagli consultare le seguenti appendici:

a. Il codice del programma in "Appendice 1: Codice".
b. Diagramma UML delle classi in "Appendice 2: UML, diagramma delle classi"
c. Javadoc del progetto in "Appendice 3: Javadoc del progetto".

# 3 Come usare il programma

Quando il programma viene eseguito, la prima schermata che viene mostrata all'utente è la seguente:

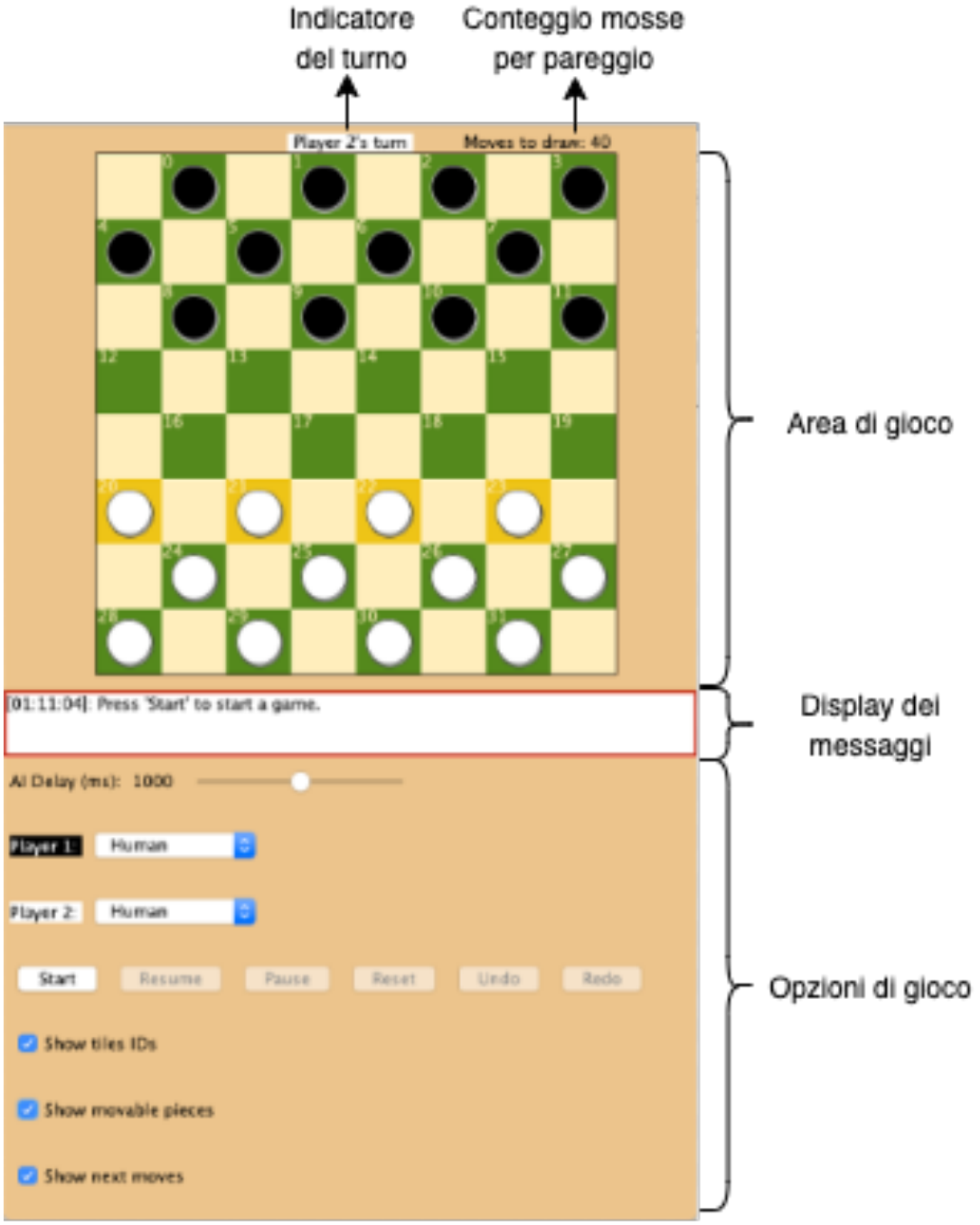



*Figure 4 - Schermata principale*

Si tratta dell'unica finestra con la quale l'utente può interagire.

## 3.1 Area di gioco

L'area di gioco ha due funzioni principali:

1. Mostrare all'utente lo stato di gioco corrente della partita.
2. Permettere all'utente di selezionare una mossa nel caso in cui il giocatore corrente, indicato da "Indicatore del turno", sia un giocatore umano ("Human").

Di seguito una descrizione dei comandi presenti all'interno dell'area "Area di gioco":

- Indicatore di turno: indica quale giocatore (Player 1 o Player 2) deve effettuare la prossima mossa.
- Conteggio mosse per pareggio: come da regolamento ufficiale della dama italiana [1], questo contatore rimane sempre a 40 finché non ci sono dame sulla damiera. Dal momento in cui una dama comincia a far parte del gioco, ogni mossa che viene fatta senza catturare nessun pezzo, indipendentemente da chi lo fa, provoca il decremento del contatore. Con almeno una dama sulla damiera, se viene fatta una mossa dove viene catturato un pezzo, il contatore viene reimpostato con il suo valore iniziale, ovvero: 40.
  Sei il contatore raggiunge lo zero, significa che la partita è in "stallo" e viene dichiarato il pareggio (o "patta").

## 3.2 Display dei messaggi

Viene utilizzato per comunicare all'utente maggiori dettagli sugli eventi che si verificano durante una partita e su eventuali azioni richieste per proseguire.

## 3.3 Opzioni di gioco

Di seguito una descrizione dei comandi presenti all'interno dell'area "Opzioni di gioco":

- AI Delay (ms): è uno slider che permette di definire il ritardo prima che un giocatore classificato come giocatore non umano inizi a valutare quale mossa scegliere. Può assumere un valore, espresso in millisecondi, che oscilla fra 0 e 2000 compresi.
- Player 1 e Player 2: permettono di selezionare la tipologia di giocatore, rispettivamente per il giocatore 1 e il giocatore 2. Le possibili opzioni sono le seguenti (il prefisso "AI" indica che si tratta di una tipologia di giocatore gestita dal computer):
  - Human: le mosse devono essere selezionate utilizzando "3.1 Area di gioco".
  - AI – Random: le mosse sono selezionate automaticamente dal programma in modo casuale.
  - AI – MinMax: le mosse sono selezionate automaticamente dal programma scegliendo la mossa migliore secondo l'algoritmo del min-max (maggiori dettagli in "4 Algoritmi di AI")
- Start: questo pulsante permette di dare il via al gioco.
- Resume: questo pulsante permette di riprendere il gioco dopo aver messo il gioco in pausa con il pulsante "Pause".
- Pause: permette di mettere il gioco in pausa e di modificare eventualmente i campi Player 1 e Player 2.
- Undo: permette di annullare l'ultima mossa fatta.
- Redo: permette di rieseguire l'ultima mossa annullata.
- Show tiles IDs: solo se il flag è attivo, le celle scure mostreranno il loro id in alto a sinistra.
- Show movable pieces: solo se il flag è attivo, i pezzi, che appartengono al giocatore a cui tocca scegliere la prossima mossa, che hanno almeno una mossa possibile, vengono indicati evidenziando in giallo la cella sulla quale si trovano.

- Show next moves: solo se il flag è attivo, i pezzi, che appartengono al giocatore umano a cui tocca scegliere la prossima mossa, che hanno almeno una mossa possibile, se vengono selezionati vengono evidenziate in azzurro le celle sulle quali si possono spostare.

# 4 Algoritmi di AI

In questo paragrafo saranno descritti gli algoritmi che permettono al computer di giocare autonomamente. Nel programma sono previsti sono due tipi di AI:

1. AI – Random: le mosse sono selezionate automaticamente dal computer in modo casuale.
2. AI – MinMax: le mosse sono selezionate automaticamente dal computer scegliendo la mossa migliore secondo l'algoritmo del min-max.

Il primo tipo di AI sarà utile (per il quale forse è anche eccessivo parlare di AI) per verificare se AI – MinMax effettua delle "mosse sensate". Questo aspetto sarà approfondito nel paragrafo "5 Verifica empirica dell'efficacia di MinMax".

## 4.1 Metodi per affrontare il problema

Si possono pensare vari metodi per affrontare il problema di scrivere un programma in grado di giocare a dama effettuando "buone mosse" (ci si riferisce a mosse che portano il giocatore che le effettua in uno stato di gioco per lui più vantaggioso e che quindi ha più probabilità di condurlo alla vittoria):

1. Metodo analitico
2. Metodo If-Then
3. Look ahead

### 4.1.1 Metodo analitico

Conoscendo la posizione dei pezzi potremmo pensare di determinare la prossima mossa seguendo una strategia predefinita (come fa un giocatore umano). Il problema è che per il momento non è stata ancora scoperta nessuna strategia o tattica che permette di scegliere "buone mosse" effettuando un'analisi diretta della damiera.

### 4.1.2 Metodo If-Then

Viene fatta una classifica di tutte le possibili mosse, assegnando loro un peso determinato secondo particolari criteri, per esempio: si potrebbe considerare come mossa di peso elevato una mossa dove si riesce a catturare una dama avversaria, oppure una dove si riesce a fare una cattura multipla, mentre si potrebbe considerare come mossa di peso minore una mossa che porta un pezzo in una posizione che nel prossimo turno permetterà al giocatore avversario di catturarlo.

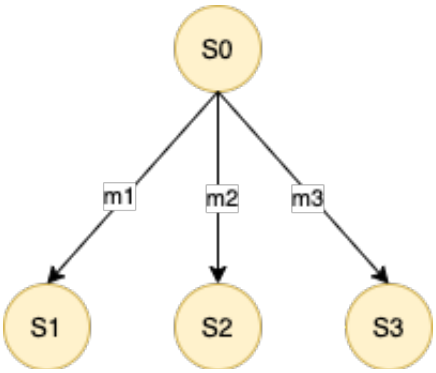



*Figure 5 - Esempio Metodo If Then*

Prediamo come esempio l'albero sopra riportato. La radice dell'albero (S0) rappresenta lo stato di gioco (maggiori dettagli in "2.1 Rappresentazione dello stato di gioco") di partenza, mentre le foglie S1, S2 e S3 rappresentano gli stati nei quali si può passare se si effettua rispettivamente la mossa m1, m2 o m3.

Formalizziamo adesso i seguenti concetti, che torneranno utili anche in seguito:

| | |
|---|---|
| S | Insieme di tutti i possibili stati di gioco. |
| M | Insieme di tutte le mosse che possono essere fatte in uno stato di gioco. Una mossa è caratterizzata dalle seguenti proprietà:<br>1. $\mathbf{m.startID} \in [0;31] \sqsubseteq \mathbb{N}$ si riferisce all'indice della cella di partenza della mossa.<br>2. $\mathbf{m.endID} \in [0;31] \sqsubseteq \mathbb{N}$ si riferisce all'indice della cella di arrivo della mossa.<br>3. $\mathbf{m.weight} \in \mathbb{R}$ si riferisce al peso associato alla mossa. |
| $moves: S\ x\ S$ | Funzione che prende come input uno stato di gioco e restituisce l'insieme delle mosse legali[1] possibili a partire da esso. |
| $weight: (S\ x\ M)\ x\ \mathbb{R}$ | Funzione che prende come input uno stato di gioco e una mossa legale e restituisce il peso associato a tale mossa. |

Ritornando all'esempio precedente, possiamo adesso esprimere in modo più formale la scelta della mossa migliore secondo questo approccio:

[1] Con mossa legale si intende una mossa le cui proprietà startID e endID si riferiscono a mosse che rispettano il regolamento di gioco [1].

| | |
|---|---|
| $M_0 = moves(S0)$ | Mosse legali disponibili nello stato S0 dell'esempio precedente. |
| $\forall\, m \in M_0 . m.weight = weight(S0, m)$ | Assegnazione del peso alle mosse disponibili nello stato S0. |
| $\max\{m.weight \mid m \in M_0\}$<br>$\arg m$ | Scelta della mossa migliore effettuata scegliendo una delle mosse di peso massimo. |

### 4.1.3 Look ahead and evaluate

Per tutti gli stati che possono essere raggiunti partendo dallo stato corrente, si calcola il loro valore statico grazie ad un'apposita funzione chiamata funzione di valutazione statica la quale valuta staticamente[2] quanto uno stato è "buono"per il giocatore 1 (la scelta è del tutto arbitraria), rispetto a determinate caratteristiche.

Formalizziamo adesso i seguenti concetti, che torneranno utili anche in seguito:

| | |
|---|---|
| $staticEval: S\ x\ \mathbb{R}$ | Funzione di valutazione statica. Prende come unico parametro uno stato di gioco e ne restituisce la valutazione statica sotto forma di valore reale. |
| $applyMove: (S\ x\ M)\ x\ S$ | Funzione che prede uno stato di gioco e una mossa legale in tale stato come parametro. Restituisce il nuovo stato che si ottiene applicando la mossa. |

Sfruttando ancora una volta l'esempio precedente, possiamo adesso esprimere in modo più formale la scelta della mossa migliore secondo questo approccio:

| | |
|---|---|
| $M_0 = moves(S0)$ | Mosse legali disponibili nello stato S0 dell'esempio precedente. |
| $S0_{next} = \{s \in S \mid m \in M_0 \wedge s = applyMove(S0, m)\}$ | Viene determinato l'insieme degli stati raggiungibili a partire da S0 applicando ad esso tutto le mosse legali possibili ($M_0$). |
| $\max\{staticEval(s) \mid m \in M_0 \wedge s = applyMove(S0, m)\}$<br>$\arg m$ | Scelta della mossa migliore effettuata scegliendo una delle mosse che conduce in uno stato il cui valore statico è il massimo fra tutti quelli raggiungibili. |

[2] Con "staticamente" si intende dire che la valutazione viene fatta valutando solamente lo stato corrente, senza fare valutazioni che riguardano possibili stati futuri.

### 4.1.4 Min-Max

L'algoritmo del Min-Max determina la prossima mossa basandosi sulle seguenti assunzioni:

1. Il giocatore che deve scegliere la prossima mossa viene denominato "Max" e questo nome allude al fatto che l'obiettivo di questo giocatore è quello di raggiungere stati la cui valutazione statica, dalla prospettiva di Max, è massima.
2. Il giocatore avversario, che dovrà scegliere la mossa nel turno successivo, viene denominato "Min" e questo nome allude al fatto che l'obiettivo di questo giocatore è quello di raggiungere stati la cui valutazione statica, dalla prospettiva di Max, è minima. Come descritto in "1.1 Proprietà del gioco" il gioco della dama è un gioco che gode della proprietà "zero-sum", perciò se il giocatore "Min" gioca cercando di condurre il gioco verso stati dove Max ha meno possibilità di raggiungere stati di valore alto, equivale a dire che Min gioca in modo ottimo, al pari di Max.

Per spiegare la logica di funzionamento dell'algoritmo del Min-Max, sfruttiamo il seguente esempio:

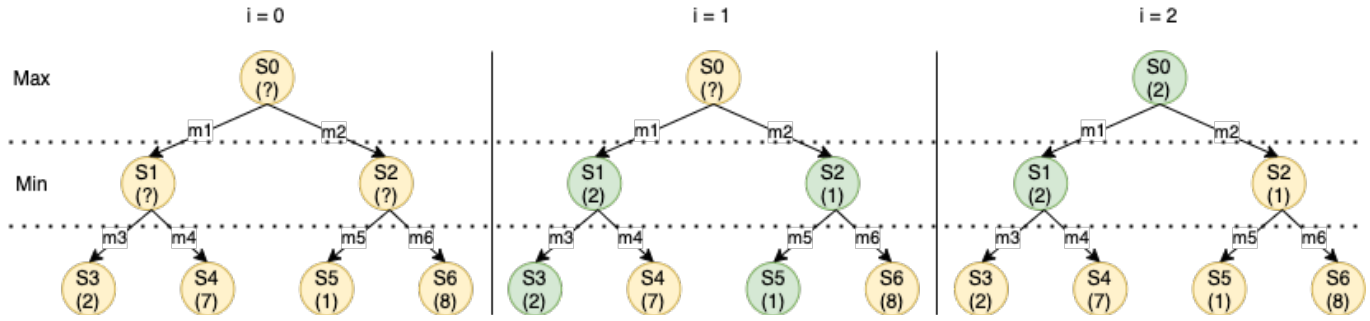



*Figure 6 - Simulazione Min-Max*

Supponiamo che l'albero rappresentato per i = 0 sia l'albero che rappresenta tutti gli stati raggiungibili partendo da un generico stato S0. I numeri che compaiono fra parentesi rappresentano il valore che assume la funzione di valutazione statica in quello stato. Se fra le parentesi si trova un "?" significa che il valore statico di quello stato non è stato ancora determinato per quel nodo/stato. Inizialmente vengono valutate solo le foglie dell'albero dalla prospettiva di Max.
A questo punto si procede a ritroso, salendo di un livello, partendo quindi dal livello 3 si raggiunge il livello 2 dove è il turno del giocatore "Min".
Assumendo che "Min" si comporti come descritto in precedenza, la mossa che sceglie, partendo da S1, è la mossa che gli permette di raggiungere il nodo figlio con il valore più basso e quindi in questo caso, partendo da S1, la mossa che sceglie è m3 che lo porta nello stato S3. Secondo la stessa logica, possiamo dire che partendo dallo stato S2 il giocatore "Min" sceglie la mossa m5 che lo porta nello stato S5.
A livello 1 è il turno di "Max", il quale, dovendo scegliere la strada che lo conduce allo stato di valore massimo, sceglie la mossa m1 che lo porta nello stato S1.

Concettualmente l'algoritmo è molto semplice, ma nella pratica non è applicabile in questa sua versione, in quanto richiederebbe un eccessivo costo computazionale.
Infatti, se volessimo eseguire l'algoritmo del "Min-Max" esattamente in questo modo, dovremmo per prima cosa generare l'albero degli stati completo, ovvero dovremmo espandere i nodi dell'albero, finché tutte le sue foglie non rappresentano uno stato terminale del gioco. Nel gioco della dama, partendo da un generico stato di gioco, se ci si trova in una situazione nella quale si può

catturare, in media si può scegliere solo 1 mossa, mentre se ci si trova in un situazioni in cui non si può catturare nessun pezzo in media si può scegliere 8 mosse (dati recuperati da [4]). Quindi, per fare un'analisi qualitativa, si può assumere che in media il numero di mosse che possono essere scelte partendo da un generico stato (da questo momento in poi questo valore, chiamato anche "branching factor", sarà indicato con la lettera "b") è pari a $(1 + 8)/2 \cong 4$ (approssimando all'interno più piccolo).
Per avere un'idea di quanto possa essere grande l'albero completo delle mosse per la dama italiana non rimane che trovare quanti turni dura in media una partita di dama italiana, da cui possiamo dedurre quanto è profondo l'albero completo degli stati di gioco (da questo momento in la profondità dell'albero, chiamata anche "depth", sarà indicata con la lettera "d").

Sul web non sono state trovate informazioni in merito, perciò per avere un'idea approssimativa di quanto possa valere d è stata fatta la seguente analisi:

1. Sono stati recuperati dall'archivio delle partite di dama italiana [5] tutti i risultati disponibili in formato PDN (Portable Draughts Notation) [6] di tornei di dama italiana.
   Per la precisione gli archivi analizzati sono i seguenti:

| Archivio | Link |
|---|---|
| ci2004ita.pdn | http://www.federdama.it/cms/servizi/download/database-di-partite/doc_download/42-il-database-completo-con-le-partite-del-campionato-italiano-assoluto-2004-di-dama-italiana |
| ci2005ita.pdn | http://www.federdama.it/cms/servizi/download/database-di-partite/doc_download/40-il-database-completo-con-le-partite-del-campionato-italiano-assoluto-2005-di-dama-italiana |
| ci2010.pdn | http://www.federdama.it/cms/servizi/download/database-di-partite/doc_download/58-partite-del-campionato-italiano-assoluto-2010-di-dama-italiana |

*Table 3 - Archivi analizzati*

2. Sono stati raccolti i seguenti dati:

| Archivio | Numero partite totali | Vittorie bianco | Vittorie nero | Pareggi | Numero di mosse di tutte le partite | Numero di mosse per partita (media) |
|---|---|---|---|---|---|---|
| ci2004ita.pdn | 190 | 36 | 42 | 112 | 11555 | 11555/190 = 60,8 |
| ci2005ita.pdn | 153 | 27 | 39 | 87 | 10199 | 10199/153= 66,67 |
| ci2010.pdn | 688 | 143 | 144 | 401 | 38814 | 38814/688 = 56,4 |
| TOT | 1031 | 206 | 225 | 600 | 60568 | 60568/1031=58,7 |

*Table 4 - Risultati dell'analisi degli archivi*

Ne segue dunque che si può stimare d=59.

Questo significa che nel caso medio, l'albero è composto da:

$$b^d = 4^{59} \text{ foglie}$$

Volendo esprime questo numero in base 10:

$$4^{59} = 10^x = 10^{36}$$
$$x = log_{10}(4^{59}) = 2 * 59 * log_{10}(2) \approx 36$$

È evidente che si tratta di un numero troppo elevato da calcolare e per rendersene conto può essere utile fare un rapido test:

$$Numero\ nanosecondi\ in\ 1\ anno\ =\ 3{,}154 * 10^7\ \cancel{sec}/anni * 10^9 ns/\cancel{sec}\ \cong\ 10^{16}\ ns/anni$$
$$Numero\ nanosecondi\ trascorsi\ dall'inizo\ dell'universo\ =\ 10^{16}\ ns/anni\ *\ 10^{10} anni\ =\ 10^{26}\ ns$$

Assumendo che la funzione di valutazione statica impieghi 1ns per essere calcolata, se dall'inizio della storia dell'universo ogni singolo ns fosse stato utilizzato per risolvere questo problema, rimarrebbero ancora $10^{36} - 10^{26}\ ns = 9.999999999 * 10^{35}\ ns = 3.16887646 \times 10^{19}\ anni$

#### *4.1.4.1 Descrizione delle scelte implementative*

Considerando l'elevato costo computazionale descritto nel paragrafo precedente il min-max non può essere applicato sull'intero albero degli stati.
Infatti nella pratica si utilizza una sua versione nella quale si applica all'albero costruito fino ad un predeterminato livello di profondità (parametro che viene chiamato "ply").
Esistono anche algoritmi più raffinati che variano il parametro ply in funzione delle fasi di gioco in cui ci si trova (per maggiori dettagli consultare [7]).

##### 4.1.4.1.1 Scelta parametro Ply

Per questo progetto il parametro ply è stato scelto nel seguente modo.
Per ognuno dei 3 valori del parametro d, elencati nella tabella seguente, sono state simulate 5 partite fra Player 1 = "AI - MinMax", Player 2 = "AI - Random" impostando "AI Delay" = 0 ed è stato calcolato il tempo medio necessario a Player 1 per scegliere una mossa:

| d | Tempo medio per scegliere una mossa (ns) | Tempo medio per scegliere una mossa (sec) |
|---|---|---|
| 5 | 32632005,7 | 0,032632006 |
| 7 | 483320051,3 | 0,483320051 |
| 9 (*) | 20255524674 | 20,25552467 |

*Table 5 - Analisi tempo medio per una mossa*

(*) Nel caso della prova con d=9, è stata simulata una sola partita in quanto è evidente che con questo valore i tempi di gioco diventano eccessivamente lunghi.
Alla luce dei risultati ottenuti, è stato deciso di utilizzare d=7, che rappresenta un buon compromesso fra profondità raggiunta e tempo necessario per scegliere una mossa.
In "Appendice 4: Analisi parametro Ply" si può trovare il dettaglio dei dati utilizzati per effettuare questa analisi.

#### 4.1.4.1.2 Funzione di valutazione statica

La funzione di valutazione statica si occupa di valutare quanto un uno stato di gioco sia "buono" per un giocatore assegnandoli un valore numerico. Una buona funzione di valutazione (secondo quanto scritto in [8]) j ha le seguenti caratteristiche:

1. Ordina gli stati terminali dando un peso elevato a stati in cui il giocatore per il quale viene valutato lo stato vince, un valore minore per gli stati in cui ottiene un pareggio e un valore ancora più piccolo per gli stati in cui viene sconfitto.
2. La computazione non deve richiedere troppo tempo
3. La funzione deve restituire valori che hanno una forte correlazione con la probabilità di vittoria, quindi ad elevati valori devono corrispondere elevate probabilità di vittoria.

La funzione di valutazione statica utilizzata in questo progetto valuta uno stato di gioco nel seguente modo.

Viene sommato il valore di tutti i pezzi del giocatore per il quale viene valutato lo stato e da questo numero viene sottratta la somma del valore di tutti i pezzi del giocatore avversario.

I valori sono associati ai pezzi nel seguente modo:

| Pezzo | Valore |
|---|---|
| Pedina | 1 |
| Dama | 2 |

*Table 6 - Valore dei pezzi*

La scelta di una buona funzione di valutazione statica non è semplice, in particolar modo non è facile stimare quanto peso attribuire alle caratteristiche (o "features") che si prendono in considerazione per valutare lo stato.
In programmi più avanzati rispetto a quello proposto con questo progetto, il peso delle features viene determinato sfruttando tecniche di machine learning [8].

# 5 Verifica empirica dell'efficacia di MinMax

Questo capitolo si pone l'obiettivo di valutare se "AI -MinMax" è in grado di giocare in modo migliore rispetto a "AI - Random". L'ipotesi che si intende verificare con questa prova è la seguente: Se "AI – MinMax" è in grado di vincere la maggior parte delle partite giocando contro "AI - Random" allora si può concludere che "AI - MinMax" è in grado di sceglie mosse migliori rispetto ad un giocatore che gioca in modo casuale e che quindi le mosse scelte siano in qualche modo delle "buone scelte". Ovviamente valutare quanto le mosse scelte siano buone è un compito ben più arduo, per il quale sarebbero necessarie analisi più approfondite.

Per questa prova sono state simulate 20 partite fra Player 1 = "AI - MinMax", Player 2 = "AI - Random" impostando "AI Delay" = 0. Nella tabella riportata di seguito i risultati della prova:

| Giocatore | Vittorie Player 1 | Vittorie Player 2 | Pareggi |
|---|---|---|---|
| Player 1 = "AI - MinMax" | 20 | 0 | 0 |

Dai risultati raccolti l'ipotesi fatta all'inizio di questo capitolo risulta essere confermata.

# Appendice 1: Codice

## 5.1 AIMinMax

```java
package com.dca.checkers.ai;

import com.dca.checkers.model.GameState;
import com.dca.checkers.model.Move;
import com.dca.checkers.model.Player;

import java.util.List;

/**
 * The {@code AIRandomPlayer} class represents a AI player that updates
 * the board based one alpha beta algorithm.
 */
public class AIMinMax implements Player {

   /**
    * Depth of the tree to build.
    */
   private static final int depth = 7;

   /** Depth of the tree to build. */
   private boolean isBlack;

   /**
    * Flag that tells if the move has been performed.
    */
   private boolean moveDone;


   @Override
   public boolean isHuman() {
      return false;
   }

   @Override
   synchronized public void updateGame(GameState gameState) {
      moveDone = false;
      // Nothing to do
      if (gameState == null || gameState.isGameOver()) {
         moveDone = true;
         return;
      }
      isBlack = gameState.isP1Turn();
      //Select best move
      MinMaxResult bestResult = minMax(gameState.copy(), null, depth, true);
      //Apply best move
      gameState.move(bestResult.move.getStartIndex(), bestResult.move.getEndIndex());
      moveDone = true;
   }

   @Override
   public boolean hasSkipped() {
      return false;
   }

   @Override
   public boolean hasMoved() {
      return moveDone;
   }

   private class MinMaxResult {
      Move move;
      double value;
```

```
      public MinMaxResult(Move m, double v) {
         move = m;
         value = v;
      }

   }

   /**
    * Execute min-max algorithm in order to find the best move.
    *
    * @param g the game state to evaluate
    * @param depth the maximum recursion depth
    * @param isMaxPlayer flag that tells if the current player is max (true) or min
(false)
    * @return the result of min max algorithm.
    */
   private MinMaxResult minMax(GameState g, Move m, int depth, boolean isMaxPlayer) {
      if(depth == 0 || g.isGameOver()) return new MinMaxResult(m, g.value(isBlack));

      double maxVal;
      double minVal;
      Move bestMove = null;
      double bestValue = 0;

      if(isMaxPlayer) {
         maxVal = Integer.MIN_VALUE;
         //Get the available moves
         List<Move> moves = g.getAllMoves();
         //Evaluate all games state reachable with each possible move
         for (Move possibleMove : moves) {
            GameState childState = g.copy();
            childState.move(possibleMove.getStartIndex(), possibleMove.getEndIndex());
            MinMaxResult resChild = minMax(childState, possibleMove, depth - 1, false);
            if(resChild.value > maxVal) {
               maxVal = resChild.value;
               bestMove = possibleMove;
               bestValue = resChild.value;
            }
         }
         return new MinMaxResult(bestMove, bestValue);

      } else {//Min player
         minVal = Integer.MAX_VALUE;
         //Get the available moves
         List<Move> moves = g.getAllMoves();
         //Evaluate all games state reachable with each possible move
         for (Move possibleMove : moves) {
            GameState childState = g.copy();
            childState.move(possibleMove.getStartIndex(), possibleMove.getEndIndex());
            MinMaxResult resChild = minMax(childState, possibleMove, depth - 1, true);
            if(resChild.value < minVal) {
               minVal = resChild.value;
               bestMove = possibleMove;
               bestValue = resChild.value;
            }
         }
         return new MinMaxResult(bestMove, bestValue);
      }


   }

   @Override
   public String toString() {
      return getClass().getSimpleName() + "[isHuman=" + isHuman() + "]";
   }
}
```

## 5.2 AIRandomPlayer

```java
package com.dca.checkers.ai;

import com.dca.checkers.model.*;

import java.util.List;

/**
 * The {@code AIRandomPlayer} class represents a AI player who plays randomly
 */
public class AIRandomPlayer implements Player {

   /**
    * Flag that tells if the move has been performed.
    */
   private boolean moveDone;

   @Override
   public boolean isHuman() {
      return false;
   }

   @Override
   synchronized public void updateGame(GameState gameState) {
      moveDone = false;
      // Nothing to do
      if (gameState == null || gameState.isGameOver()) {
         moveDone = true;
         return;
      }

      // Get the available moves
      GameState copy = gameState.copy();
      List<Move> moves = copy.getAllMoves();
      // Choose a random move
      int moveId = (int)(Math.random() * moves.size()-1);
      Move selectedMove = moves.get(moveId);
      gameState.move(selectedMove.getStart(), selectedMove.getEnd());
      moveDone = true;
   }

   @Override
   public boolean hasSkipped() {
      return false;
   }

   @Override
   public boolean hasMoved() {
      return moveDone;
   }

   @Override
   public String toString() {
      return getClass().getSimpleName() + "[isHuman=" + isHuman() + "]";
   }

}
```

## 5.3 Board

```java
package com.dca.checkers.model;

import java.awt.Point;
import java.util.ArrayList;
import java.util.List;

/**
 * The {@code Board} class represents a game state for checkers. A standard
 * checker board is 8 x 8 (64) tiles, alternating white/black. Checkers are
 * only allowed on black tiles and can therefore only move diagonally. The
 * board is optimized to use as little memory space as possible and only uses
 * 3 integers to represent the state of the board (3 bits for each of the 32
 * tiles). This makes it fast and efficient to {@link #copy()} the board state.
 * <p>
 * This class uses integers to represent the state of each tile and
 * specifically uses these constants for IDs: {@link #EMPTY},
 * {@link #BLACK_CHECKER}, {@link #WHITE_CHECKER}, {@link #BLACK_KING},
 * {@link #WHITE_KING}.
 * <p>
 * Tile states can be retrieved through {@link #get(int)} and
 * {@link #get(int, int)}. Tile states can be set through
 * {@link #set(int, int)} and {@link #set(int, int, int)}. The entire game can
 * be reset with {@link #reset()}.
 */
public class Board {
   /** Number of rows */
   private final int nRows = 8;

   /** Number of columns */
   private final int nCols = 8;

   /** An ID indicating a point was not on the checker board. */
   public static final int INVALID = -1;

   /** The ID of an empty checker board tile. */
   public static final int EMPTY = 0b000;

   /** The ID of a white checker in the checker board. */
   public static final int BLACK_CHECKER = 0b110; //4 * 1 + 2 * 1 + 1 * 0;

   /** The ID of a white checker in the checker board. */
   public static final int WHITE_CHECKER = 0b100; //4 * 1 + 2 * 0 + 1 * 0;

   /** The ID of a black checker that is also a king. */
   public static final int BLACK_KING = 0b111; //4 * 1 + 2 * 1 + 1 * 1;

   /** The ID of a white checker that is also a king. */
   public static final int WHITE_KING = 0b101; //4 * 1 + 2 * 0 + 1 * 1;

   /** The current state of the board, represented as three integers. */
   private int[] state;

   /**
    * Constructs a new checker game board, pre-filled with a new game state.
    */
   public Board() {
      reset();
   }

   /**
    * Creates an exact copy of the board. Any changes made to the copy will
    * not affect the current object.
    *
    * @return a copy of this checker board.
    */
```

```
    public Board copy() {
        Board copy = new Board();
        copy.state = state.clone();
        return copy;
    }

    /**
     * Resets the checker board to the original game state with black checkers
     * on top and white on the bottom. There are both 12 black checkers and 12
     * white checkers.
     */
    public void reset() {

        // Reset the state
        this.state = new int[3];
        for (int i = 0; i < 12; i ++) {
            set(i, BLACK_CHECKER);
            set(31 - i, WHITE_CHECKER);
        }
    }

    /**
     * Searches through the checker board and finds black tiles that match the
     * specified ID.
     *
     * @param id   the ID to search for.
     * @return a list of points on the board with the specified ID. If none
     * exist, an empty list is returned.
     */
    public List<Point> find(int id) {

        // Find all black tiles with matching IDs
        List<Point> points = new ArrayList<>();
        for (int i = 0; i < 32; i ++) {
            if (get(i) == id) {
                points.add(toPoint(i));
            }
        }

        return points;
    }

    /**
     * Sets the ID of a black tile on the board at the specified location.
     * If the location is not a black tile, nothing is updated. If the ID is
     * less than 0, the board at the location will be set to {@link #EMPTY}.
     *
     * @param x       the x-coordinate on the board (from 0 to 7 inclusive).
     * @param y       the y-coordinate on the board (from 0 to 7 inclusive).
     * @param id   the new ID to set the black tile to.
     * @see  #set(int, int)
     */
    public void set(int x, int y, int id) {
        set(toIndex(x, y), id);
    }

    /**
     * Sets the ID of a black tile on the board at the specified location.
     * If the location is not a black tile, nothing is updated. If the ID is
     * less than 0, the board at the location will be set to {@link #EMPTY}.
     *
     * @param index    the index of the black tile (from 0 to 31 inclusive).
     * @param id   the new ID to set the black tile to.
     * @see #set(int, int, int)
     */
    public void set(int index, int id) {

        // Out of range
```

```
        if (!isValidIndex(index)) {
            return;
        }

        // Invalid ID, so just set to EMPTY
        if (id < 0) {
            id = EMPTY;
        }

        // Set the state bits
        for (int i = 0; i < state.length; i ++) {
            boolean set = ((1 << (state.length - i - 1)) & id) != 0;
            this.state[i] = setBit(state[i], index, set);
        }
    }

    /**
     * Gets the ID corresponding to the specified point on the checker board.
     *
     * @param x    the x-coordinate on the board (from 0 to 7 inclusive).
     * @param y    the y-coordinate on the board (from 0 to 7 inclusive).
     * @return the ID at the specified location or {@link #INVALID} if the
     * location is not on the board or the location is a white tile.
     * @see #get(int)
     * @see #set(int, int)
     * @see #set(int, int, int)
     */
    public int get(int x, int y) {
        return get(toIndex(x, y));
    }

    /**
     * Gets the ID corresponding to the specified point on the checker board.
     *
     * @param index    the index of the black tile (from 0 to 31 inclusive).
     * @return the ID at the specified location or {@link #INVALID} if the
     * location is not on the board.
     * @see  #get(int, int)
     * @see #set(int, int)
     * @see #set(int, int, int)
     */
    public int get(int index) {
        if (!isValidIndex(index)) {
            return INVALID;
        }
        return getBit(state[0], index) * 4 + getBit(state[1], index) * 2
                + getBit(state[2], index);
    }

    /**
     * Converts a black tile index (0 to 31 inclusive) to an (x, y) point, such
     * that index 0 is (1, 0), index 1 is (3, 0), ... index 31 is (7, 7).
     *
     * @param index    the index of the black tile to convert to a point.
     * @return the (x, y) point corresponding to the black tile index or the
     * point (-1, -1) if the index is not between 0 - 31 (inclusive).
     * @see #toIndex(int, int)
     * @see #toIndex(Point)
     */
    public static Point toPoint(int index) {
        int y = index / 4;
        int x = 2 * (index % 4) + (y + 1) % 2;
        return !isValidIndex(index)? new Point(-1, -1) : new Point(x, y);
    }

    /**
     * Converts a point to an index of a black tile on the checker board, such
     * that (1, 0) is index 0, (3, 0) is index 1, ... (7, 7) is index 31.
```

```
     *
     * @param x    the x-coordinate on the board (from 0 to 7 inclusive).
     * @param y    the y-coordinate on the board (from 0 to 7 inclusive).
     * @return the index of the black tile or -1 if the point is not a black
     * tile.
     * @see  #toIndex(Point)
     * @see #toPoint(int)
     */
    public static int toIndex(int x, int y) {

        // Invalid (x, y) (i.e. not in board, or white tile)
        if (!isValidPoint(new Point(x, y))) {
            return -1;
        }

        return y * 4 + x / 2;
    }

    /**
     * Converts a point to an index of a black tile on the checker board, such
     * that (1, 0) is index 0, (3, 0) is index 1, ... (7, 7) is index 31.
     *
     * @param p    the point to convert to an index.
     * @return the index of the black tile or -1 if the point is not a black
     * tile.
     * @see #toIndex(int, int)
     * @see #toPoint(int)
     */
    public static int toIndex(Point p) {
        return (p == null)? -1 : toIndex(p.x, p.y);
    }

    /**
     * Sets or clears the specified bit in the target value and returns
     * the updated value.
     *
     * @param target   the target value to update.
     * @param bit     the bit to update (from 0 to 31 inclusive).
     * @param set        true to set  the bit, false to clear the bit.
     * @return the updated target value with the bit set or cleared.
     * @see #getBit(int, int)
     */
    public static int setBit(int target, int bit, boolean set) {

        // Nothing to do
        if (bit < 0 || bit > 31) {
            return target;
        }

        // Set the bit
        if (set) {
            target |= (1 << bit);
        }

        // Clear the bit
        else {
            target &= (~(1 << bit));
        }

        return target;
    }

    /**
     * Gets the state of a bit and determines if it is set (1) or not (0).
     *
     * @param target   the target value to get the bit from.
     * @param bit     the bit to get (from 0 to 31 inclusive).
     * @return 1 if and only if the specified bit is set, 0 otherwise.
```

```
     * @see #setBit(int, int, boolean)
     */
    public static int getBit(int target, int bit) {

        // Out of range
        if (bit < 0 || bit > 31) {
            return 0;
        }

        return (target & (1 << bit)) != 0? 1 : 0;
    }

    /**
     * Gets the middle point on the checker board between two points.
     *
     * @param p1   the first point of a black tile on the checker board.
     * @param p2   the second point of a black tile on the checker board.
     * @return the middle point between two points or (-1, -1) if the points
     * are not on the board, are not distance 2 from each other in x and y,
     * or are on a white tile.
     * @see #middle(int, int)
     * @see #middle(int, int, int, int)
     */
    public static Point middle(Point p1, Point p2) {

        // A point isn't initialized
        if (p1 == null || p2 == null) {
            return new Point(-1, -1);
        }

        return middle(p1.x, p1.y, p2.x, p2.y);
    }

    /**
     * Gets the middle point on the checker board between two points.
     *
     * @param index1   the index of the first point (from 0 to 31 inclusive).
     * @param index2   the index of the second point (from 0 to 31 inclusive).
     * @return the middle point between two points or (-1, -1) if the points
     * are not on the board, are not distance 2 from each other in x and y,
     * or are on a white tile.
     * @see #middle(Point, Point)
     * @see #middle(int, int, int, int)
     */
    public static Point middle(int index1, int index2) {
        return middle(toPoint(index1), toPoint(index2));
    }

    /**
     * Gets the middle point on the checker board between two points.
     *
     * @param x1   the x-coordinate of the first point.
     * @param y1   the y-coordinate of the first point.
     * @param x2   the x-coordinate of the second point.
     * @param y2   the y-coordinate of the second point.
     * @return the middle point between two points or (-1, -1) if the points
     * are not on the board, are not distance 2 from each other in x and y,
     * or are on a white tile.
     * @see #middle(int, int)}
     * @see #middle(Point, Point)
     */
    public static Point middle(int x1, int y1, int x2, int y2) {

        // Check coordinates
        int dx = x2 - x1, dy = y2 - y1;
        if (x1 < 0 || y1 < 0 || x2 < 0 || y2 < 0 || // Not in the board
              x1 > 7 || y1 > 7 || x2 > 7 || y2 > 7) {
            return new Point(-1, -1);
```

```
      } else if (x1 % 2 == y1 % 2 || x2 % 2 == y2 % 2) { // white tile
         return new Point(-1, -1);
      } else if (Math.abs(dx) != Math.abs(dy) || Math.abs(dx) != 2) {
         return new Point(-1, -1);
      }

      return new Point(x1 + dx / 2, y1 + dy / 2);
   }

   /**
    * Checks if an index corresponds to a black tile on the checker board.
    *
    * @param testIndex    the index to check.
    * @return true if and only if the index is between 0 and 31 inclusive.
    */
   public static boolean isValidIndex(int testIndex) {
      return testIndex >= 0 && testIndex < 32;
   }

   /**
    * Checks if a point corresponds to a black tile on the checker board.
    *
    * @param testPoint    the point to check.
    * @return true if and only if the point is on the board, specifically on
    * a black tile.
    */
   public static boolean isValidPoint(Point testPoint) {

      if (testPoint == null) {
         return false;
      }

      // Check that it is on the board
      final int x = testPoint.x, y = testPoint.y;
      if (x < 0 || x > 7 || y < 0 || y > 7) {
         return false;
      }

      // Check that it is on a black tile
      return x % 2 != y % 2;

   }

   /**
    * Validates all ID related values for the startClick, end, and middle (if the
    * move is a skip).
    *
    * @param isP1Turn   the flag indicating if it is player 1's turn.
    * @param startIndex the startClick index of the move.
    * @param endIndex   the end index of the move.
    * @return true if and only if all IDs are valid.
    */
   private boolean validateIDs(boolean isP1Turn, int startIndex, int endIndex) {

      // Check if end is clear
      if (get(endIndex) != Board.EMPTY) {
         return false;
      }

      // Check if proper ID
      int id = get(startIndex);
      if ((isP1Turn && id != Board.BLACK_CHECKER && id != Board.BLACK_KING) ||
(!isP1Turn && id != Board.WHITE_CHECKER && id != Board.WHITE_KING)) {
         return false;
      }

      // Check the middle
      Point middle = Board.middle(startIndex, endIndex);
```

```
      int midID = get(Board.toIndex(middle));
      return midID == Board.INVALID || ((isP1Turn || midID == Board.BLACK_CHECKER ||
midID == Board.BLACK_KING) && (!isP1Turn || midID == Board.WHITE_CHECKER || midID ==
Board.WHITE_KING));

      // Passed all tests
   }

   /**
    * Checks that the move is diagonal and magnitude 1 or 2 in the correct
    * direction. If the magnitude is not 2 (i.e. not a skip), it checks that
    * no skips are available by other checkers of the same player.
    *
    * @param isP1Turn   the flag indicating if it is player 1's turn.
    * @param startIndex the startClick index of the move.
    * @param endIndex   the end index of the move.
    * @return true if and only if the move distance is valid.
    */
   private boolean validateDistance(boolean isP1Turn, int startIndex, int endIndex) {

      // Check that it was a diagonal move
      Point start = Board.toPoint(startIndex);
      Point end = Board.toPoint(endIndex);
      int dx = end.x - start.x;
      int dy = end.y - start.y;
      if (Math.abs(dx) != Math.abs(dy) || Math.abs(dx) > 2 || dx == 0) {
         return false;
      }

      // Check that it was in the right direction
      int id = get(startIndex);
      if ((id == Board.WHITE_CHECKER && dy > 0) || (id == Board.BLACK_CHECKER && dy <
0)) {
         return false;
      }

      // Check that if this is not a skip, there are none available
      Point middle = Board.middle(startIndex, endIndex);
      int midID = get(Board.toIndex(middle));
      if (midID < 0) {

         // Get the correct checkers
         List<Point> checkers;
         if (isP1Turn) {
            checkers = find(Board.BLACK_CHECKER);
            checkers.addAll(find(Board.BLACK_KING));
         } else {
            checkers = find(Board.WHITE_CHECKER);
            checkers.addAll(find(Board.WHITE_KING));
         }

         // Check if any of them have a skip available
         for (Point p : checkers) {
            int index = Board.toIndex(p);
            if (!getPieceSkips(index).isEmpty()) {
               return false;
            }
         }
      }

      // Passed all tests
      return true;
   }

   /**
    * Checks if the specified checker is safe (i.e. the opponent cannot skip
    * the checker).
    *
```

```
     * @param checker the point where the test checker is located at.
     * @return true if and only if the checker at the point is safe.
     */
    public boolean isSafe(Point checker) {

        // Trivial cases
        if (checker == null) {
            return true;
        }
        int index = Board.toIndex(checker);
        if (index < 0) {
            return true;
        }
        int id = get(index);
        if (id == Board.EMPTY) {
            return true;
        }

        // Determine if it can be skipped
        boolean isBlack = (id == Board.BLACK_CHECKER || id == Board.BLACK_KING);
        List<Point> check = new ArrayList<>();
        addPoints(check, checker, Board.BLACK_KING, 1);
        for (Point p : check) {
            int start = Board.toIndex(p);
            int tid = get(start);

            // Nothing here
            if (tid == Board.EMPTY || tid == Board.INVALID) {
                continue;
            }

            // Check ID
            boolean isWhite = (tid == Board.WHITE_CHECKER || tid == Board.WHITE_KING);
            if (isBlack && !isWhite) {
                continue;
            }
            boolean isKing = (tid == Board.BLACK_KING || tid == Board.BLACK_KING);

            // Determine if valid skip direction
            int dx = (checker.x - p.x) * 2;
            int dy = (checker.y - p.y) * 2;
            if (!isKing && (isWhite ^ (dy < 0))) {
                continue;
            }
            int endIndex = Board.toIndex(new Point(p.x + dx, p.y + dy));
            if (isValidSkip(start, endIndex)) {
                return false;
            }
        }

        return true;
    }

    /**
     * Gets a list of move end-points for a given startClick index.
     *
     * @param start the center index to look for moves around.
     * @return the list of points such that the startClick to a given point
     * represents a move available.
     * @see #getPieceMoves(int)
     */
    public List<Point> getPieceMoves(Point start) {
        return getPieceMoves(Board.toIndex(start));
    }

    /**
     * Gets a list of move end-points for a given startClick index.
     *
```

```
     * @param startIndex the center index to look for moves around.
     * @return the list of points such that the startClick to a given point
     * represents a move available.
     * @see #getPieceMoves(Point)
     */
    public List<Point> getPieceMoves(int startIndex) {

        // Trivial cases
        List<Point> endPoints = new ArrayList<>();
        if (!Board.isValidIndex(startIndex)) {
            return endPoints;
        }

        // Determine possible points
        int id = get(startIndex);
        Point p = Board.toPoint(startIndex);
        addPoints(endPoints, p, id, 1);

        // Remove invalid points
        for (int i = 0; i < endPoints.size(); i++) {
            Point end = endPoints.get(i);
            if (get(end.x, end.y) != Board.EMPTY) {
                endPoints.remove(i--);
            }
        }

        return endPoints;
    }

    /**
     * Gets a list of skip end-points for a given starting point.
     *
     * @param start the center index to look for skips around.
     * @return the list of points such that the startClick to a given point
     * represents a skip available.
     * @see #getPieceSkips(int)
     */
    public List<Point> getPieceSkips(Point start) {
        return getPieceSkips(Board.toIndex(start));
    }

    /**
     * Gets a list of skip end-points for a given startClick index.
     *
     * @param startIndex the center index to look for skips around.
     * @return the list of points such that the startClick to a given point
     * represents a skip available.
     * @see #getPieceSkips(Point)
     */
    public List<Point> getPieceSkips(int startIndex) {

        // Trivial cases
        List<Point> endPoints = new ArrayList<>();
        if (!Board.isValidIndex(startIndex)) {
            return endPoints;
        }

        // Determine possible points
        int id = get(startIndex);
        Point p = Board.toPoint(startIndex);
        addPoints(endPoints, p, id, 2);

        // Remove invalid points
        for (int i = 0; i < endPoints.size(); i++) {

            // Check that the skip is valid
            Point end = endPoints.get(i);
            if (!isValidSkip(startIndex, Board.toIndex(end))) {
```

```
            endPoints.remove(i--);
         }
      }



      return endPoints;
   }

   /**
    * Checks if a skip is valid.
    *
    * @param startIndex the startClick index of the skip.
    * @param endIndex   the end index of the skip.
    * @return true if and only if the skip can be performed.
    */
   public boolean isValidSkip(int startIndex, int endIndex) {

      // Check that end is empty
      if (get(endIndex) != Board.EMPTY) {
         return false;
      }

      // Check that middle is enemy
      int id = get(startIndex);
      int midID = get(Board.toIndex(Board.middle(startIndex, endIndex)));

      //Check if starting e middle position are valid and not empty
      if (id == Board.INVALID || id == Board.EMPTY) return false;
      if (midID == Board.INVALID || midID == Board.EMPTY) return false;

      //Check that midID is an enemy for id
      if ((id == Board.WHITE_KING || id == Board.WHITE_CHECKER) && (midID ==
Board.WHITE_KING || midID == Board.WHITE_CHECKER))
         return false;
      if ((id == Board.BLACK_KING || id == Board.BLACK_CHECKER) && (midID ==
Board.BLACK_KING || midID == Board.BLACK_CHECKER))
         return false;

      //Check that skip is not performed by a normal checkers versus a king
      return (id != Board.WHITE_CHECKER || midID != Board.BLACK_KING) && (id !=
Board.BLACK_CHECKER || midID != Board.WHITE_KING);

   }

   /**
    * Adds points that could potentially result in moves/skips.
    *
    * @param points the list of points to add to.
    * @param p      the center point.
    * @param id     the ID at the center point.
    * @param delta  the amount to add/subtract.
    */
   public static void addPoints(List<Point> points, Point p, int id, int delta) {

      // Add points moving down
      boolean isKing = (id == Board.BLACK_KING || id == Board.WHITE_KING);
      if (isKing || id == Board.BLACK_CHECKER) {
         points.add(new Point(p.x + delta, p.y + delta));
         points.add(new Point(p.x - delta, p.y + delta));
      }

      // Add points moving up
      if (isKing || id == Board.WHITE_CHECKER) {
         points.add(new Point(p.x + delta, p.y - delta));
         points.add(new Point(p.x - delta, p.y - delta));
      }
   }
```

```
   @Override
   public String toString() {
      StringBuilder obj = new StringBuilder(getClass().getName() + "[");
      for (int i = 0; i < 31; i ++) {
         obj.append(get(i)).append(", ");
      }
      obj.append(get(31));

      return obj + "]";
   }

   public int getRows() {
      return nRows;
   }

   public int getCols() {
      return nCols;
   }
}
```

## 5.4 GameManager

```
package com.dca.checkers.model;

import com.dca.checkers.ui.CheckerBoard;
import com.dca.checkers.ui.OptionPanel;

import java.awt.*;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

/**
 * The {@code GameManager} represents a sort of referee for a Checker game.
 * It is the joint point between the UI and logic part of the application and it runs on
its dedicated thread.
 */
public class GameManager extends Thread {

   /**
    * Reference to thread instantiated for GameManager works.
    */
   private Thread t;

   /**
    * The player in control of the black checkers.
    */
   private Player player1;

   /**
    * The player in control of the white checkers.
    */
   private Player player2;

   /**
    * Board boardUI reference
    */
   private CheckerBoard boardUI;

   /**
    * Current game state managed
    */
   private GameState gameState;

   /**
    * Tells if game is paused
```

```
     */
    private boolean isPaused;

    /**
     * Tells if game is ready to start
     */
    private boolean isReadyToStart;

    /**
     * Tells if game is on going
     */
    private boolean isOnGoing;

    /**
     * Tells if game is over
     */
    private boolean isOver;

    /**
     * Option panel r
     */
    private OptionPanel opt;

    /**
     * The amount of milliseconds before a computer player takes a move.
     */
    private int AIDelay = 1000;

    /**
     * The history of the game
     */
    private List<GameState> history;

    /**
     * Track the current game state position in history
     */
    private int curHistoryIndex;

    /**
     * Track the last valid index in history.
     * When a some undos are performed and a new move is taken, all
     * state saved in history are no more useful.
     */
    private int lastIndexValid;

    public GameManager(GameState gameState, CheckerBoard boardUI, OptionPanel opt) {
        this.gameState = new GameState();
        this.player1 = opt.getPlayer1();
        this.player2 = opt.getPlayer2();
        this.boardUI = boardUI;
        this.gameState = gameState == null ? new GameState() : gameState;
        this.opt = opt;
        this.isPaused = false;
        this.isReadyToStart = true;
        this.isOnGoing = false;
        this.isOver = false;
        this.curHistoryIndex = 0;
        this.lastIndexValid = 0;
        this.history = new ArrayList<>();
        this.history.add(this.gameState.copy());
        //Set UI
        updateUI();
    }

    @Override
    public void run() {
        System.out.println("Running game manager thread");
```

```
        while (true) {
            waitStart();
            handleGameplay();
            //setUIReadyToStart();
        }

    }

    @Override
    public void start() {
        if (t == null) {
            t = new Thread(this);
            t.start();
        }
    }

    /** Handle the game until it's over. */
    public void handleGameplay() {
        Player currentPlayer;
        while (!gameState.isGameOver()) {
            //If game is paused wait
            waitResume();
            currentPlayer = getCurrentPlayer();
            //Write to console who must take next move
            if (gameState.isP1Turn()) writeToConsole("It's Player 1's turn.");
            else writeToConsole("It's Player 2's turn.");
            //Wait only if next to move is a computer player
            if (!currentPlayer.isHuman()) {
                System.out.println("Current player is not human! Wait a bit ...");
                try {
                    Thread.sleep(AIDelay);
                } catch (InterruptedException e) {
                    System.err.println("An error occurred during sleep.\n");
                    e.printStackTrace();
                }
            }
            currentPlayer.updateGame(gameState);
            waitPlayerChoice(currentPlayer);
            if (currentPlayer.hasMoved()) addHistory(gameState.copy());
            updateUI();
        }
        gameOver();
    }

    /**
     * Add the game state g to the game history.
     * @param g the game state to save in history.
     * */
    private void addHistory(GameState g) {
        if (curHistoryIndex < history.size()) {
            history.add(++curHistoryIndex, g);
        } else { //curHistoryIndex == history.size()
            history.add(g);
            curHistoryIndex++;
        }
        lastIndexValid = curHistoryIndex;
    }

    /**
     * Check if it's currently possible to perform an undo.
     *
     * @return true if it's possible, false otherwise.
     */
    private boolean undoIsPossible() {
        return curHistoryIndex > 0;
    }

    /**
```

```
     * Check if it's currently possible to perform a redo.
     *
     * @return true if it's possible, false otherwise.
     */
    private boolean redoIsPossible() {
        return curHistoryIndex < lastIndexValid;
    }

    /**
     * Redo the last move if any is available.
     */
    public void redo() {
        if (redoIsPossible()) {
            gameState.setGameState(history.get(++curHistoryIndex).getGameState());
            updateUI();
        }

    }

    /**
     * Undo the last move if any is available.
     */
    public void undo() {
        if (undoIsPossible()) {
            gameState.setGameState(history.get(--curHistoryIndex).getGameState());
            updateUI();
        }
    }

    /**
     * Wait the game start.
     */
    synchronized private void waitStart() {
        while (!isOnGoing) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     * Wait until a player has taken a decision for his turn (skip or move).
     *
     * @param player the player to wait.
     */
    synchronized private void waitPlayerChoice(Player player) {
        while (!(player.hasMoved() || player.hasSkipped())) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    /** Wait the game resume. */
    private synchronized void waitResume() {
        while (!isOnGoing) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```java
/**
 * Set the player 1.
 *
 * @param player1 the new player 1.
 */
public void setPlayer1(Player player1) {
   System.out.println("Player 1 set.");
   this.player1 = (player1 == null) ? new HumanPlayer() : player1;
   if (gameState.isP1Turn() && !this.player1.isHuman()) {
      boardUI.cancelLastSelection();
   }
   writeToConsole("Player 1 type changed.");
   boardUI.repaint();
}

/**
 * Set the player 2.
 * @param player2 the new player 2.
 */
public void setPlayer2(Player player2) {
   System.out.println("Player 2 setted.");
   this.player2 = (player2 == null) ? new HumanPlayer() : player2;
   if (!gameState.isP1Turn() && !this.player2.isHuman()) {
      boardUI.cancelLastSelection();
   }
   writeToConsole("Player 2 type changed.");
   boardUI.repaint();
}

/**
 * Return the next player to play.
 * @return the player who must take a decision.
 */
public Player getCurrentPlayer() {
   if (gameState.isP1Turn()) return player1;
   else return player2;
}

/**
 * Handles a click performed on the board. If the current
 * player is not human, this method does nothing. Otherwise, current human player is
 * infomed about the event.
 *
 * @param sel the selected point on the board.
 */
synchronized public void handleBoardClick(Point sel) {
   // The gameState is over or the current player isn't human
   if (!isOnGoing || gameState.isGameOver() || !getCurrentPlayer().isHuman()) {
      return;
   }
   HumanPlayer currentPlayer = (HumanPlayer) getCurrentPlayer();
   //Communicate to the current human player object the selection on the board
   currentPlayer.handleBoardClick(gameState, boardUI, sel);
   updateUI();
   notifyAll();
}

/**
 * If boardUI is available, update it
 */
private void updateUI() {
   if (isPaused) setUIPaused();
   if (isReadyToStart) setUIReadyToStart();
   if (isOnGoing) setUIOnGoing();
   if (isOver) setUIOver();
   boardUI.repaint();
}
```

```
    /**
     * Request to resetClick the game
     */
    synchronized public void resetClick() {
        writeToConsole("Board reset done.");
        this.isPaused = false;
        this.isReadyToStart = true;
        this.isOnGoing = false;
        this.isOver = false;
        this.gameState.restart();
        this.curHistoryIndex = 0;
        this.history = new ArrayList<>();
        this.history.add(gameState.copy());
        this.lastIndexValid = 0;
        updateUI();
    }

    /**
     * Request to start the game
     */
    synchronized public void startClick() {
        writeToConsole("Game started.");
        this.isPaused = false;
        this.isReadyToStart = false;
        this.isOnGoing = true;
        player1 = opt.getPlayer1();
        player2 = opt.getPlayer2();
        updateUI();
        notifyAll();
    }

    /**
     * Request to resume the paused game
     */
    synchronized public void resumeClick() {
        writeToConsole("Game resumed.");
        this.isPaused = false;
        this.isReadyToStart = false;
        this.isOnGoing = true;
        player1 = opt.getPlayer1();
        player2 = opt.getPlayer2();
        updateUI();
        notifyAll();
    }

    /**
     * Request to pause the current game
     */
    synchronized public void pauseClick() {
        writeToConsole("Game pausing...");
        this.isPaused = true;
        this.isReadyToStart = false;
        this.isOnGoing = false;
        //If the current player is a Human, skip the wait for his move
        Player current = getCurrentPlayer();
        if (current.isHuman()) {
            ((HumanPlayer) current).skipNextMove();
        }
        notifyAll();
    }

    /**
     * Setup for game over state.
     */
    synchronized public void gameOver() {
        writeToConsole("Game over.");
        this.isPaused = false;
        this.isReadyToStart = false;
```

```
        this.isOnGoing = false;
        this.isOver = true;
        updateUI();
    }

    /** Setup UI for OnGoing state. */
    synchronized private void setUIOnGoing() {
        opt.cmbPlayer1Type.setEnabled(false);
        opt.cmbPlayer2Type.setEnabled(false);
        opt.btnStart.setEnabled(false);
        opt.btnResume.setEnabled(false);
        opt.btnPause.setEnabled(true);
        opt.btnRest.setEnabled(false);
        opt.btnUndo.setEnabled(false);
        opt.btnRedo.setEnabled(false);
    }

    /** Setup UI for ReadyToStart state. */
    synchronized private void setUIReadyToStart() {
        writeToConsole("Press 'Start' to start a game.");
        opt.cmbPlayer1Type.setEnabled(true);
        opt.cmbPlayer2Type.setEnabled(true);
        opt.btnStart.setEnabled(true);
        opt.btnResume.setEnabled(false);
        opt.btnPause.setEnabled(false);
        opt.btnRest.setEnabled(false);
        opt.btnUndo.setEnabled(false);
        opt.btnRedo.setEnabled(false);
    }

    /** Setup UI for Paused state. */
    synchronized private void setUIPaused() {
        writeToConsole("Game is paused.");
        opt.cmbPlayer1Type.setEnabled(true);
        opt.cmbPlayer2Type.setEnabled(true);
        opt.btnStart.setEnabled(false);
        opt.btnResume.setEnabled(true);
        opt.btnPause.setEnabled(false);
        opt.btnRest.setEnabled(true);
        opt.btnUndo.setEnabled(undoIsPossible());
        opt.btnRedo.setEnabled(redoIsPossible());
    }

    /** Setup UI for Over state. */
    synchronized private void setUIOver() {
        writeToConsole("Game is over.");
        opt.cmbPlayer1Type.setEnabled(true);
        opt.cmbPlayer2Type.setEnabled(true);
        opt.btnStart.setEnabled(false);
        opt.btnResume.setEnabled(false);
        opt.btnPause.setEnabled(false);
        opt.btnRest.setEnabled(true);
    }

    /**
     * Write a message in console.
     * @param msg the message to append.
     */
    private void writeToConsole(String msg) {
        String date = new SimpleDateFormat("hh:mm:ss").format(new Date());
        opt.txtAreaConsole.append("[" + date + "]: " + msg + "\n");
    }

    /**
     * Set delay for a AI move.
     * @param value the new value for AI delay.
     */
    public void setDelay(int value) {
```

```
        AIDelay = value;
    }

}
```

## 5.5 GameState

```
package com.dca.checkers.model;


import java.awt.*;
import java.util.ArrayList;
import java.util.List;

/**
 * The {@code GameState} class represents a game of checkers and ensures that all
 * moves made are valid as per the rules of checkers.
 */
public class GameState implements State {


    /** The current state of the checker board. */
    private Board board;

    /** The flag indicating if it is player 1's turn. */
    private boolean isP1Turn;

    /** The index of the last skip, to allow for multiple skips in a turn. */
    private int skipIndex;

    /**
     * Number of moves excecuted by p1 from last skip ex
     */
    private int cntMovesFromLastSkip;

    /**
     * Max number of moves without a skip required to declare a draw
     */
    private final int maxNumMovesForDraw = 40;

    /**
     * Flag that tells if current state is a draw.
     */
    private boolean draw;

    public GameState() {
        restart();
    }

    public GameState(String state) {
        setGameState(state);
    }

    public GameState(Board board, boolean isP1Turn, int skipIndex) {
        this.board = (board == null)? new Board() : board;
        this.isP1Turn = isP1Turn;
        this.skipIndex = skipIndex;
    }

    /**
     * Creates a copy of this game such that any modifications made to one are
     * not made to the other.
     *
     * @return an exact copy of this game.
     */
    public GameState copy() {
        GameState g = new GameState();
```

```
        g.board = board.copy();
        g.isP1Turn = isP1Turn;
        g.skipIndex = skipIndex;
        g.cntMovesFromLastSkip = cntMovesFromLastSkip;
        g.draw = draw;
        return g;
    }

    /**
     * Resets the game of checkers to the initial state.
     */
    public void restart() {
        this.board = new Board();
        this.isP1Turn = false;
        this.skipIndex = -1;
        this.cntMovesFromLastSkip = 0;
        this.draw = false;
    }

    /**
     * Attempts to make a move from the startClick point to the end point.
     *
     * @param start    the startClick point for the move.
     * @param end  the end point for the move.
     * @return true if and only if an update was made to the game state.
     * @see #move(int, int)
     */
    public boolean move(Point start, Point end) {
        if (start == null || end == null) {
            return false;
        }
        return move(Board.toIndex(start), Board.toIndex(end));
    }

    /**
     * Attempts to make a move given the startClick and end index of the move.
     *
     * @param startIndex    the startClick index of the move.
     * @param endIndex    the end index of the move.
     * @return true if and only if an update was made to the game state.
     * @see #move(Point, Point)
     */
    public boolean move(int startIndex, int endIndex) {

        // Validate the move
        Move m = getMove(startIndex, endIndex);
        if (m == null) //Invalid move!
            return false;

        // Make the move
        Point middle = Board.middle(startIndex, endIndex);
        int midIndex = Board.toIndex(middle);
        this.board.set(endIndex, board.get(startIndex));
        this.board.set(midIndex, Board.EMPTY);
        this.board.set(startIndex, Board.EMPTY);

        // Make the checker a king if necessary
        Point end = Board.toPoint(endIndex);
        int id = board.get(endIndex);
        boolean switchTurn = false;
        if (end.y == 0 && id == Board.WHITE_CHECKER) {
            this.board.set(endIndex, Board.WHITE_KING);
            switchTurn = true;
        } else if (end.y == 7 && id == Board.BLACK_CHECKER) {
            this.board.set(endIndex, Board.BLACK_KING);
            switchTurn = true;
        }
```

```
        // Check if the turn should switch (i.e. no more skips)
        boolean midValid = Board.isValidIndex(midIndex);
        if (midValid) {
            this.skipIndex = endIndex;
        }
        if (!midValid || board.copy().getPieceSkips(endIndex).isEmpty()) {
            switchTurn = true;
        }
        //Handle draw check
        if (!draw) {//Draw not declared yet
            if (hasKing()) {
                if (m.getType() == MoveType.SKIP) cntMovesFromLastSkip = 0;
                else draw = (++cntMovesFromLastSkip) >= maxNumMovesForDraw;
            }
        }
        if (switchTurn) {
            this.isP1Turn = !isP1Turn;
            this.skipIndex = -1;
        }

        return true;
    }

    /**
     * Get number of normal moves (no skip) left to reach a draw.
     * @return the number of normal moves (no skip) left to reach a draw.
     */
    public int getNumMovesBeforeDraw() {
        return maxNumMovesForDraw - cntMovesFromLastSkip;
    }

    /**
     * Get the move (startIndex, endIndex) if exists.
     */
    private Move getMove(int startIndex, int endIndex) {
        List<Move> moves = getAllMoves();
        for (Move m : moves) {
            if (m.getStartIndex() == startIndex && m.getEndIndex() == endIndex) return m;
        }
        return null;
    }

    /**
     * Check if at least one king is present on the board.
     * @return true if at least one king is present on the board, otherwise false.
     */
    public boolean hasKing() {
        List<Point> pieces = getPlayerPieces(true);
        pieces.addAll(getPlayerPieces(false));
        int id;
        for (Point p : pieces) {
            id = board.get(Board.toIndex(p));
            if (id == Board.WHITE_KING || id == Board.BLACK_KING) return true;
        }
        return false;
    }

    /**
     * Gets a copy of the current board state.
     *
     * @return a non-reference to the current game board state.
     */
    public Board getBoard() {
        return board.copy();
    }

    /**
     * Determines if the game is over.
```

```
    *
    * @return true if the game is over.
    */
   public boolean isGameOver() {
      return getResult() != MatchResult.UNKNOWN;
   }

   /**
    * Get the current game result.
    * @return the current game result.
    */
   public MatchResult getResult() {
      if (isDraw()) return MatchResult.DRAW;
      if (currentPlayerCanMove()) return MatchResult.UNKNOWN;
      return isP1Turn ? MatchResult.P2_WIN : MatchResult.P1_WIN;
/*    // Ensure there is at least one of each checker
      List<Point> black = board.find(Board.BLACK_CHECKER);
      black.addAll(board.find(Board.BLACK_KING));

      List<Point> white = board.find(Board.WHITE_CHECKER);
      white.addAll(board.find(Board.WHITE_KING));

      if (white.isEmpty() && black.isEmpty())
         return MatchResult.UNKNOWN;

      //Now on, at least one of two player must have at least one piece

      if(white.isEmpty())
         return  MatchResult.P2_WIN;

      if(black.isEmpty())
         return  MatchResult.P1_WIN;

      //Both the player have at least one piece

      // If the current player can move => game is NOT over
      if(currentPlayerCanMove())  return MatchResult.UNKNOWN;


      // Current players has no moves => Opponent wins
      return isP1Turn ? MatchResult.P2_WIN:MatchResult.P1_WIN;*/

   }

   /** Check if a draw is occurred. */
   private boolean isDraw() {
      return draw;
   }

   /**
    * Check if the current player can move. I other words, he must have at least one
piece on the board
    * and at least one of them must have one possible move.
    * @return true if the current player can move: false othrwise.
    */
   private boolean currentPlayerCanMove() {
      return !getAllMoves().isEmpty();
//    //Get current player pieces
//    List<Point> pieces = getPlayerPieces(isP1Turn);
//
//    for (Point p : pieces) {
//       int i = Board.toIndex(p);
//       if (!board.getPieceMoves(i).isEmpty() || !board.getPieceSkips(i).isEmpty())
//          return true;
//    }
//    return false;
   }
```

```java
/**
 * Determines if the specified move is valid based on the rules of checkers.
 *
 * @param start the startClick point of the move.
 * @param end   the end point of the move.
 * @return true if the move is legal according to the rules of checkers.
 */
public boolean isValidMove(Point start, Point end) {
   return isValidMove(Board.toIndex(start), Board.toIndex(end));
}

/**
 * Determines if the specified move is valid based on the rules of checkers.
 *
 * @param startIndex the startClick index of the move.
 * @param endIndex   the end index of the move.
 * @return true if the move is legal according to the rules of checkers.
 */
public boolean isValidMove(int startIndex, int endIndex) {
   List<Move> allMoves = getAllMoves();
   for (Move m : allMoves)
      if(m.getStartIndex() == startIndex && m.getEndIndex() == endIndex) return true;

   return false;
   //return board.isValidMove(isP1Turn(), startIndex, endIndex, getSkipIndex());
}

/**
 * Check if it's player 1 turn.
 * @return true if is Player 1 turn, false otherwise.
 */
public boolean isP1Turn() {
   return isP1Turn;
}

/**
 * Set if it's player 1 turn or not.
 * @param isP1Turn the flag to use to set the turn.
 */
public void setP1Turn(boolean isP1Turn) {
   this.isP1Turn = isP1Turn;
}

/**
 * Gets all the available moves and skips for the current player.
 *
 * @return a list of valid moves that the player can make.
 */
public List<Move> getAllMoves() {

   // The next move needs to be a skip
   if (getSkipIndex() >= 0) {

      List<Move> moves = new ArrayList<>();
      List<Point> skips = getSkips(getSkipIndex());
      for (Point end : skips) {
         moves.add(new Move(getSkipIndex(), Board.toIndex(end), MoveType.SKIP));
      }

      return moves;
   }

   // Get the checkers
   List<Point> checkers = new ArrayList<>();
   Board b = getBoard();
   if (isP1Turn()) {
      checkers.addAll(b.find(Board.BLACK_CHECKER));
      checkers.addAll(b.find(Board.BLACK_KING));
```

```
        } else {
            checkers.addAll(b.find(Board.WHITE_CHECKER));
            checkers.addAll(b.find(Board.WHITE_KING));
        }

        // Determine if there are any skips
        List<Move> moves = new ArrayList<>();
        for (Point checker : checkers) {
            int index = Board.toIndex(checker);
            List<Point> skips = getSkips(index);
            for (Point end : skips) {
                Move m = new Move(index, Board.toIndex(end), MoveType.SKIP);
                moves.add(m);
            }
        }

        if (moves.isEmpty()) { //No skips found
            // There are no skips, add the regular moves
            for (Point checker : checkers) {
                int index = Board.toIndex(checker);
                List<Point> movesEnds = b.getPieceMoves(index);
                for (Point end : movesEnds) {
                    moves.add(new Move(index, Board.toIndex(end), MoveType.NORMAL));
                }
            }
        }

        return moves;
    }

    /**
     * Gets all the available moves starting from startIndex.
     * @param startIndex the start index.
     * @return a list of valid moves that the player can make with piece in startIndex.
     */
    public List<Move> getAllMoves(int startIndex) {
        List<Move> moves = getAllMoves();
        for (int i = 0; i < moves.size(); i++) {
            if (moves.get(i).getStartIndex() != startIndex) moves.remove(i--);
        }
        return moves;
    }

    /**
     * Check if the selected tiles startIndex has at least one move
     * @param startIndex the start index.
     * @return true if piece on tile startIndex has at least one move.
     */
    public boolean hasMove(int startIndex) {
        return getAllMoves(startIndex).size() > 0;
    }

    /**
     * Check if the point p has at least one move available.
     * @param p the point on the board to check.
     * @return true if piece on tile startIndex has at least one move.
     */
    public boolean hasMove(Point p) {
        return hasMove(Board.toIndex(p));
    }

    /**
     * Gets the number of skips that can be made in one turn from a given startClick
     * index.
     *
     * @param startIndex    the startClick index of the skips.
     * @param isP1Turn    the original player turn flag.
     * @return the maximum number of skips available from the given point.
```

```
     */
    private int getSkipDepth(int startIndex, boolean isP1Turn) {

        // Trivial case
        if (isP1Turn != isP1Turn()) {
            return 0;
        }

        // Recursively get the depth
        List<Point> skips = getSkips(startIndex);
        int depth = 0;
        for (Point end : skips) {
            int endIndex = Board.toIndex(end);
            move(startIndex, endIndex);
            int testDepth = getSkipDepth(endIndex, isP1Turn);
            if (testDepth > depth) {
                depth = testDepth;
            }
        }

        return depth + (skips.isEmpty()? 0 : 1);
    }

    /**
     * Gets a list of skip end-points for a given startClick index.
     *
     * @param startIndex the center index to look for skips around.
     * @return the list of points such that the startClick to a given point
     * represents a skip available.
     */
    public List<Point> getSkips(int startIndex) {
        return board.getPieceSkips(startIndex);
    }

    /**
     * Get the index of last skip.
     * @return the index of last skip.
     */
    public int getSkipIndex() {
        return skipIndex;
    }

    /**
     * Gets the current game state as a string of data that can be parsed by
     * {@link #setGameState(String)}.
     *
     * @return a string representing the current game state.
     * @see #setGameState(String)
     */
    public String getGameState() {

        // Add the game board
        StringBuilder stateBuilder = new StringBuilder();
        for (int i = 0; i < 32; i++) {
            stateBuilder.append(board.get(i));
        }
        String state = stateBuilder.toString();

        // Add the other info
        state += (isP1Turn? "1" : "0");
        state += skipIndex;

        return state;
    }

    /**
     * Parses a string representing a game state that was generated from
     * {@link #getGameState()}.
```

```
     *
     * @param state    the game state.
     * @see #getGameState()
     */
    public void setGameState(String state) {

        restart();

        // Trivial cases
        if (state == null || state.isEmpty()) {
            return;
        }

        // Update the board
        int n = state.length();
        for (int i = 0; i < 32 && i < n; i ++) {
            try {
                int id = Integer.parseInt("" + state.charAt(i));
                this.board.set(i, id);
            } catch (NumberFormatException e) {
                System.err.println("Impossible to parse character: " + i);
            }
        }

        // Update the other info
        if (n > 32) {
            this.isP1Turn = (state.charAt(32) == '1');
        }
        if (n > 33) {
            try {
                this.skipIndex = Integer.parseInt(state.substring(33));
            } catch (NumberFormatException e) {
                this.skipIndex = -1;
            }
        }
    }

    /**
     * Static evaluation of the current state from player 1 perspective if evalForP1 ==
true; otherwise
     * eval it from player 2 perspective.
     * @param evalForP1 the flag used to decide if current state must be evaluated for
player 1 or player 2.
     */
    @Override
    public double value(boolean evalForP1) {
        //GameState is not over
        //if(isEndingPhase())
        // return endStateValue1(evalForP1);
        //else
            return stateValue1(evalForP1);
    }

    /**
     * Tell if the game is in its final phase.
     * In others words, it tells if on the board are present only kings.
     * @return true if the game is ending; false otherwise.
     */
    private boolean isEndingPhase() {
        List <Point> checkers;
        checkers = board.find(Board.BLACK_CHECKER);
        if(checkers.size() > 0) return false;
        checkers = board.find(Board.WHITE_CHECKER);
        return checkers.size() <= 0;
    }

    /**
     * Counts the value of player's pieces and subtracts from it
```

```
    * the value of opponent's pieces.
    * @param evalForP1 flag that tells if current game state must be evaluated for
player 1 (true) or player 2 (false).
    * @return current state game value for player 1 or player 2.
    */
   private double stateValue1(boolean evalForP1) {
      double value = 0;
      final double W_CHECKER = 1;
      final double W_KING = 2;

      if(evalForP1) {
         //Number of pieces
         value += board.find(Board.BLACK_CHECKER).size() * W_CHECKER;
         value += board.find(Board.BLACK_KING).size() * W_KING;
         value -= board.find(Board.WHITE_CHECKER).size() * W_CHECKER;
         value -= board.find(Board.WHITE_KING).size() * W_KING;
      } else {//Eval for P2
         value += board.find(Board.WHITE_CHECKER).size() * W_CHECKER;
         value += board.find(Board.WHITE_KING).size() * W_KING;
         value -= board.find(Board.BLACK_CHECKER).size() * W_CHECKER;
         value -= board.find(Board.BLACK_KING).size() * W_KING;
      }

      return value;
   }

   /**
    * Advanced pawns are more threatening than pawns that are on the back of the board.
    * Therefore, since advanced pawns are much closer to become Kings, they got extra
value.
    * Of course, kings are still evaluated more than any pawn.
    *
    * @param evalForP1 flag that tells if current game state must be evaluated for
player 1 (true) or player 2 (false).
    * @return current state game value for player 1 or player 2.
    */
   private double stateValue2(boolean evalForP1) {
      double value;
      final double W_CHECKER_PLAYER_SIDE = 5;
      final double W_CHECKER_OPPONENT_SIDE = 7;
      final double W_KING = 10;
      List<Point> kings;
      List<Point> checkers;
      int countPlayerSides = 0;
      int countOpponentSide = 0;

      if(evalForP1) {
         kings = board.find(Board.BLACK_KING);
         value = kings.size() * W_KING;
         checkers = board.find(Board.BLACK_CHECKER);

         for (Point p : checkers) {
            if(Board.toIndex(p) >= 16)
               countOpponentSide++;
            else
               countPlayerSides++;
         }
         value += countOpponentSide * W_CHECKER_OPPONENT_SIDE;
         value += countPlayerSides * W_CHECKER_PLAYER_SIDE;
      } else {//Eval for P2
         kings = board.find(Board.WHITE_KING);
         value = kings.size() * W_KING;
         checkers = board.find(Board.WHITE_CHECKER);
         for (Point p : checkers) {
            if(Board.toIndex(p) < 16) countOpponentSide++;
            else  countPlayerSides++;
         }
         value += countOpponentSide * W_CHECKER_OPPONENT_SIDE;
```

```
            value += countPlayerSides * W_CHECKER_PLAYER_SIDE;
        }

        return value;
    }

    /**
     * For each piece (king) of the player we sum all the distances between it and all
the opponent's pieces. If the
     * player has more kings than the opponent will prefer a game position that minimizes
this sum (he wants to
     * attack), otherwise he will prefer this sum to be as big as possible (run away).
     *
     * @param evalForP1 flag that tells if current game state must be evaluated for
player 1 (true) or player 2 (false).
     * @return current state game value for player 1 or player 2.
     */
    private double endStateValue1(boolean evalForP1) {
        //Get pieces
        List<Point> playerPieces = getPlayerPieces(evalForP1);
        List<Point> opponentPieces = getPlayerPieces(!evalForP1);
        double distanceOverall = 0;
        //Calculate overall distance
        for (Point cP : playerPieces) {
            for (Point oP : opponentPieces) {
                distanceOverall += cP.distance(oP);
            }
        }
        //Check if current player has more pieces
        double maxDistance = Math.sqrt(Math.pow(board.getRows(),2) +
Math.pow(board.getCols(),2));
        if(playerPieces.size()  > opponentPieces.size()) {
            //Current player has more pieces, so he should aim to minimize the distance
            return (maxDistance * 12 * 12) - (distanceOverall);
        }else {
            //Current player has less pieces, so he should aim to maximise the distance
            return distanceOverall;
        }
    }

    /**
     * Get list of point on the board corresponding to pieces of player 1 if isP1 ==
true; otherwise for player 2.
     * @return the list of point on the board of the indicated player.
     */
    private List<Point> getPlayerPieces(boolean isP1) {
        List<Point> pieces;
        if(isP1) {
            pieces = board.find(Board.WHITE_CHECKER);
            pieces.addAll(board.find(Board.WHITE_KING));
        } else{ //Player 2 turn
            pieces = board.find(Board.BLACK_CHECKER);
            pieces.addAll(board.find(Board.BLACK_KING));
        }
        return pieces;
    }


}
```

## 5.6 HumanPlayer

```
package com.dca.checkers.model;

import com.dca.checkers.ui.CheckerBoard;
import com.dca.checkers.ui.CheckersWindow;
```

```java
import java.awt.*;

/**
 * The {@code HumanPlayer} class represents a user of the checkers game that
 * can update the game by clicking on tiles on the board.
 */
public class HumanPlayer implements Player {

   /**
    * Flag that tells if the move has been selected by the user
    */
   private boolean moveSelected;

   /** Flag that tells if the current turn must be skipped (no more wait for input) */
   private boolean skipMove;

   @Override
   public boolean isHuman() {
      return true;
   }

   @Override
   synchronized public void updateGame(GameState gameState) {
      moveSelected = false;
      skipMove = false;
   }

   @Override
   synchronized public boolean hasSkipped() {
      return skipMove;
   }

   synchronized public boolean hasMoved() {
      return moveSelected;
   }

   /** Tell if the next move is skipped, */
   synchronized public void skipNextMove() {
      skipMove = true;
      notifyAll();
   }

   /**
    * Handle a click over the board.
    *
    * @param curGameState the game state to update.
    * @param boardUI the board UI to update.
    * @param sel the selec poitn on the board.
    */
   public synchronized void handleBoardClick(GameState curGameState, CheckerBoard
boardUI, Point sel) {
      // The gameState is over or the current player isn't human
      if (curGameState.isGameOver()) return;

      // Determine if a move should be attempted
      //if (Board.isValidPoint(sel) && Board.isValidPoint(UI.getLastSelection())) {
      if (curGameState.isValidMove(boardUI.getLastSelection(), sel)) {
         boolean change = curGameState.isP1Turn();
         moveSelected = curGameState.move(boardUI.getLastSelection(), sel);
         if (moveSelected) notifyAll();
         change = (curGameState.isP1Turn() != change);
         boardUI.setLastSelection(change ? null : sel);
      } else {
         boardUI.setLastSelection(sel);
      }

      // Check if the selection is valid
      boardUI.setLastSelectionValid(curGameState.hasMove(boardUI.getLastSelection()));
```

```
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + "[isHuman=" + isHuman() + "]";
    }

}
```

## 5.7 MatchResult

```
package com.dca.checkers.model;

/**
 * The {@code MatchResult} enum represents all the possible match outcomes.
 */
public enum MatchResult {
    P1_WIN,
    P2_WIN,
    DRAW,
    UNKNOWN
}
```

## 5.8 Move

```


package com.dca.checkers.model;

import java.awt.*;
import java.util.Objects;

/**
 * The {@code Move} class represents a move and contains a weight associated
 * with the move.
 */
public class Move {

    /**
     * The startClick index of the move.
     */
    private byte startIndex;

    /** The end index of the move. */
    private byte endIndex;

    /** The move type */
    private MoveType type;


    public Move(int startIndex, int endIndex, MoveType type) {
        setStartIndex(startIndex);
        setEndIndex(endIndex);
        this.type = type;
    }

    public Move(Point start, Point end, MoveType type) {
        setStartIndex(Board.toIndex(start));
        setEndIndex(Board.toIndex(end));
        this.type = type;
    }

    public int getStartIndex() {
        return startIndex;
    }

    public void setStartIndex(int startIndex) {
        this.startIndex = (byte) startIndex;
```

```
    }

    public int getEndIndex() {
        return endIndex;
    }

    public void setEndIndex(int endIndex) {
        this.endIndex = (byte) endIndex;
    }

    public MoveType getType() { return type; }

    public void setType(MoveType type) { this.type = type; }

    public Point getStart() {
        return Board.toPoint(startIndex);
    }

    public void setStart(Point start) {
        setStartIndex(Board.toIndex(start));
    }

    public Point getEnd() {
        return Board.toPoint(endIndex);
    }

    public void setEnd(Point end) {
        setEndIndex(Board.toIndex(end));
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Move move = (Move) o;
        return startIndex == move.startIndex && endIndex == move.endIndex;
    }

    @Override
    public int hashCode() {
        return Objects.hash(startIndex, endIndex);
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + "[startIndex=" + startIndex + ", " +
"endIndex=" + endIndex;
    }
}
```

## 5.9 MoveType

```
package com.dca.checkers.model;

/**
 * The {@code MoveType} enum represents all the possible move types.
 */
public enum MoveType {
    SKIP,
    NORMAL
}
```

## 5.10 Player

```
/* Name: Player
 * Author: Devon McGrath
 * Description: This class represents a player of the system.
 */

package com.dca.checkers.model;

/**
 * The {@code Player} class is an interface class that represents a player in a
 * game of checkers.
 */
public interface Player {

   /**
    * Determines how the game is updated. If true, the user must interact with
    * the user interface to make a move. Otherwise, the game is updated via
    * {@link #updateGame(GameState)}.
    *
    * @return true if this player represents a user.
    */
   boolean isHuman();

   /**
    * Updates the gameState state to take a move for the current player. If there
    * is a move available that is multiple skips, it may be performed at once
    * by this method or one skip at a time.
    *
    * @param gameState the game state to update.
    */
   void updateGame(GameState gameState);

   /**
    * Tells if the player has skipped its turn.
    *
    * @return true if the player has skipped his turn, false otherwise.
    */
   boolean hasSkipped();

   /**
    * Tells if the player has moved
    *
    * @return true if the player has moved, false otherwise.
    */
   boolean hasMoved();

}
```

## 5.11 State

```
package com.dca.checkers.model;

/**
 * The {@code State} class interface for game state classes.
 */
public interface State extends Cloneable {

    /**
     * Returns the value of the state.
     * A value of 0 means the goal has been reached
     * @param evalForP1 tells if current state must be evaluated for player 1 (true) or
player 2 (false)
     * @return current state value.
     **/
    double value(boolean evalForP1);
}
```

## 5.12 CheckersBoard

```java
package com.dca.checkers.ui;

import com.dca.checkers.model.*;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.List;

/**
 * The {@code CheckerBoard} class is a graphical user interface component that
 * is capable of drawing any checkers gameState state.
 */
public class CheckerBoard extends JButton {

   private static final long serialVersionUID = -6014690893709316364L;

   /**
    * The number of pixels of padding between this component's border and the
    * actual checker board that is drawn.
    */
   private static final int PADDING = 16;

   /**
    * The gameState of checkers that is being played on this component.
    * The same instance is shared with @see gameState.
    */
   private GameState gameState;

   /**
    * The window containing this checker board UI component.
    */
   private CheckersWindow window;

   /**
    * The last point that the current human player selected on the checker board.
    */
   private Point selected;

   /**
    * The flag to determine if the selected tile is valid for the current human player
    */
   private boolean selectionValid;

   /**
    * The colour of the light tiles (by default, this is white).
    */
   private Color colorLightTile;

   /**
    * Color of reachable tiles from a movable piece
    */
   private Color colorNextTiles;

   /**
    * The colour of the tile id label.
    */
   private Color colorTileId;

   /**
    * The colour of the dark tiles (by default, this is black).
    */
   private Color colorDarkTile;
```

```
   /**
    * Color for a movable piece
    */
   private Color colorMovablePiece;

   /**
    * Tells if the tiles id must be shown
    */
   private boolean showTilesId;

   /**
    * Tells if movable pieces for the current player must be shown
    */
   private boolean showMovablePieces;

   /**
    * Tells if next tiles reachable from a movable pieces must be shown
    */
   private boolean showNextTiles;

   public CheckerBoard(CheckersWindow window, GameState gameState, boolean showTilesId,
boolean showMovablePieces, boolean showNextMoves) {

      // Setup the component
      super.setBorderPainted(false);
      super.setFocusPainted(false);
      super.setContentAreaFilled(false);
      super.setBackground(Color.LIGHT_GRAY);
      this.addActionListener(new ClickListener());

      // Setup the board settings
      this.colorLightTile = new Color(254, 234, 184);
      this.colorDarkTile = new Color(79, 124, 38);
      this.colorMovablePiece = new Color(233, 185, 52);
      this.colorTileId = colorLightTile;
      this.colorNextTiles = new Color(58, 188, 229);
      this.window = window;
      this.showTilesId = showTilesId;
      this.showMovablePieces = showMovablePieces;
      this.showNextTiles = showNextMoves;
      //Setup game
      this.gameState = (gameState == null) ? new GameState() : gameState;
   }

   /**
    * Draws the current checkers gameState state.
    */
   @Override
   public void paint(Graphics g) {
      super.paint(g);

      Graphics2D g2d = (Graphics2D) g;
      g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
      GameState gameState = this.gameState.copy();

      // Perform calculations
      final int BOX_PADDING = 8;
      final int W = getWidth(), H = getHeight();
      final int DIM = W < H ? W : H, BOX_SIZE = (DIM - 2 * PADDING) / 8;
      final int OFFSET_X = (W - BOX_SIZE * 8) / 2 + 5;
      final int OFFSET_Y = (H - BOX_SIZE * 8) / 2 + 5;
      final int CHECKER_SIZE = Math.max(0, BOX_SIZE - 2 * BOX_PADDING);

      // Draw checker board
      g.setColor(Color.BLACK);
      g.drawRect(OFFSET_X - 1, OFFSET_Y - 1, BOX_SIZE * 8 + 1, BOX_SIZE * 8 + 1);
```

```
      g.setColor(colorLightTile);
      g.fillRect(OFFSET_X, OFFSET_Y, BOX_SIZE * 8, BOX_SIZE * 8);
      g.setColor(colorDarkTile);

      //Get all moves for the select piece (if any available) and show them if required
      List<Move> selectedPieceMoves = gameState.getAllMoves(Board.toIndex(selected));

      for (int y = 0; y < 8; y++) {
         for (int x = (y + 1) % 2; x < 8; x += 2) {
            if (showMovablePieces && isMovablePiece(x, y))
g.setColor(colorMovablePiece);
            else if (showNextTiles && containsMoveEndsIn(selectedPieceMoves,
Board.toIndex(x, y)))
               g.setColor(colorNextTiles);
            else g.setColor(colorDarkTile);
            g.fillRect(OFFSET_X + x * BOX_SIZE, OFFSET_Y + y * BOX_SIZE, BOX_SIZE,
BOX_SIZE);
         }
      }

      // Highlight the selected tile if valid
      if (Board.isValidPoint(selected)) {
         g.setColor(selectionValid ? Color.GREEN : Color.RED);
         g.fillRect(OFFSET_X + selected.x * BOX_SIZE, OFFSET_Y + selected.y * BOX_SIZE,
BOX_SIZE, BOX_SIZE);
      }

      // Draw the checkers
      int balckCount = 0;
      Board b = gameState.getBoard();
      for (int y = 0; y < 8; y++) {
         int cy = OFFSET_Y + y * BOX_SIZE + BOX_PADDING;
         for (int x = (y + 1) % 2; x < 8; x += 2) {
            int id = b.get(x, y);
            int cx = OFFSET_X + x * BOX_SIZE + BOX_PADDING;

            //Set tile id
            if (showTilesId) {
               g.setColor(colorTileId);
               g.drawString(balckCount + "", cx - 7, cy + 2);
               balckCount++;
            }

            // Empty, just skip
            if (id == Board.EMPTY) {
               continue;
            }

            // Black checker
            if (id == Board.BLACK_CHECKER) {
               g.setColor(Color.DARK_GRAY);
               g.fillOval(cx + 1, cy + 2, CHECKER_SIZE, CHECKER_SIZE);
               g.setColor(Color.LIGHT_GRAY);
               g.drawOval(cx + 1, cy + 2, CHECKER_SIZE, CHECKER_SIZE);
               g.setColor(Color.BLACK);
               g.fillOval(cx, cy, CHECKER_SIZE, CHECKER_SIZE);
               g.setColor(Color.LIGHT_GRAY);
               g.drawOval(cx, cy, CHECKER_SIZE, CHECKER_SIZE);
            }

            // Black king
            else if (id == Board.BLACK_KING) {
               g.setColor(Color.DARK_GRAY);
               g.fillOval(cx + 1, cy + 2, CHECKER_SIZE, CHECKER_SIZE);
               g.setColor(Color.LIGHT_GRAY);
               g.drawOval(cx + 1, cy + 2, CHECKER_SIZE, CHECKER_SIZE);
               g.setColor(Color.DARK_GRAY);
               g.fillOval(cx, cy, CHECKER_SIZE, CHECKER_SIZE);
```

```
            g.setColor(Color.LIGHT_GRAY);
            g.drawOval(cx, cy, CHECKER_SIZE, CHECKER_SIZE);
            g.setColor(Color.BLACK);
            g.fillOval(cx - 1, cy - 2, CHECKER_SIZE, CHECKER_SIZE);
         }

         // White checker
         else if (id == Board.WHITE_CHECKER) {
            g.setColor(Color.LIGHT_GRAY);
            g.fillOval(cx + 1, cy + 2, CHECKER_SIZE, CHECKER_SIZE);
            g.setColor(Color.DARK_GRAY);
            g.drawOval(cx + 1, cy + 2, CHECKER_SIZE, CHECKER_SIZE);
            g.setColor(Color.WHITE);
            g.fillOval(cx, cy, CHECKER_SIZE, CHECKER_SIZE);
            g.setColor(Color.DARK_GRAY);
            g.drawOval(cx, cy, CHECKER_SIZE, CHECKER_SIZE);
         }

         // White king
         else if (id == Board.WHITE_KING) {
            g.setColor(Color.LIGHT_GRAY);
            g.fillOval(cx + 1, cy + 2, CHECKER_SIZE, CHECKER_SIZE);
            g.setColor(Color.DARK_GRAY);
            g.drawOval(cx + 1, cy + 2, CHECKER_SIZE, CHECKER_SIZE);
            g.setColor(Color.LIGHT_GRAY);
            g.fillOval(cx, cy, CHECKER_SIZE, CHECKER_SIZE);
            g.setColor(Color.DARK_GRAY);
            g.drawOval(cx, cy, CHECKER_SIZE, CHECKER_SIZE);
            g.setColor(Color.WHITE);
            g.fillOval(cx - 1, cy - 2, CHECKER_SIZE, CHECKER_SIZE);
         }

         // Any king (add some extra highlights)
         if (id == Board.BLACK_KING || id == Board.WHITE_KING) {
            g.setColor(new Color(255, 63, 43));
            g.drawOval(cx - 1, cy - 2, CHECKER_SIZE, CHECKER_SIZE);
            g.drawOval(cx + 1, cy, CHECKER_SIZE - 4, CHECKER_SIZE - 4);
            //g.drawString("K",cx+10, cy+15);
         }
      }
   }

   // Draw the player turn sign
   String msg = gameState.isP1Turn() ? "Player 1's turn" : "Player 2's turn";
   int width = g.getFontMetrics().stringWidth(msg);
   Color back = gameState.isP1Turn() ? Color.BLACK : Color.WHITE;
   Color front = gameState.isP1Turn() ? Color.WHITE : Color.BLACK;
   g.setColor(back);
   g.fillRect(W / 2 - width / 2 - 5, OFFSET_Y - 17, width + 10, 15);
   g.setColor(front);
   g.drawString(msg, W / 2 - width / 2, OFFSET_Y - 5);

   // Draw number of moves to draw
   msg = "Moves to draw: " + gameState.getNumMovesBeforeDraw();
   g.setColor(Color.BLACK);
   g.drawString(msg, W / 2 + 90, OFFSET_Y - 5);

   // Draw a gameState over sign
   if (gameState.isGameOver()) {
      MatchResult result = gameState.getResult();
      g.setFont(new Font("Arial", Font.BOLD, 20));
      switch (result) {
         case P1_WIN:
            msg = "Player 1 WIN!";
            break;
         case P2_WIN:
            msg = "Player 2 WIN!";
            break;
```

```
            case DRAW:
               msg = "DRAW!";
               break;
            default:
               msg = "UNKOWN RESULT";
         }


         width = g.getFontMetrics().stringWidth(msg);
         g.setColor(new Color(240, 240, 255));
         g.fillRoundRect(W / 2 - width / 2 - 5, OFFSET_Y + BOX_SIZE * 4 - 16, width +
10, 30, 10, 10);
         g.setColor(Color.RED);
         g.drawString(msg, W / 2 - width / 2, OFFSET_Y + BOX_SIZE * 4 + 7);
      }
   }

   /**
    * Check if at least one of the moves in selectedPieceMoves ends in endIndex.
    * @param selectedPieceMoves the moves to check.
    * @param endIndex end index to find.
    * @return true if at least one move ends in endIndex.
    */
   private boolean containsMoveEndsIn(List<Move> selectedPieceMoves, int endIndex) {
      for (Move m : selectedPieceMoves) {
         if (m.getEndIndex() == endIndex) return true;
      }
      return false;
   }

   /**
    * Tell if a piece, in position (x,y) on the board, can be moved.
    * @param x the x position of the piece.
    * @param y the y position of the piece.
    * @return true if at least one move is currently available for piece in position
(x,y); false otherwise.
    */
   private boolean isMovablePiece(int x, int y) {
      return gameState.hasMove(new Point(x, y));
   }

   /**
    * Cancel last selection (if any).
    */
   public void cancelLastSelection() {
      selected = null;
   }

   /**
    * Cancel last selection (if any).
    * @return the las selected point on the board.
    */
   public Point getLastSelection() {
      return selected;
   }

   /**
    * Cancel last selection (if any).
    * @param p the new selected tile.
    */
   public void setLastSelection(Point p) {
      selected = p;
   }

   /**
    * Set tiles id visibility.
    * @param isVisible the new value to set for tiles id visibility flag.
    */
```

```
    public void setTileIdVisibility(boolean isVisible) {
        showTilesId = isVisible;
        repaint();
    }

    /**
     * Set last selection as valid (if any).
     * @param selectionValid the flag that tells if the last selection is valid (true) or
not (false).
     */
    public void setLastSelectionValid(boolean selectionValid) {
        this.selectionValid = selectionValid;
    }

    /**
     * Show (show == true) or hide (show == false) pieces that can be moved.
     * @param show the new value for the flag,
     */
    public void setShowMovablePieces(boolean show) {
        showMovablePieces = show;
        repaint();
    }

    /**
     * Show (show == true) or hide (show == false) next moves of the selected piece.
     * @param show the new value for the flag,
     */
    public void setShowNextMoves(boolean show) {
        showNextTiles = show;
        repaint();
    }

    /**
     * The {@code ClickListener} class is responsible for responding to click
     * events on the checker board component. It uses the coordinates of the
     * mouse relative to the location of the checker board component.
     */
    private class ClickListener implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            // Get the new mouse coordinates and handle the click
            Point p = CheckerBoard.this.getMousePosition();
            if (p != null) {
                int x = p.x;
                int y = p.y;
                // Determine what square (if any) was selected
                final int W = getWidth(), H = getHeight();
                final int DIM = W < H ? W : H, BOX_SIZE = (DIM - 2 * PADDING) / 8;
                final int OFFSET_X = (W - BOX_SIZE * 8) / 2;
                final int OFFSET_Y = (H - BOX_SIZE * 8) / 2;
                x = (x - OFFSET_X) / BOX_SIZE;
                y = (y - OFFSET_Y) / BOX_SIZE;
                Point sel = new Point(x, y);
                window.clickOnBoard(sel);
            }
        }

    }

}
```

## 5.13 CheckersWindow

```java
package com.dca.checkers.ui;

import com.dca.checkers.model.GameManager;
import com.dca.checkers.model.GameState;
import com.dca.checkers.model.Player;

import javax.swing.*;
import java.awt.*;

/**
 * The {@code CheckersWindow} class is responsible for managing a window. This
 * window contains a game of checkers and also options to change the settings
 * of the game with an {@link OptionPanel}.
 */
public class CheckersWindow extends JFrame {

   private static final long serialVersionUID = 8782122389400590079L;

   /** The default width for the checkers window. */
   public static final int DEFAULT_WIDTH = 520;

   /** The default height for the checkers window. */
   public static final int DEFAULT_HEIGHT = 825;

   /** The default title for the checkers window. */
   public static final String DEFAULT_TITLE = "Checkers";

   /** The checker board component playing the updatable game. */
   private CheckerBoard board;

   /**
    * Reference to the game manager
    */
   private GameManager gameManager;

   /**
    * Reference to the option panel
    */
   private OptionPanel opts;

   public CheckersWindow() {
      this(DEFAULT_WIDTH, DEFAULT_HEIGHT, DEFAULT_TITLE);
   }

   public CheckersWindow(int width, int height, String title) {

      // Setup the window
      super(title);
      super.setSize(width, height);
      super.setLocationByPlatform(true);

      // Setup the components
      GameState startState = new GameState();
      JPanel layout = new JPanel(new BorderLayout());
      this.opts = new OptionPanel(this);
      this.board = new CheckerBoard(this, startState, opts.getTilesIdVisibility(),
opts.getShowMovablePieces(), opts.getShowNextMoves());
      layout.add(board, BorderLayout.CENTER);
      layout.add(opts, BorderLayout.SOUTH);
      layout.setBackground(new Color(231, 187, 134));
      this.add(layout);
      gameManager = new GameManager(startState, board, opts);
      gameManager.start();
   }
```

```java
/**
 * Updates the type of player that is being used for player 1.
 *
 * @param player1  the new player instance to control player 1.
 */
public void setPlayer1(Player player1) {
   System.out.println("Requested set of player 1.");
   gameManager.setPlayer1(player1);
}

/**
 * Updates the type of player that is being used for player 2.
 *
 * @param player2  the new player instance to control player 2.
 */
public void setPlayer2(Player player2) {
   System.out.println("Requested set of player 2.");
   gameManager.setPlayer2(player2);
}

/**
 * Handle a click over the game board.
 *
 * @param sel the select point on the game board.
 */
public void clickOnBoard(Point sel) {
   System.out.println("Requested click request.");
   gameManager.handleBoardClick(sel);
}

/**
 * Set tiles id visibility.
 * @param isVisible the new value to set for tiles id visibility flag.
 */
public void setTileIdVisibility(boolean isVisible) {
   board.setTileIdVisibility(isVisible);
}

/**
 * Resets the game of checkers in the window.
 */
public void resetClick() {
   gameManager.resetClick();
}

/**
 * Start the game
 */
public void startClick() {
   gameManager.startClick();
}

/**
 * Resume the paused game
 */
public void resumeClick() {
   gameManager.resumeClick();
}

/**
 * Pause the current game
 */
public void pauseClick() {
   gameManager.pauseClick();
}

/**
 * Show (show == true) or hide (show == false) pieces that can be moved.
```

```
     * @param show the new value for the flag,
     */
    public void setShowMovablePieces(boolean show) {
        board.setShowMovablePieces(show);
    }

    /**
     * Show (show == true) or hide (show == false) next moves of the selected piece.
     * @param show the new value for the flag,
     */
    public void setShowNextMoves(boolean show) {
        board.setShowNextMoves(show);
    }

    /**
     * Undo the last move
     */
    public void undoMove() {
        gameManager.undo();
    }

    /**
     * Redo the last move
     */
    public void redoMove() {
        gameManager.redo();
    }

    /**
     * Set delay for a AI move.
     * @param value the new delay value.
     */
    public void setDelay(int value) {
        gameManager.setDelay(value);
    }

}
```

## 5.14 OptionPanel

```
/* Name: OptionPanel
 * Author: Devon McGrath
 * Description: This class is a user interface to interact with a checkers
 * game window.
 */

package com.dca.checkers.ui;

import com.dca.checkers.ai.AIMinMax;
import com.dca.checkers.ai.AIRandomPlayer;
import com.dca.checkers.model.HumanPlayer;
import com.dca.checkers.model.Player;

import javax.swing.*;
import java.awt.*;

/**
 * The {@code OptionPanel} class provides a user interface component to control
 * options for the game of checkers being played in the window {@link CheckersWindow}.
 */
public class OptionPanel extends JPanel {

    private static final long serialVersionUID = -4763875452164030755L;

    /**
     * The button that when clicked, starts the game.
     */
    public JButton btnStart;
```

```java
    /**
     * The button that when clicked, reset the game.
     */
    public JButton btnRest;
    /**
     * The button that when clicked, restarts the game if it was previously paused.
     */
    public JButton btnResume;
    /**
     * The button that when clicked, pauses the game.
     */
    public JButton btnPause;
    /**
     * The button that when clicked, undo the last move.
     */
    public JButton btnUndo;
    /**
     * The button that when clicked, redo the last move.
     */
    public JButton btnRedo;
    /**
     * The combo box that changes what type of player player 1 is.
     */
    public JComboBox<String> cmbPlayer1Type;
    /**
     * The combo box that changes what type of player player 2 is.
     */
    public JComboBox<String> cmbPlayer2Type;
    /**
     * Flag for tiles ids visibility
     */
    public JCheckBox chbTilesId;
    /**
     * Flag to show piece that the current player can move
     */
    public JCheckBox chbShowMovablePieces;
    /**
     * Flag to show next moves of selected piece
     */
    public JCheckBox chbShowNextMoves;
    /**
     * Console text area used to send messages to user
     */
    public JTextArea txtAreaConsole;
    /**
     * Slider to set delay of AI moves
     */
    public JSlider sliderDelay;
    /**
     * Used to show delay value
     */
    public JLabel labelDelayValue;
    /**
     * The checkers window to update when an option is changed.
     */
    private CheckersWindow window;

    /**
     * Creates a new option panel for the specified checkers window.
     *
     * @param window the window with the game of checkers to update.
     */
    public OptionPanel(CheckersWindow window) {
        super(new GridLayout(0, 1));

        this.window = window;

        // Initialize the components
```

```
      final String[] playerTypeOpts = {"Human", "AI - Random", "AI - MinMax"};
      this.sliderDelay = new JSlider(JSlider.HORIZONTAL, 0, 2000, 1000);
      this.labelDelayValue = new JLabel(sliderDelay.getValue() + "");
      this.btnStart = new JButton("Start");
      this.btnResume = new JButton("Resume");
      this.btnPause = new JButton("Pause");
      this.btnRest = new JButton("Reset");
      this.btnUndo = new JButton("Undo");
      this.btnRedo = new JButton("Redo");
      this.cmbPlayer1Type = new JComboBox<>(playerTypeOpts);
      this.cmbPlayer2Type = new JComboBox<>(playerTypeOpts);
      this.chbTilesId = new JCheckBox("Show tiles IDs", true);
      this.chbShowMovablePieces = new JCheckBox("Show movable pieces", true);
      this.chbShowNextMoves = new JCheckBox("Show next moves", true);
      this.txtAreaConsole = new JTextArea();
      this.txtAreaConsole.setEditable(false);
      this.txtAreaConsole.setRows(3);
      this.btnStart.addActionListener(e -> window.startClick());
      this.btnResume.addActionListener(e -> window.resumeClick());
      this.btnPause.addActionListener(e -> window.pauseClick());
      this.btnRest.addActionListener(e -> window.resetClick());
      this.btnUndo.addActionListener(e -> window.undoMove());
      this.btnRedo.addActionListener(e -> window.redoMove());
      this.sliderDelay.addChangeListener(e -> {
         labelDelayValue.setText(sliderDelay.getValue() + "");
         window.setDelay(sliderDelay.getValue());
      });
      this.cmbPlayer1Type.addActionListener(e ->
window.setPlayer1(getPlayer(cmbPlayer1Type)));
      this.cmbPlayer2Type.addActionListener(e ->
window.setPlayer2(getPlayer(cmbPlayer2Type)));
      this.chbTilesId.addActionListener(e ->
window.setTileIdVisibility(chbTilesId.isSelected()));
      this.chbShowMovablePieces.addActionListener(e ->
window.setShowMovablePieces(chbShowMovablePieces.isSelected()));
      this.chbShowNextMoves.addActionListener(e ->
window.setShowNextMoves(chbShowNextMoves.isSelected()));
      JScrollPane pan0 = new JScrollPane(txtAreaConsole);
      new SmartScroller(pan0);
      JPanel pan1 = new JPanel(new FlowLayout(FlowLayout.LEFT));
      JPanel pan2 = new JPanel(new FlowLayout(FlowLayout.LEFT));
      JPanel pan3 = new JPanel(new FlowLayout(FlowLayout.LEFT));
      JPanel pan4 = new JPanel(new FlowLayout(FlowLayout.LEFT));
      JPanel pan5 = new JPanel(new FlowLayout(FlowLayout.LEFT));
      JPanel pan6 = new JPanel(new FlowLayout(FlowLayout.LEFT));
      JPanel pan7 = new JPanel(new FlowLayout(FlowLayout.LEFT));

      pan0.setBackground(new Color(214, 34, 28));
      pan1.setBackground(new Color(231, 187, 134));
      pan2.setBackground(new Color(231, 187, 134));
      pan3.setBackground(new Color(231, 187, 134));
      pan4.setBackground(new Color(231, 187, 134));
      pan5.setBackground(new Color(231, 187, 134));
      pan6.setBackground(new Color(231, 187, 134));
      pan7.setBackground(new Color(231, 187, 134));


      // Add components to the layout
      JLabel txtDelay = new JLabel("AI Delay (ms): ");
      pan1.add(txtDelay);
      pan1.add(labelDelayValue);
      pan1.add(sliderDelay);

      JLabel txtP1 = new JLabel("Player 1: ");
      txtP1.setOpaque(true);
      txtP1.setBackground(Color.BLACK);
      txtP1.setForeground(Color.WHITE);
      pan2.add(txtP1);
```

```
      pan2.add(cmbPlayer1Type);
      JLabel txtP2 = new JLabel("Player 2: ");
      txtP2.setOpaque(true);
      txtP2.setBackground(Color.WHITE);
      txtP2.setForeground(Color.BLACK);
      pan3.add(txtP2);
      pan3.add(cmbPlayer2Type);
      pan4.add(btnStart);
      pan4.add(btnResume);
      pan4.add(btnPause);
      pan4.add(btnRest);
      pan4.add(btnUndo);
      pan4.add(btnRedo);
      pan5.add(chbTilesId);
      pan6.add(chbShowMovablePieces);
      pan7.add(chbShowNextMoves);
      this.add(pan0);
      this.add(pan1);
      this.add(pan2);
      this.add(pan3);
      this.add(pan4);
      this.add(pan5);
      this.add(pan6);
      this.add(pan7);
   }

   /**
    * Get the type of player select for player 1.
    * @return the player 1 object.
    */
   public Player getPlayer1() {
      return getPlayer(cmbPlayer1Type);
   }

   /**
    * Get the type of player select for player 2.
    * @return the player 2 object.
    */
   public Player getPlayer2() {
      return getPlayer(cmbPlayer2Type);
   }

   /**
    * Gets a new instance of the type of player selected for the specified
    * combo box.
    *
    * @param playerOpts the combo box with the player options.
    * @return a new instance of a {@link com.dca.checkers.model.Player} object that
corresponds
    * with the type of player selected.
    */
   private Player getPlayer(JComboBox<String> playerOpts) {

      Player player = new HumanPlayer();
      if (playerOpts == null) {
         return player;
      }

      // Determine the type
      String type = "" + playerOpts.getSelectedItem();
      if (type.equals("AI - Random")) {
         player = new AIRandomPlayer();
      }
      if (type.equals("AI - MinMax")) {
         player = new AIMinMax();
      }

      return player;
```

```
    }

    /**
     * Get the flag that tells tiles id must be shown or not.
     * @return true if the flag that tells tiles id must be shown or not is checked,
otherwise return false.
     */
    public boolean getTilesIdVisibility() {
        return chbTilesId.isSelected();
    }

    /**
     * Get the flag that tells if movable pieces must be shown.
     * @return true if the flag that tells if movable pieces must be shown is checked,
otherwise return false.
     */
    public boolean getShowMovablePieces() {
        return chbShowMovablePieces.isSelected();
    }

    /**
     * Get the flag that tells if moves of movable pieces must be shown.
     * @return true if the flag that tells if moves of movable pieces must be shown is
checked, otherwise return false.
     */
    public boolean getShowNextMoves() {
        return chbShowNextMoves.isSelected();
    }

}
```

## 5.15 SmartController

```
package com.dca.checkers.ui;

import java.awt.Component;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;

/**
 * The {@code SmartScroller} will attempt to keep the viewport positioned based on
 * the users interaction with the scrollbar. The normal behaviour is to keep
 * the viewport positioned to see new data as it is dynamically added.
 * <p>
 * Assuming vertical scrolling and data is added to the bottom:
 * <p>
 * - when the viewport is at the bottom and new data is added,
 * then automatically scroll the viewport to the bottom
 * - when the viewport is not at the bottom and new data is added,
 * then do nothing with the viewport
 * <p>
 * Assuming vertical scrolling and data is added to the top:
 * <p>
 * - when the viewport is at the top and new data is added,
 * then do nothing with the viewport
 * - when the viewport is not at the top and new data is added, then adjust
 * the viewport to the relative position it was at before the data was added
 * <p>
 * Similar logic would apply for horizontal scrolling.
 */
public class SmartScroller implements AdjustmentListener {

    private final static int HORIZONTAL = 0;
    private final static int VERTICAL = 1;

    private final static int START = 0;
    private final static int END = 1;
```

```
   private int viewportPosition;

   private JScrollBar scrollBar;
   private boolean adjustScrollBar = true;

   private int previousValue = -1;
   private int previousMaximum = -1;

   /**
    * Convenience constructor.
    * Scroll direction is VERTICAL and viewport position is at the END.
    *
    * @param scrollPane the scroll pane to monitor
    */
   public SmartScroller(JScrollPane scrollPane) {
      this(scrollPane, VERTICAL, END);
   }

   /**
    * Convenience constructor.
    * Scroll direction is VERTICAL.
    *
    * @param scrollPane       the scroll pane to monitor
    * @param viewportPosition valid values are START and END
    */
   public SmartScroller(JScrollPane scrollPane, int viewportPosition) {
      this(scrollPane, VERTICAL, viewportPosition);
   }

   /**
    * Specify how the SmartScroller will function.
    *
    * @param scrollPane       the scroll pane to monitor
    * @param scrollDirection  indicates which JScrollBar to monitor.
    *                         Valid values are HORIZONTAL and VERTICAL.
    * @param viewportPosition indicates where the viewport will normally be
    *                         positioned as data is added.
    *                         Valid values are START and END
    */
   public SmartScroller(JScrollPane scrollPane, int scrollDirection, int
viewportPosition) {
      if (scrollDirection != HORIZONTAL && scrollDirection != VERTICAL)
         throw new IllegalArgumentException("invalid scroll direction specified");

      if (viewportPosition != START && viewportPosition != END)
         throw new IllegalArgumentException("invalid viewport position specified");

      this.viewportPosition = viewportPosition;

      if (scrollDirection == HORIZONTAL) scrollBar =
scrollPane.getHorizontalScrollBar();
      else scrollBar = scrollPane.getVerticalScrollBar();

      scrollBar.addAdjustmentListener(this);

      //  Turn off automatic scrolling for text components

      Component view = scrollPane.getViewport().getView();

      if (view instanceof JTextComponent) {
         JTextComponent textComponent = (JTextComponent) view;
         DefaultCaret caret = (DefaultCaret) textComponent.getCaret();
         caret.setUpdatePolicy(DefaultCaret.NEVER_UPDATE);
      }
   }

   @Override
```

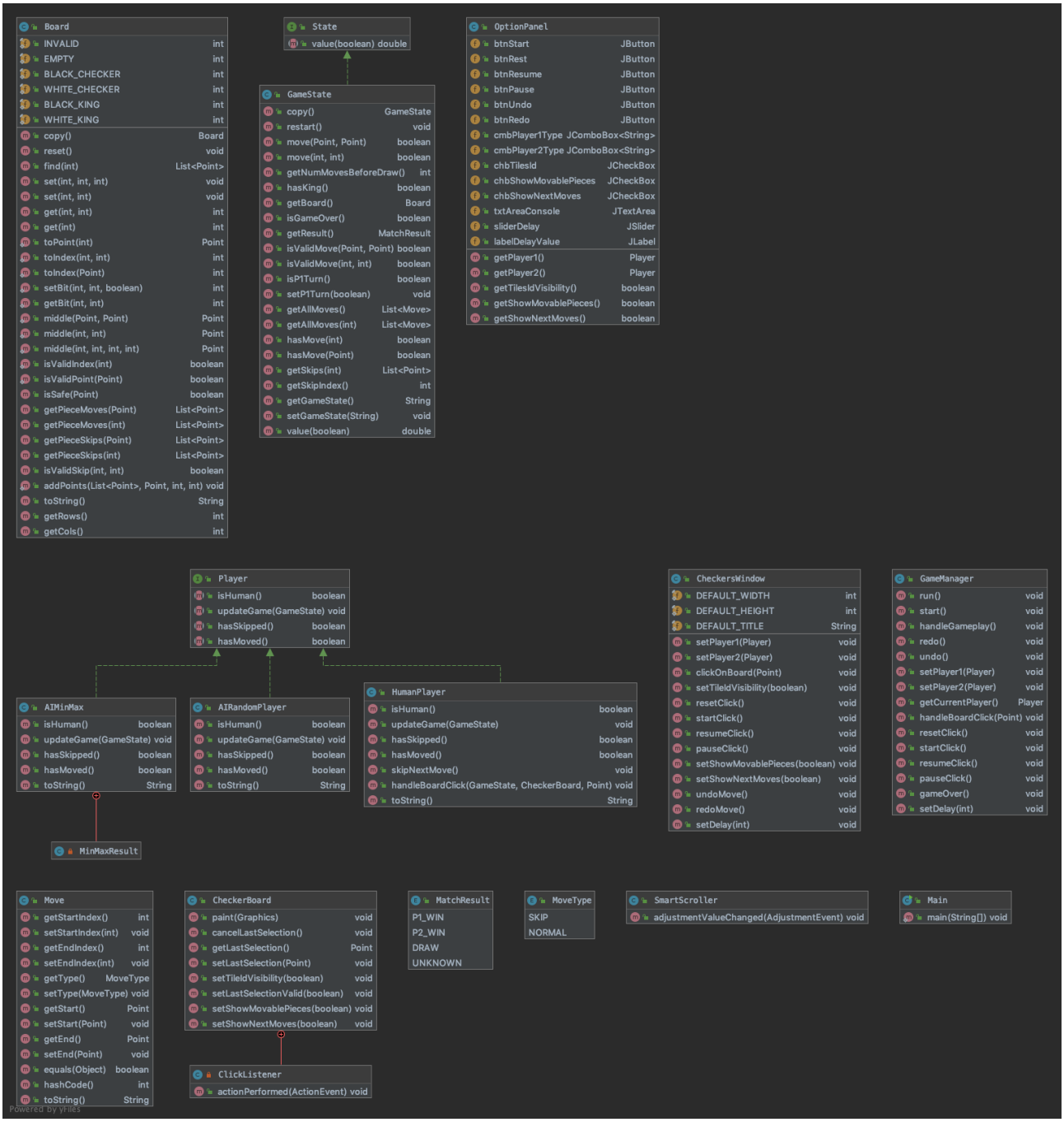
```
   public void adjustmentValueChanged(final AdjustmentEvent e) {
      SwingUtilities.invokeLater(() -> checkScrollBar(e));
   }

   /*
    *  Analyze every adjustment event to determine when the viewport
    *  needs to be repositioned.
    */
   private void checkScrollBar(AdjustmentEvent e) {
      //  The scroll bar listModel contains information needed to determine
      //  whether the viewport should be repositioned or not.

      JScrollBar scrollBar = (JScrollBar) e.getSource();
      BoundedRangeModel listModel = scrollBar.getModel();
      int value = listModel.getValue();
      int extent = listModel.getExtent();
      int maximum = listModel.getMaximum();

      boolean valueChanged = previousValue != value;
      boolean maximumChanged = previousMaximum != maximum;

      //  Check if the user has manually repositioned the scrollbar

      if (valueChanged && !maximumChanged) {
         if (viewportPosition == START) adjustScrollBar = value != 0;
         else adjustScrollBar = value + extent >= maximum;
      }

      //  Reset the "value" so we can reposition the viewport and
      //  distinguish between a user scroll and a program scroll.
      //  (ie. valueChanged will be false on a program scroll)

      if (adjustScrollBar && viewportPosition == END) {
         //  Scroll the viewport to the end.
         scrollBar.removeAdjustmentListener(this);
         value = maximum - extent;
         scrollBar.setValue(value);
         scrollBar.addAdjustmentListener(this);
      }

      if (adjustScrollBar && viewportPosition == START) {
         //  Keep the viewport at the same relative viewportPosition
         scrollBar.removeAdjustmentListener(this);
         value = value + maximum - previousMaximum;
         scrollBar.setValue(value);
         scrollBar.addAdjustmentListener(this);
      }

      previousValue = value;
      previousMaximum = maximum;
   }

}
```

## 5.16 Main

```
package com.dca.checkers;

import com.dca.checkers.ui.CheckersWindow;

import javax.swing.*;

public class Main {

   public static void main(String[] args) {

      //Set the look and feel to the OS look and feel
```

```
      try {
         UIManager.setLookAndFeel(
               UIManager.getSystemLookAndFeelClassName());
      } catch (Exception e) {
         e.printStackTrace();
      }

      // Create a window to display the checkers game
      CheckersWindow window = new CheckersWindow();
      window.setDefaultCloseOperation(CheckersWindow.EXIT_ON_CLOSE);
      window.setVisible(true);
   }
}
```

# Appendice 2: UML, diagramma delle classi



## Appendice 3: Javadoc del progetto

Nella cartella “Javadoc” consegnata insieme alla presente relazione, aprire ilf file “index.html” con un browser qualsiasi per navigare la documentazione del progetto generata utilizzando il tool “Javadoc”.

## Appendice 4: Analisi parametro Ply

| Id partita | Tempo in ns |
|---|---|
| 1 | 177184356 |
| 1 | 10060214 |
| 1 | 135651479 |
| 1 | 89903501 |
| 1 | 348791137 |
| 1 | 2135844 |
| 1 | 15964888 |
| 1 | 214870543 |
| 1 | 201421904 |
| 1 | 17810695 |
| 1 | 13855698 |
| 1 | 156907902 |
| 1 | 423361 |
| 1 | 2958846 |
| 1 | 4632846 |
| 1 | 84947898 |
| 1 | 5135309 |
| 1 | 13787899 |
| 1 | 1164212 |
| 1 | 5816490 |
| 1 | 2283290 |
| 1 | 6102239 |
| 1 | 35775 |
| 2 | 53578113 |
| 2 | 3187085 |
| 2 | 47139105 |
| 2 | 25629898 |
| 2 | 1842390 |
| 2 | 74909890 |
| 2 | 10252656 |
| 2 | 11560050 |
| 2 | 39828851 |
| 2 | 33267480 |
| 2 | 5297715 |
| 2 | 8781810 |
| 2 | 38354010 |
| 2 | 12764205 |
| 2 | 6148843 |
| 2 | 4271581 |
| 2 | 19406285 |
| 2 | 26793411 |

| | |
|---|---|
| 2 | 29827480 |
| 2 | 573888 |
| 2 | 3686908 |
| 2 | 22625875 |
| 2 | 27044515 |
| 2 | 13930452 |

| | |
|---|---|
| 2 | 744154 |
| 2 | 12682954 |
| 2 | 13642660 |
| 2 | 40774 |
| 3 | 59556408 |
| 3 | 39828986 |
| 3 | 682656 |
| 3 | 6629918 |
| 3 | 53227479 |
| 3 | 4605650 |
| 3 | 5992366 |
| 3 | 31571479 |
| 3 | 1655752 |
| 3 | 5084841 |
| 3 | 1411874 |
| 3 | 9793687 |
| 3 | 58006427 |
| 3 | 11115309 |
| 3 | 145405989 |
| 3 | 6360629 |
| 3 | 53926953 |
| 3 | 38666752 |
| 3 | 47109461 |
| 3 | 36612448 |
| 3 | 162158 |
| 3 | 1224930 |
| 3 | 10772325 |
| 3 | 4455529 |
| 3 | 17910 |
| 4 | 54639182 |
| 4 | 41345148 |
| 4 | 764128 |
| 4 | 6574132 |
| 4 | 59348882 |
| 4 | 17661311 |
| 4 | 73981990 |
| 4 | 3582026 |

| | |
|---|---|
| 4 | 158393950 |
| 4 | 101429727 |
| 4 | 87564543 |
| 4 | 1741468 |
| 4 | 16283629 |
| 4 | 38181934 |
| 4 | 208527424 |
| 4 | 2161726 |
| 4 | 63539620 |
| 4 | 23595798 |
| 4 | 5432602 |

| | |
|---|---|
| 4 | 21328846 |
| 4 | 439449 |
| 4 | 4425273 |
| 4 | 20713565 |
| 4 | 30349577 |
| 4 | 778124 |
| 4 | 5168245 |
| 4 | 2828893 |
| 4 | 20683 |
| 5 | 54623610 |
| 5 | 27625851 |
| 5 | 301941 |
| 5 | 13645374 |
| 5 | 72759211 |
| 5 | 61331394 |
| 5 | 33913854 |
| 5 | 1219980 |
| 5 | 27100338 |
| 5 | 6966797 |
| 5 | 38285852 |
| 5 | 9775729 |
| 5 | 51203606 |
| 5 | 28214414 |
| 5 | 162051 |
| 5 | 3667381 |
| 5 | 28893471 |
| 5 | 3015127 |
| 5 | 12607923 |
| 5 | 2536623 |
| 5 | 10668348 |
| 5 | 4292746 |
| 5 | 4919217 |

| | |
|---|---|
| 5 | 69103 |
| 5 | 13447 |
| 5 | 8198 |

| Id partita | Tempo in ns |
|---|---|
| 1 | 1960503608 |
| 1 | 144612593 |
| 1 | 1594445054 |
| 1 | 1003180603 |
| 1 | 2002443788 |
| 1 | 70342618 |
| 1 | 2719652634 |
| 1 | 1262699593 |
| 1 | 2928744619 |
| 1 | 2657705418 |
| 1 | 103239497 |
| 1 | 3702968351 |
| 1 | 39988648 |
| 1 | 1036194221 |
| 1 | 441434223 |
| 1 | 637832890 |
| 1 | 332940486 |
| 1 | 115665629 |
| 1 | 137442486 |
| 1 | 43502993 |
| 1 | 72386254 |
| 1 | 286530881 |
| 1 | 80038832 |
| 1 | 57836136 |
| 1 | 18836068 |
| 1 | 20431811 |
| 1 | 217196784 |
| 1 | 367718879 |
| 1 | 2561704 |
| 1 | 17460552 |
| 1 | 186217159 |
| 1 | 67760268 |
| 1 | 71830969 |
| 1 | 104304751 |
| 1 | 618108748 |
| 1 | 279745173 |
| 1 | 311115254 |
| 1 | 82664403 |
| 1 | 544135813 |

| | |
|---|---|
| 1 | 305146511 |
| 1 | 893106335 |
| 1 | 463501287 |
| 1 | 154698755 |
| 1 | 680744335 |
| 1 | 1006790438 |
| 1 | 412591956 |
| 1 | 242787319 |

| | |
|---|---|
| 1 | 12922 |
| 2 | 1091015688 |
| 2 | 453258653 |
| 2 | 471714385 |
| 2 | 208242946 |
| 2 | 1309634 |
| 2 | 7207022 |
| 2 | 30585255 |
| 2 | 548249258 |
| 2 | 83563843 |
| 2 | 36818771 |
| 2 | 828193405 |
| 2 | 130467435 |
| 2 | 58615160 |
| 2 | 763868642 |
| 2 | 13906791 |
| 2 | 61406092 |
| 2 | 72803 |
| 2 | 22510 |
| 2 | 20095 |
| 3 | 1088957691 |
| 3 | 85442875 |
| 3 | 1145861810 |
| 3 | 988748975 |
| 3 | 1312057465 |
| 3 | 1043339130 |
| 3 | 290408138 |
| 3 | 973676186 |
| 3 | 339680269 |
| 3 | 51132253 |
| 3 | 446860565 |
| 3 | 128464834 |
| 3 | 2091158133 |
| 3 | 1880187101 |
| 3 | 1578038607 |

| | |
|---|---|
| 3 | 843412546 |
| 3 | 864440230 |
| 3 | 231011570 |
| 3 | 387559131 |
| 3 | 7969895 |
| 3 | 1269515 |
| 3 | 3210226 |
| 3 | 9064415 |
| 3 | 23101 |
| 4 | 1128856259 |
| 4 | 455523832 |
| 4 | 430500845 |
| 4 | 228102647 |

| | |
|---|---|
| 4 | 303136146 |
| 4 | 1823343081 |
| 4 | 126915607 |
| 4 | 273289953 |
| 4 | 305051247 |
| 4 | 40597278 |
| 4 | 820502003 |
| 4 | 146895655 |
| 4 | 651862843 |
| 4 | 182656017 |
| 4 | 854165489 |
| 4 | 70022206 |
| 4 | 200282916 |
| 4 | 1150343558 |
| 4 | 157204249 |
| 4 | 907810900 |
| 4 | 16365799 |
| 4 | 152242389 |
| 4 | 35780 |
| 4 | 12305 |
| 5 | 1078402689 |
| 5 | 449582013 |
| 5 | 470932852 |
| 5 | 113523929 |
| 5 | 18677303 |
| 5 | 13313889 |
| 5 | 53404402 |
| 5 | 228154528 |
| 5 | 98290901 |
| 5 | 12773324 |

| | |
|---|---|
| 5 | 83526764 |
| 5 | 174876752 |
| 5 | 874647545 |
| 5 | 267657448 |
| 5 | 134181552 |
| 5 | 45156985 |
| 5 | 51031279 |
| 5 | 6272279 |
| 5 | 27313726 |
| 5 | 21754716 |
| 5 | 19805 |

| Id partita | Tempo in ns |
|---|---|
| 1 | 24229807886 |
| 1 | 14428107967 |
| 1 | 7242276392 |
| 1 | 14702114972 |
| 1 | 3879767846 |
| 1 | 7440269611 |
| 1 | 21504991299 |
| 1 | 6081946183 |
| 1 | 1059773661 |
| 1 | 9415271706 |
| 1 | 61468672225 |
| 1 | 2895135532 |
| 1 | 9114686268 |
| 1 | 71363275320 |
| 1 | 1,04817E+11 |
| 1 | 3089205262 |
| 1 | 3239819133 |
| 1 | 88663734267 |
| 1 | 33683016000 |
| 1 | 12803808879 |
| 1 | 181438911 |
| 1 | 3057972786 |
| 1 | 2025938374 |
| 1 | 26936 |
| 1 | 10996 |

| Prova | Media per 1 mossa (ns) | Media per 1 mossa (s) |
|---|---|---|
| d=5 | 32632005,7 | 0,032632006 |
| d=7 | 483320051,3 | 0,483320051 |
| d=9 | 20255524674 | 20,25552467 |

# 6 Bibliografia

[1] «Regolamento FID (Federazione Italiana Dama),» [Online]. Available: http://www.fid.it/regolamenti/capo1.htm.

[2] «Teoria dei Giochi,» [Online]. Available: https://it.wikipedia.org/wiki/Teoria_dei_giochi#Giochi_a_somma_zero.

[3] «Board Representation,» [Online]. Available: https://www.chessprogramming.org/Board_Representation.

[4] «Reviewing the game of Checkers,» [Online]. Available: http://webdocs.cs.ualberta.ca/~duane/publications/pdf/1991hpai.pdf.

[5] «Archivo delle partite di dama italiana,» [Online]. Available: http://www.federdama.it/cms/servizi/download/database-di-partite.

[6] «Portable Draughts Notation (PDN),» [Online]. Available: https://en.wikipedia.org/wiki/Portable_Draughts_Notation.

[7] «Some Studies in Machine Learning Using the Game of Checkers,» [Online]. Available: http://www2.stat.duke.edu/~sayan/R_stuff/Datamatters.key/Data/samuel_1959_B-95.pdf.

[8] «Search: Games, Minimax, and Alpha-Beta,» [Online]. Available: https://www.youtube.com/watch?v=STjW3eH0Cik&ab_channel=MITOpenCourseWare.

[9] «British Museum Algorithm,» [Online]. Available: https://en.wikipedia.org/wiki/British_Museum_algorithm.