Francesco Pilotti-268500
Julian Sparber-260324

Progetto di ingegneria del software
MASTER CHESS

Docente: Prof. Edoardo Bontà

Specifica del problema

Specifica del problema:

È richiesta la creazione di un programma che permetta a due utenti di giocare a scacchi offline e da un solo dispositivo.

Il programma deve simulare una scacchiera con 64 caselle distribuite su 8 righe ed 8 colonne a colori alternati (bianco e nero) con pedine di due colori :il primo tipo di colore bianco (assegnato al primo giocatore) ed il secondo tipo di colore nero,assegnato al secondo giocatore.

Le pedine dovranno essere di sei tipi con altrettanti tipi di movimenti :

- ***Re***: spostamento di 1 casella in qualsiasi direzione purché la casella d'arrivo non sia nella traiettoria di movimento di un pezzo avversario
- ***Regina***: spostamento di un qualsiasi numero di caselle diagonalmente,orizzontalmente o verticalmente
- ***Torre***: spostamento di un qualsiasi numero di caselle orizzontalmente o verticalmente
- ***Alfiere***: spostamento di un qualsiasi numero di caselle diagonalmente
- ***Cavallo***: spostamento “ad elle”,ossia di due caselle verticalmente ed una a sinistra o a destra rispetto la posizione di arrivo verticale oppure di due caselle orizzontalmente ed una in basso o in alto rispetto la posizione d'arrivo orizzontale
- ***Pedone***: spostamento di una casella(con possibilità di spostamento di due caselle se se il suddetto non è mai stato mosso) verso la parte opposta della scacchiera

Ogni pedina può catturarne un'altra semplicemente muovendocisi sopra,eccezion fatta per il pedone che può mangiare una pedina se e solo se quest'ultima si trova in una delle due caselle diagonali adiacenti al pedone poste lungo la sua traiettoria verso la parte opposta della scacchiera.

In particolare si richiede l'implementazione di cinque regole degli scacchi:

- ***Promozione del pedone***: un pedone che riesce ad arrivare all'estremità opposta della scacchiera deve essere sostituito da una pedina a scelta tra regina,alfiere,torre,cavallo
- ***En passant***: se un pedone eseguendo la mossa di due caselle in avanti si trova a fianco di un pedone avversario,alla mossa successiva quest'ultimo può essere catturato come se si fosse mosso di una sola casella
- ***Arrocco***: per una volta sola il re può muoversi di due caselle a sinistra o a destra in direzione di una delle due torri e la torre verso cui si è mosso il re deve essere frapposta tra la casella di partenza del re e quella d'arrivo.Questo evento si ha quando sono soddisfatte tre condizioni:
    - 1)Torre e re non sono stati mossi
    - 2)Non vi sono pezzi frapposti tra torre e re
    - 3)Le caselle occupate dal re prima e dopo il movimento(inclusa quella di mezzo) non devono essere sotto scacco

- ***Scacco***: se dopo il turno di un giocatore la traiettoria di una pedina incrocia la casella in cui è situato il re avversario,quel re si dice “sotto scacco” e l'avversario deve necessariamente muoverlo in una casella vuota che non sia a sua volta nella traiettoria di una pedina del colore opposto
- ***Scacco matto***: se un re sotto scacco non può muoversi in alcuna casella avviene lo scacco matto.Il giocatore sotto scacco matto perde la partita

L'interfaccia grafica dovrà essere minimale e dovranno essere mostrati dei messaggi in caso di scacco o scacco matto,si richiede inoltre un messaggio che avvisi di quale giocatore sia il turno.

Per quanto riguarda la scacchiera,la casella inferiore sinistra dovrà essere nera,e i pezzi dovranno essere disposti secondo tale schema:

Torre	Cavallo	Alfiere	Re	Regina	Alfiere	Cavallo	Torre
Pedone	Pedone	Pedone	Pedone	Pedone	Pedone	Pedone	Pedone
Pedone	Pedone	Pedone	Pedone	Pedone	Pedone	Pedone	Pedone
Torre	Cavallo	Alfiere	Re	Regina	Alfiere	Cavallo	Torre

I pezzi mangiati devono essere visibili ai lati della scacchiera,separati tra di loro in base al colore.

Non è richiesta la possibilità di chiedere la resa. Si eviti l'implementazione della raccolta di statistiche relative alle partite o di un qualsiasi tipo di dato.

Si richiede che il programma sia scritto e commentato in lingua inglese per favorirne le migliorie da parte della community di git nell'ottica del codice open source

Specifica dei requisiti

Specifica dei requisiti:

Di seguito vi somo riportate le tabelle dei casi d'uso inerenti ai diagrammi,gli attori sono i due giocatori della partita,uno controllante i pezzi bianchi(che muoverà per primo) e l'altro i pezzi neri.

Come da specifica,all'avvio dell'applicazione viene disegnata direttamente la scacchiera con i 32 pezzi: sedici bianchi(nel dettaglio:un re,una regina,due alfieri,due cavalli,due torri ed otto pedoni) e sedici neri(in numero analogo a quelli bianchi).Il giocatore all'inizio potrà effettuare tre mosse:

- Movimento del cavallo
- Avanzamento del pedone di una casella
- Avanzamento del pedone di due caselle

Il movimento del pedone di due caselle è permesso solamente se lo specifico pedone non è mai stato mosso,come da regolamento degli scacchi.

Abbiamo quattro casi d'uso generali: la selezione della pedina da muovere,la deselezione di una pedina selezionata , la selezione della casella in cui la si vuole spostare(se il movimento è permesso dalle regole) e la selezione di una casella non permessa

CASO D'USO	Selezione di una pedina
ID	00
ATTORE	Giocatore
PRECONDIZIONE	Aver avviato il gioco
CORSO D'AZIONE BASE	La pedina selezionata viene evidenziata inscrivendola in una circonferenza
POSTCONDIZIONE	La pedina può essere mossa
PERCORSO ALTERNATIVO	1)Viene selezionata una casella vuota e non accade nulla. 2)La pedina viene deselezionata

CASO D'USO	Deselezione di una pedina
ID	01
ATTORE	Giocatore
PRECONDIZIONE	Aver selezionato una pedina
CORSO D'AZIONE BASE	La pedina selezionata viene deselezionata
POSTCONDIZIONE	Si può selezionare un'altra pedina
PERCORSO ALTERNATIVO	Muovere la pedina selezionata

CASO D'USO	Movimento della pedina selezionata
ID	02
ATTORE	Giocatore
PRECONDIZIONE	Aver selezionato una pedina da muovere
CORSO D'AZIONE BASE	La pedina selezionata si muove in un'altra casella
POSTCONDIZIONE	Inizia il turno del giocatore sucessivo
PERCORSO ALTERNATIVO	Viene selezionata una casella in cui non è possibile muovere la pedina e viene mostrato un messaggio d'errore

CASO D'USO	Movimento non consentito
ID	02-bis
ATTORE	Giocatore
PRECONDIZIONE	Aver selezionato una pedina da muovere
CORSO D'AZIONE BASE	Si tenta di muovere la pedina selezionata in una casella in cui non può spostarsi
POSTCONDIZIONE	Il movimento non viene effettuato
PERCORSO ALTERNATIVO	Viene selezionata una casella in cui è possibile muovere la pedina

Qui di seguito vengono riportati i 3 casi d'uso dei movimenti che possono essere effettuati durante il primo turno:

CASO D'USO	Avanzamento del pedone di una casella(primo turno)
ID	03
ATTORE	Giocatore
PRECONDIZIONE	Aver selezionato un pedone
CORSO D'AZIONE BASE	1)Il pedone selezionato si muove verticalmente nella casella successiva scelta dal giocatore tramite il click del mouse
POSTCONDIZIONE	Inizia il turno del giocatore successivo
PERCORSO ALTERNATIVO	1)Il giocatore cerca di muovere il pedone in una casella non consentita,quindi viene mostrato un messaggio d'errore 2)Il giocatore muove il pedone di due caselle 3)Il giocatore muove il cavallo

CASO D'USO	Movimento del pedone di due caselle(primo turno)
ID	04
ATTORE	Giocatore
PRECONDIZIONE	Aver selezionato un pedone
CORSO D'AZIONE BASE	1)Il pedone selezionato si muove nella seconda casella successiva scelta dal giocatore tramite il click del mouse
POSTCONDIZIONE	Inizia il turno del giocatore successivo
PERCORSO ALTERNATIVO	1)Il giocatore cerca di muovere il pedone in una casella non consentita,quindi viene mostrato un messaggio d'errore 2)Il giocatore muove il pedone di una casella 3)Il giocatore muove il cavallo

CASO D'USO	Movimento del cavallo(primo turno)
ID	05
ATTORE	Giocatore
PRECONDIZIONE	Aver selezionato un cavallo
CORSO D'AZIONE BASE	1)Il cavallo viene mosso nella casella scelta dal giocatore tramite click del mouse
POSTCONDIZIONE	Il turno passa al giocatore sucessivo
PERCORSO ALTERNATIVO	1)Il giocatore cerca di muovere il cavallo in una casella non consentita,quindi viene mostrato un messaggio d'errore 2)Il giocatore muove il pedone

Questi sono gli unici movimenti permessi al primo turno.I movimenti che può effettuare il secondo giocatore sono i medesimi che può effettuare il primo giocatore.di seguito vengono illustrati i casi d'uso dei movimenti delle restanti pedine in momenti indefiniti della partita(torre,alfiere,re e regina).
Per completezza verranno illustrati anche i movimenti di pedone e cavallo in momenti qualsiasi del gioco

CASO D'USO	Movimento della torre
ID	06
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato una torre 2)Non vi devono essere pedine dello stesso colore nella traiettoria del movimento
CORSO D'AZIONE BASE	1)La torre viene mossa verticalmente sulla casella selezionata dal giocatore 2)La torre viene mossa orizzontalmente sulla casella selezionata dal giocatore
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	La torre viene mossa in una casella non consentita

CASO D'USO	Movimento dell' alfiere
ID	07
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato un alfiere 2)Non vi devono essere pedine dello stesso colore nella traiettoria del movimento
CORSO D'AZIONE BASE	L'alfiere viene mosso in una posizione selezionata dall'utente, diagonale rispetto la sua posizione di partenza
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	L' alfiere viene mosso in una casella non consentita

CASO D'USO	Movimento del re
ID	08
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato il re 2)Non vi devono essere pedine dello stesso colore nella traiettoria del movimento 3)La casella d'arrivo non deve essere nella traiettoria di una pedina avversaria
CORSO D'AZIONE BASE	Il re viene mosso radialmente di una casella
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	1)il re viene mosso in una casella non consentita. 2)Il re non può muoversi.Qui distinguiamo 3 casi: 2.1)Il re è nella traiettoria di una pedina avversaria e non vi sono pedine che possono difenderlo(scacco matto) 2.2) il re è “circondato” da caselle nella traiettoria di pedine avversarie e non vi sono altre pedine da muovere(partita patta) 2.3)Il re non può muoversi ma vi sono altre pedine che possono

CASO D'USO	Movimento della regina
ID	09
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato una regina 2)Non vi devono essere pedine dello stesso colore nella traiettoria del movimento
CORSO D'AZIONE BASE	La regina viene mossa in una posizione scelta dall'utente,tale che la casella d'arrivo si trovi orizzontalmente,verticalmente o diagonalmente rispetto la casella di partenza
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	La regina viene mossa in una casella non consentita

CASO D'USO	Avanzamento del pedone di una casella(momento qualsiasi)
ID	10
ATTORE	Giocatore
PRECONDIZIONE	Aver selezionato un pedone
CORSO D'AZIONE BASE	1)Il pedone selezionato si muove verticalmente nella casella successiva selezionata dal giocatore tramite il click del mouse 2)Non vi devono essere pedine dello stesso colore nella casella d'arrivo
POSTCONDIZIONE	Inizia il turno del giocatore successivo
PERCORSO ALTERNATIVO	1)Il giocatore cerca di muovere il pedone in una casella non consentita,quindi viene mostrato un messaggio d'errore 2)Nella casella in cui il pedone è mosso vi è una pedina avversaria o alleata,quindi il movimento è negato

CASO D'USO	Movimento del pedone di due caselle(momento qualsiasi della partita)
ID	11
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato un pedone 2)Il pedono non deve essere mai stato mosso
CORSO D'AZIONE BASE	1)Il pedone selezionato si muove verticalmente nella seconda casella successiva scelta dal giocatore tramite il click del mouse 2)Non vi devono essere pedine dello stesso colore nella traiettoria del movimento
POSTCONDIZIONE	Inizia il turno del giocatore successivo
PERCORSO ALTERNATIVO	1)Il giocatore cerca di muovere il pedone in una casella non consentita,quindi viene mostrato un messaggio d'errore 2)Nelle caselle lungo la traiettoria del pedone vi è una pedina avversaria o alleata,quindi il movimento è negato

CASO D'USO	Movimento del cavallo(momento qualsiasi)
ID	12
ATTORE	Giocatore
PRECONDIZIONE	Aver selezionato un cavallo
CORSO D'AZIONE BASE	1)Il cavallo viene mosso (con il movimento ad elle) nella casella scelta dal giocatore tramite click del mouse
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	1)Il giocatore cerca di muovere il cavallo in una casella non consentita,quindi viene mostrato un messaggio d'errore

Dopo aver descritto i casi d'uso del movimento,di seguito verranno descritti i casi d'uso relativi al metodo di cattura dei singoli pezzi.
Eccezion fatta per il pedone(che può catturare una pedina avversaria solo diagonalmente verso la direzione dell'avversario)le caselle in cui un pezzo può catturarne un altro sono le stesse in cui quel dato pezzo può muoversi

CASO D'USO	Modalità di cattura del pedone
ID	13
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato un pedone 2)Avere lungo una delle due caselle diagonali adiacenti (le due caselle diagonali rivolte verso la parte opposta a quella di partenza)una pedina avversaria da catturare
CORSO D'AZIONE BASE	1)Il pedone selezionato si muove nella casella selezionata dal giocatore tramite il click del mouse mangiando la pedina avversaria,che verrà spostata nella zona apposita per le pedine mangiate
POSTCONDIZIONE	Inizia il turno del giocatore successivo
PERCORSO ALTERNATIVO	1)Il pedone si muove verticalmente di una casella 2)Si muove verticalmente di due caselle(se non è stato mosso)

CASO D'USO	Modalità di cattura del cavallo
ID	14
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato un cavallo 2)Avere una pedina avversaria su una delle 8 caselle in cui il cavallo può muoversi
CORSO D'AZIONE BASE	1)Il cavallo selezionato si muove nella casella selezionata dal giocatore tramite il click del mouse mangiando la pedina avversaria,che verrà spostata nella zona apposita per le pedine mangiate
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	1)Il giocatore muove il cavallo in una casella non occupata da una pedina

CASO D'USO	Modalità di cattura della torre
ID	15
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato una torre 2)Non vi devono essere pedine nella traiettoria del movimento verso il pezzo da mangiare
CORSO D'AZIONE BASE	1)La torre selezionata si muove nella casella selezionata dal giocatore tramite il click del mouse mangiando la pedina avversaria,che verrà spostata nella zona apposita per le pedine mangiate
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	La torre viene mossa in una casella non occupata da una pedina

CASO D'USO	Modalità di cattura dell' alfiere
ID	16
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato un alfiere 2)Non vi devono essere pedine nella traiettoria del movimento verso il pezzo da mangiare
CORSO D'AZIONE BASE	L'alfiere selezionato si muove nella casella selezionata dal giocatore tramite il click del mouse mangiando la pedina avversaria,che verrà spostata nella zona apposita per le pedine mangiate
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	L' alfiere viene mosso in una casella non occupata da una pedina

CASO D'USO	Modalità di cattura del re
ID	17
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato il re 2)La casella d'arrivo non deve essere nella traiettoria di una pedina avversaria
CORSO D'AZIONE BASE	Il re si muove nella casella selezionata dal giocatore tramite il click del mouse mangiando la pedina avversaria,che verrà spostata nella zona apposita per le pedine mangiate
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	1)il re tenta di mangiare una pedina che si trova nella traiettoria di un'altra pedina avversaria(movimento non consentito) 2)Il re si muove in una casella non occupata da una pedina

CASO D'USO	Modalità di cattura della regina
ID	18
ATTORE	Giocatore
PRECONDIZIONE	1)Aver selezionato una regina 2)Non vi devono essere pedine nella traiettoria del movimento verso la pedina da mangiare
CORSO D'AZIONE BASE	La regina selezionato si muove nella casella selezionata dal giocatore tramite il click del mouse mangiando la pedina avversaria,che verrà spostata nella zona apposita per le pedine mangiate
POSTCONDIZIONE	Il turno passa al giocatore successivo
PERCORSO ALTERNATIVO	La regina viene mossa in una casella non occupata da una pedina

Ora verranno descritti i casi d'uso delle cinque regole di cui è espressamente richiesta l'implementazione:

- Promozione del pedone
- En passant
- Arrocco
- Scacco
- Scacco matto

CASO D'USO	Promozione del pedone
ID	19
ATTORE	Giocatore
PRECONDIZIONE	1)Avere un pedone nella penultima riga del campo avversario. 2)La casella frontale deve essere libera
CORSO D'AZIONE BASE	Il pedone si sposta nell'ultima riga del campo
POSTCONDIZIONE	Il pedone deve diventare una pedina a scelta tra torre,alfiere,cavallo o regina
PERCORSO ALTERNATIVO	1)Viene mossa un'altra pedina

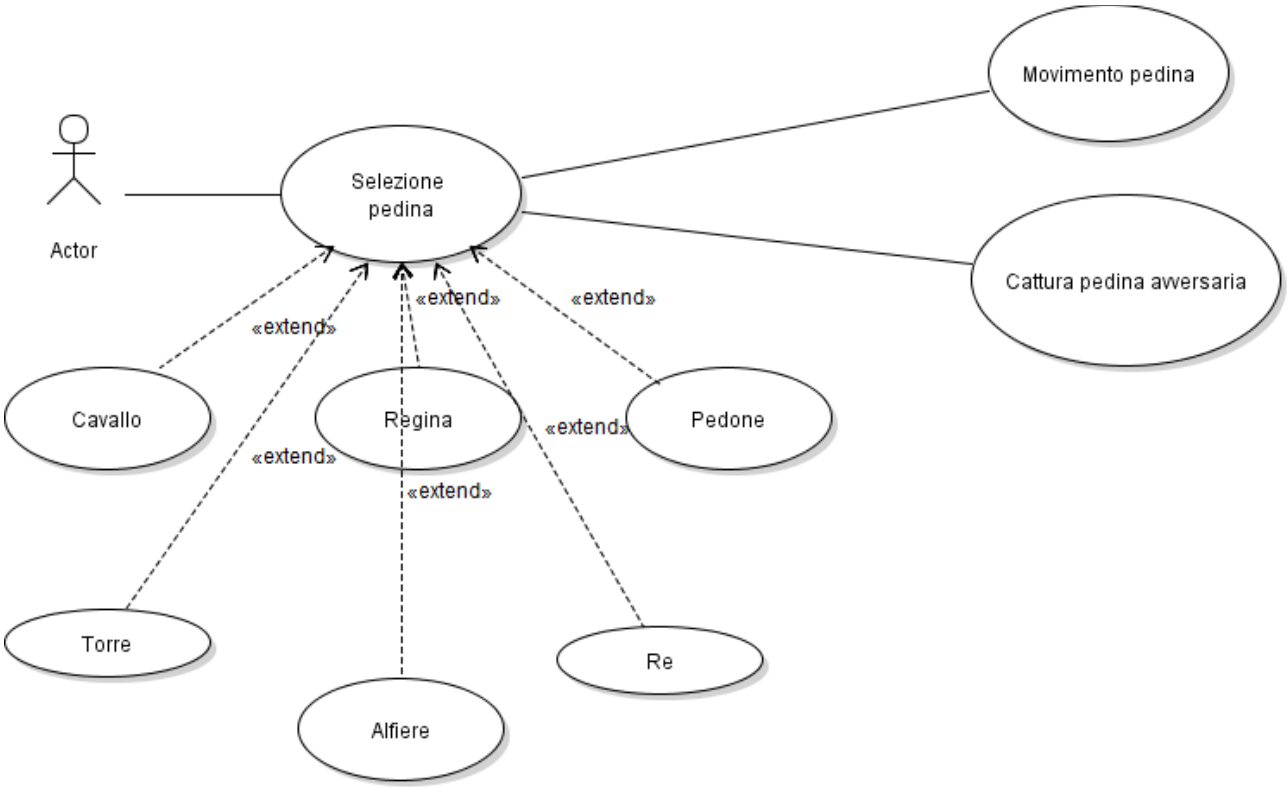
CASO D'USO	En passant
ID	20
ATTORE	Giocatore
PRECONDIZIONE	1)Il pedone deve trovarsi a fianco di un pedone avversario che ha appena effettuato il movimento di due caselle
CORSO D'AZIONE BASE	Il pedone si sposta diagonalmente di una casella nella direzione del pedone avversario in modo da trovarsi dietro a quest'ultimo.
POSTCONDIZIONE	Il pedone avversario viene mangiato
PERCORSO ALTERNATIVO	1)Il giocatore muove un'altra pedina

CASO D'USO	Arrocco
ID	21
ATTORE	Giocatore
PRECONDIZIONE	1)Il re e la torre alleata posizionata nel lato in cui si vuole effettuare l'arrocco non devono essere stati mossi. 2)Le caselle tra loro devono essere libere e non minacciate 3)Il re non deve essere minacciato
CORSO D'AZIONE BASE	Il giocatore seleziona il re e poi seleziona la casella a lato della torre interessata
POSTCONDIZIONE	Il re si sposta di due caselle nella direzione della torre alleata designata,la torre si frappone tra la casella di partenza del re e la casella d'arrivo
PERCORSO ALTERNATIVO	

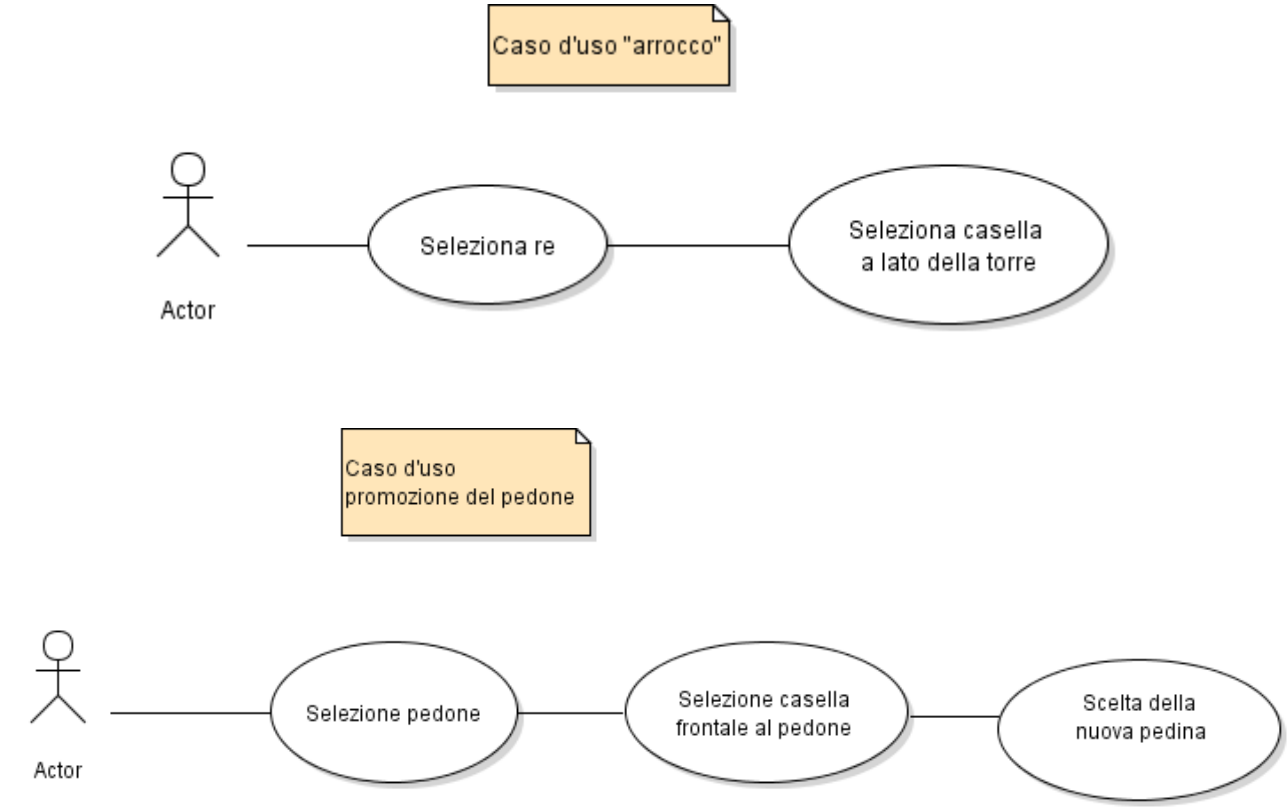
CASO D'USO	Scacco
ID	22
ATTORE	Giocatore
PRECONDIZIONE	1) Il re è minacciato da una pedina avversaria
CORSO D'AZIONE BASE	Il giocatore deve necessariamente spostare il re in una casella che non sia nella traiettoria di una pedina avversaria o frapporre una pedina tra il re e la minaccia
POSTCONDIZIONE	Il re non è più minacciato
PERCORSO ALTERNATIVO	Il giocatore non può muovere il re in caselle non minacciate(scacco matto)

CASO D'USO	Scacco matto
ID	23
ATTORE	Giocatore
PRECONDIZIONE	1)Il re è minacciato da una pedina avversaria
CORSO D'AZIONE BASE	Il giocatore non può muovere il re o frapporre una pedina tra quest'ultimo e la pedina che lo minaccia
POSTCONDIZIONE	Il re si sposta di due caselle nella direzione della torre alleata designata,la torre si frappone tra la casella di partenza del re e la casella d'arrivo
PERCORSO ALTERNATIVO	

Di seguito viene illustrato lo svolgimento tipico di un turno di gioco tramite un diagramma UML dei casi d'uso



Per completezza vengono illustrati anche i casi d'uso relativi ad arrocco e promozione del pedone



Analisi e progettazione

Analisi e progettazione:

Linguaggio di programmazione:

Per la realizzazione dei requisiti richiesti si è scelto di adottare il linguaggio di programmazione ad oggetti **C#**.
Un linguaggio di programmazione ad oggetti permette di esprimere questi ultimi in classi raggruppanti metodi ed attributi propri degli oggetti da usare(un oggetto a sua volta è un'entità descritta da una classe).

Questo linguaggio permette di sfruttare le seguenti proprietà:

- **Information hiding**: principio alla base dell'incapsulamento secondo cui i dettagli dell'implementazione di una classe sono nascosti all'utente finale
- **Incapsulamento**: tecnica di nascondere il funzionamento di parti dello stesso programma tra loro al fine di evitare la ripercussione di errori
- **Ereditarietà**: Implementare classi “figlie” di una classe generale specializzandone le caratteristiche
- **Polimorfismo**: garanzia della medesima interfaccia in più oggetti di tipo differente

Una particolare menzione viene fatta all'**overload** dei metodi(avviene quando vi sono due metodi con lo stesso nome ma con un argomento diverso),ai **delegati** (Tipi che incapsulano in modo sicuro un metodo) e alle **classi parziali**(classi definite su due o più file),caratteristiche sfruttate in questo progetto.

Ambiente di sviluppo:

Per lo sviluppo del programma sono stati usati gli IDE(Integrated Development Environment) Xamarin Studio 5.10 e Monodevelop 5.10.1 entrambi con framework mono 4.0.1.Si sono scelti questi due ambienti per via della compatibilità con Archlinux ed Ubuntu,sistemi sul quale è stato scritto il programma.

GUI:

È stato scelto di usare le librerie GTK# per creare l'interfaccia grafica richiesta per via della loro compatibilità con Monodevelop e Xamarin e per il loro stile nativo Gnome.La specifica richiede un'interfaccia minimale,per disegnare la scacchiera sono state usate delle caselle bianche e grigie disposte a colori alternati.Le immagini delle pedine sono state prese dal progetto open source Lichess(https://github.com/ornicar/lila).

Sono state impiegate classi con il seguente ruolo:casella della scacchiera,barra laterale per le pedine rimosse,box per mostrare messaggi all'utente,box per permettere all'utente di scegliere la pedina in cui promuovere il pedone ed una classe griglia per la scacchiera.

Dopo un'analisi iniziale appare evidente che si necessita di una classe per la scacchiera,una per il giocatore,una per ogni pedina,una per il gioco,una per le coordinate di un oggetto sulla scacchiera ed una per i messaggi da inviare al giocatore. In seguito si è scelto di inserire una classe atta a veicolare vari tipi di informazioni.

Partendo dalle pedine è stata creata una classe padre(Figure) da cui derivano le classi per le altre pedine,tra cui è stata aggiunta anche una pedina atta a simulare un “oggetto vuoto”,un oggetto che riempie le caselle vuote della scacchiera.

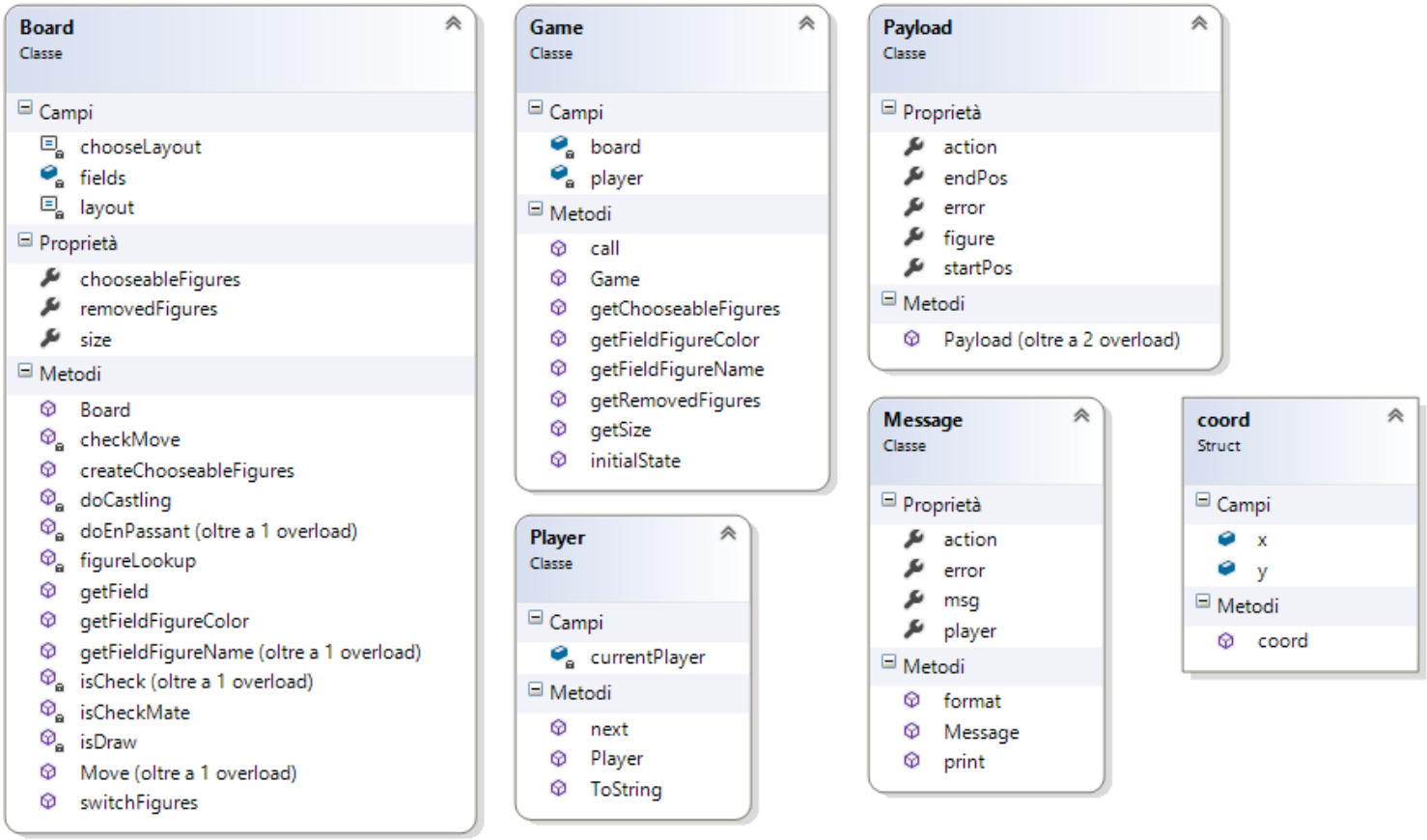
È stato scelto di inserire i metodi principali nella classe della scacchiera(classe Board).

Il file di configurazione per la compilazione su linux è contenuto nella cartella Packages, nella sezione linux binaries(accessibile tramite monodevelop).Questo file permette di creare un file .pkg con tutti i file compilati e con le immagini.

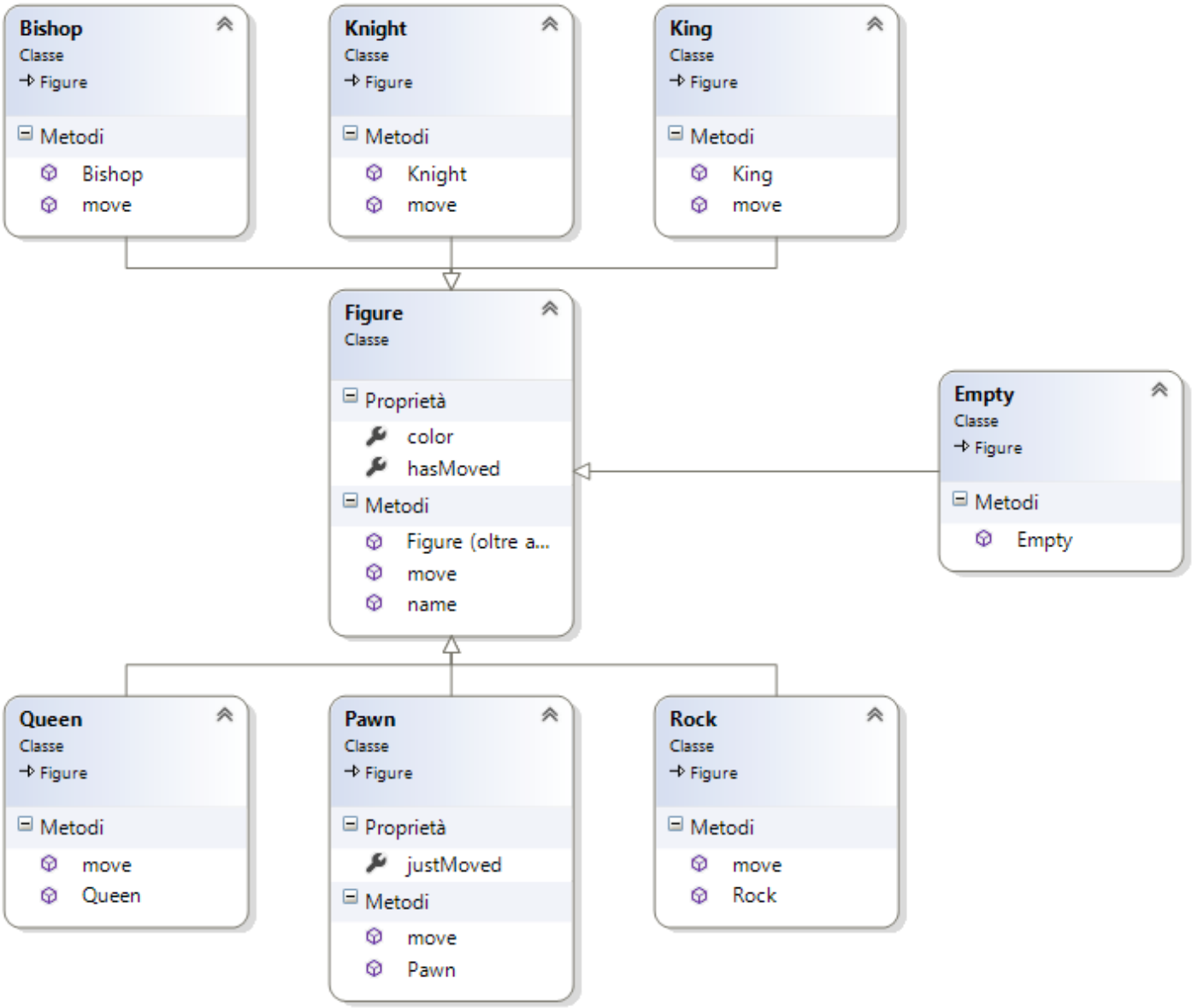
Descrizione delle classi tramite diagrammi UML:

UML sta per Unified Modeling Language ed è un linguaggio di modellizzazione basato sul paradigma ad oggetti. Qui lo useremo per descrivere il funzionamento e la struttura delle classi che compongono il programma.
Ecco una panoramica generale delle classi presenti:

- Classi che compongono il nucleo del programma

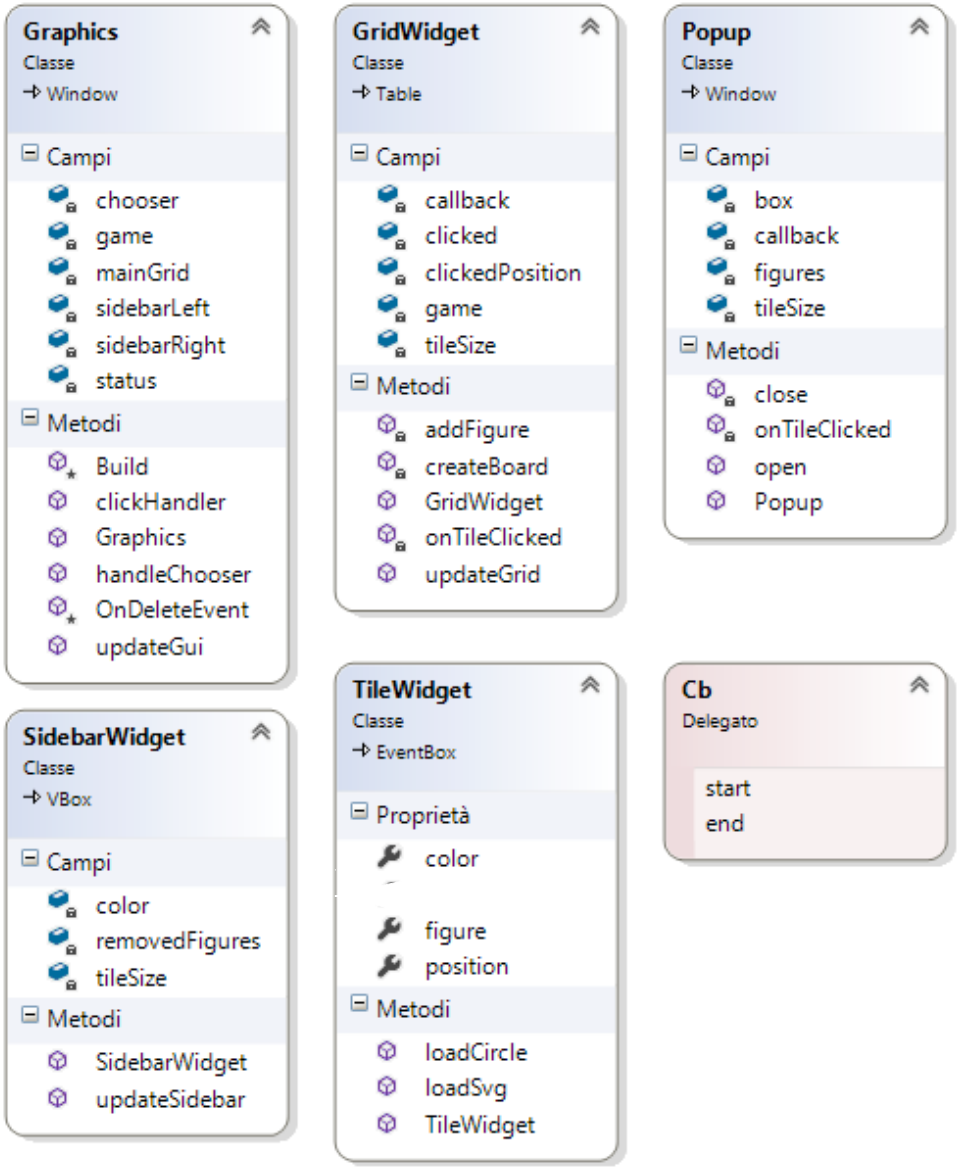


- Classi atte a simulare le pedine



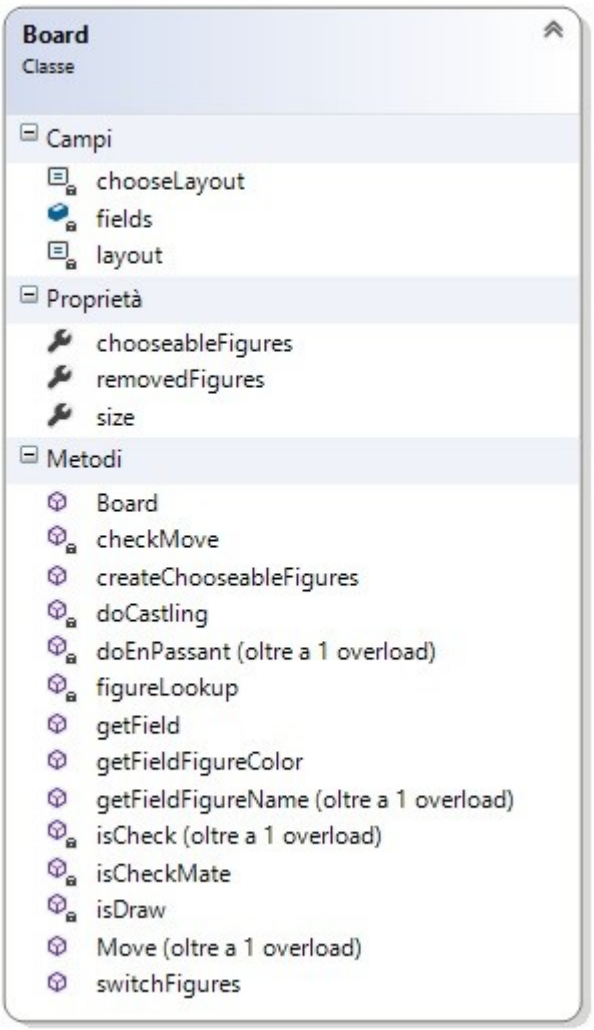
Come si evince dal diagramma,tutte le classi inerenti alle pedine(inclusa la classe empty)sono classi derivate dalla classe padre Figure.Qui è stato sfruttato in modo palese il principio dell'ereditrietà.

- Classi inerenti all'interfaccia grafica



Vediamole ora le classi centrali del programma:

***Board*:**



Classe usata per simulare una scacchiera di 64 caselle (tile) con 8 righe ed 8 colonne a colori bianco e nero alternati.
La classe ha i seguenti campi:

- **chooseLayout**:
  stringa che contiene i caratteri rappresentanti le pedine in cui si può trasformare il pedone dopo la promozione(Torre, Alfiere, Cavallo o Regina, rispettivamente rappresentate dalle lettere R,B,N,Q)

- **fields**:
  attributo rappresentante una casella della scacchiera

- **layout**:
  stringa che contiene i caratteri ,rappresentanti le pedine, ordinati secondo la disposizione di queste ultime sulla scacchiera. Le pedine sono rappresentate dalle seguenti lettere,maiuscole per le pedine bianche,minuscole per quelle nere:
  -K = king (re)
  -Q = queen (regina)
  -N = knight (cavallo)
  -R = rock (torre)
  -B = bishop (alfiere)
  -P = pawn (pedone)
  Ogni altro carattere viene letto dal programma come una casella vuota.Per semplicità,le caselle vuote sono state indicate nella stringa con una X maiuscola.

Di seguito vengono descritte le proprietà:

- **chooseableFigures**:
  Array delle pedine selezionabili

- **removedFigures**:
  Lista delle pedine mangiate,divise per colore

- **size**:
  Attributo per la grandezza della scacchiera

La classe ha i seguenti metodi:

- **Board**:
  Metodo costruttore della classe scacchiera.

- **checkMove:**
  Funzione che controlla se un movimento è effettuabile, come input ha tre campi:
  il giocatore attuale
  la casella in cui si trova la pedina da muovere
  la casella dove deve essere mossa

  Se si tenta di muovere un oggetto vuoto,uns pedina di colore diverso dal proprio o di muovere una propria pedina su una dello stesso colore il risultato è falso.
  Come output ha un booleano che vale vero se la mossa è valida,falso se non lo è.

- **createChooseableFigures:**
  Crea un array di pedine selezionabili senza assegnargli un colore

- **doCastling:**
  Metodo che gestisce l'arrocco.come input ha il giocatore attuale,la posizione iniziale della pedina e la posizione finale.Esso è possibile se e solo se la pedina nella posizione iniziale è un re che non si è ancora mosso.Funziona in 4 fasi:
    1)Sposta il re di una casella verso la torre selezionata
    2)Sposta il re di una seconda casella verso la torre selezionata
    3)Salva in una variabile temporanea la posizione del re per permettere il movimento della torre
    4)Sposta la torre dietro il re,nella casella opposta al verso del movimento
  Il metodo gestisce due casi:quello in cui viene selezionata la torre a destra del re e quello in cui viene selezionata quella a sinistra

- **doEnPassant(overload):**
  Metodo atto a gestire l'en passant.Qui abbiamo un overload:un primo metodo che ha come argomenti il giocatore e le coordinate iniziali e finali ed un secondo che in aggiunta ha un booleano che prova ad eseguire il movimento.
  Il metodo controlla prima che la pedina selezionata sia un pedone e poi il colore di quest'ultimo,in seguito controlla che ci sia un pedone a fianco che abbia effettuato il movimento doppio(contrassegnato dall'attributo justMoved)e se la casella dietro quest'ultimo sia libera.

- **FigureLookup:**
  Metodo che assegna il colore alle pedine e,a seconda della lettera(vedi campo layout),crea una nuova istanza di una pedina(o oggetto vuoto).Ha come input un char(lettera identificante la pedina da creare)e come output un oggetto di classe Figure.

- **getField:**
  Metodo che ha in input un oggetto di tipo coord e in output i valori delle coordinate in x e y.

- **getFieldFigureColor:**
  Metodo che ha in input un oggetto di tipo coord ed in output da il colore dell'oggetto in quella posizione.

- **getFieldFigureName(overload):**
  Metodo per ottenere il nome della pedina in una data casella.Qui abbiamo due metodi: il primo che ottiene il nome della pedina in una casella,ha in input un oggetto di tipo coord. Il secondo ha in input due interi(x ed y) e,invocando il primo metodo descritto sopra,li trasforma in un oggetto di tipo coord

- **isCheck(overload):**
  Metodo che permette di controllare se il re è sotto scacco dopo una data mossa e non permette di effettuarla se lo è. Abbiamo un overload:
  Il primo metodo prende un oggetto di tipo giocatore, coordinate iniziali e finali.
  Crea un oggetto coord chiamato king,un booleano chiamato noKing di valore true,un nuovo oggetto di tipo Figure(removedObj) ed un booleano(res)impostato su falso.

  Se le coordinate di inizio e fine sono diverse,removedObj assume i valori di x ed y finali,in seguito i valori iniziali diventano i nuovi valori finali(di x ed y) e nei valori iniziali verrà creato un nuovo oggetto vuoto.
  Il metodo scansiona tutta la scacchiera tramite due cicli annidati e se il colore delle pedine è uguale a quello del giocatore o se vi è il re nelle nuove coordinate,queste ultime vengono assegnate all'oggetto king e noKing viene impostato su falso.

  Se noKing è negato,viene riscansionata la scacchiera per ogni casella in cui res è negata e se il movimento è possibile il re è sotto scacco e la variabile res viene settata su true.

  Se le coordinate di inizio e fine sono nuovamente diverse,le coordinate x ed y finali diventano quelle iniziali e quelle finali prendono il valore di removedObj.

- **isCheckMate:**
  Metodo per controllare se un giocatore è sotto scacco matto,ha come input il giocatore attuale e le coordinate d'inizio del movimento. Di seguito un pezzo del codice:

```
bool res = true;
for (int x = 0; x < this.size.x && res; x++) {
  for (int y = 0; y < this.size.y && res; y++) {
    coord end = new coord (x, y);
    if (checkMove (player, start, end)) {
      res = isCheck (player, start, end);
         }
 }
}
```

  Viene creata una nuova variabile booleana(res) che,se vale false,indica che è possibile muovere il re.
  Vengono scansionate le caselle in cui res è vera e si creano nuove coordinate finali per ognuna di queste e si richiama il metodo isCheck all'interno del metodo checkMove per vedere se ogni casella in cui è spostabile il re è sotto scacco.La variabile res prende il valore di ritorno di isCheck e se è falsa è possibile muovere il re.

```
 if (res) {
      if (start.x < this.size.x - 1) {
         res = isCheckMate (player, new coord (start.x + 1, start.y));
        } else if (start.y < this.size.y - 1) {
    res = isCheckMate (player, new coord (0, start.y + 1));
  }
 }
return res;
```

  Se res è vera vi è un altro ciclo e se la posizione iniziale è minore della grandezza di x(o di y) meno uno assegna a res il valore di ritorno di isCheckMate.

- **IsDraw:**
  Di seguito un pezzo del codice del metodo:

```
 if ((doEnPassant (player, start, end, true)) ||
(checkMove (player, start, end) && !isCheck (player, start, end))) {

       return false;
}
```

  Metodo che muove le pedine,cerca i possibili movimenti,specificatamente l'En passant (o checkMove e isCheck negato)e ritorna falso se ciò è vero,mentre ritorna vero se la posizione iniziale è minore della grandezza di x(o di y) meno uno

- **Move(overload):**
  Metodo che ritorna un oggetto di tipo messaggio.Abbiamo un overload:
  -Il primo metodo ritorna un messaggio nel terminale se si seleziona una casella vuota o se il colore della pedina selezionata è diverso da quello del giocatore.
  Ha come input il giocatore attuale e le coordinate di inizio del movimento
  -Il secondo metodo è più complesso:
  ha in input giocatore,coordinate iniziali e finali.Ritorna il risultato di:
    -doEnPassant
    -doCastling(arrocco)
    -checkMove(se isCheck è falso),se la casella d'arrivo non è vuota,la pedina presente viene aggiunta alle pedine rimosse

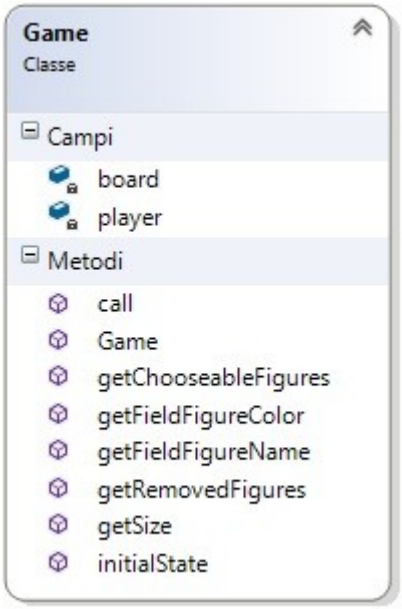
  Dopo queste operazioni pone nelle coordinate d'arrivo la pedina presente nelle coordinate di partenza e crea nelle coordinate di partenza un oggetto vuoto.
  In seguito vi è un controllo sul movimento:se quest'ultimo porta il giocatore successivo ad essere in stato di scacco o scacco matto viene stampato un messaggio.

  Come ultime operazioni vengono rimossi gli attributi justMoved(usati per l'en passant) di un eventuale pedone che ha effettuato il movimento di due caselle nel turno precedente e viene impostato su vero l'attributo justMoved di un eventuale pedone che ha appena effettuato il movimento di due caselle. Viene ritornato result.

  Se non è stato ritornato il risultato dei primi quattro casi,viene impostato su true l'attributo error di result e viene ritornato.

- **switchFigures:**
  Metodo atto a cambiare la posizione tra due oggetti pedina durante un movimento,generalmente viene sostituito un oggetto vuoto ad una pedina;
  ha in input un oggetto pedina ed un oggetto coord.
  Se il giocatore successivo è sotto scacco o scacco matto viene stampato a video il messaggio corrispondente.

Game:



Classe che richiama i metodi e le classi necessari a creare un'istanza del gioco vero e proprio.È la classe principale del programma.
I campi sono i seguenti:

- **board:**
  Campo di tipo Board usato per la scacchiera.

- **player:**
  Campo di tipo Player usato per il giocatore

La classe ha i seguenti metodi:

- **call:**
  Metodo che da in output un oggetto di tipo message e prende in input un oggetto di tipo payload chiamato data.Se in data error è negativo abbiamo tre casi:
  -caso switchFigures
  -caso move (viene bloccato il gioco in attesa del click dell'utente.Se non ci sono errori il turno passa al giocatore successivo).
  -caso checkSelection
  Infine viene ritornato result(oggetto di tipo message nominato all'inizio)

- **Game:**
  Metodo costruttore della classe Game. Istanzia un nuovo oggetto di classe Board ed un nuovo oggetto di classe Player(quest'ultimo impostato su white).
- **getChooseableFigures:**
  Metodo di get che ritorna un array di pedine selezionabili.

- **getFieldFigureColor:**
  Metodo di get che ritorna il colore della pedina su una data casella.

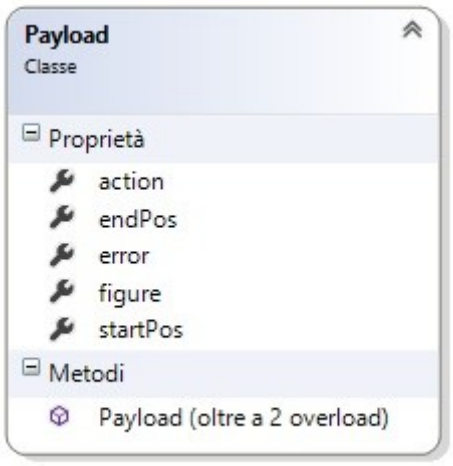
- **getFieldFigureName:**
  Metodo di get che ritorna il nome della pedina in una data casella.

- **getRemovedFigures:**
  Metodo di get che ritorna le pedine mangiate.Ha in output la lista di pedine.

- **getSize:**
  Metodo di get che ritorna la grandezza della scacchiera(generalmente 8x8).Ritorna un tipo coord.

- **initialState:**
  Metodo che ritorna un messaggio di error impostato su falso e come azione ha quella di selezionare il giocatore succcessivo.

Payload:



Oggetto con il compito di portare le informazioni descritte dalle seguenti proprietà:

- **action:**
  Stringa per l'azione da eseguire.

- **endPos:**
  Oggetto coord per la posizione finale di un movimento.

- **error:**
  Booleano per rappresentare un errore.

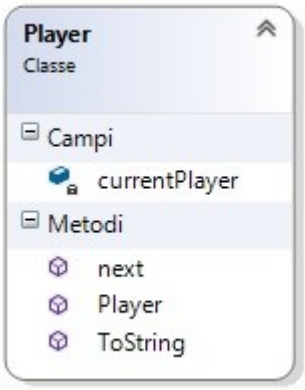
- **figure:**
  Oggetto figure per una pedina.

- **startPos:**
  Oggetto coord per la posizione iniziale di un movimento.

La classe ha i seguenti metodi:

- **Payload(due overload):**
  Abbiamo tre tipi di payload, aventi in input gli oggetti error ed action ed altri oggetti descritti di seguito:
  -Payload per la risposta della scelta ha in input una pedina e un oggetto coord
  -Payload per il comando del movimento ha in input due oggetti coord rappresentanti inizio e fine del movimento
  -Payload per lo scacco ha in input un oggetto coord contenente le coordinate di start

Player:



Classe usata per rappresentare un giocatore.
La classe ha i seguenti campi:

- **currentPlayer:**
  Stringa che identifica il colore del giocatore attuale

La classe ha i seguenti metodi:

- **next:**
  Metodo per alternare i giocatori

- **Player:**
  Metodo costruttore che assegna il player attuale alla stringa currentPlayer

- **ToString:**
  Override del metodo ToString,ha in output la stringa che rappresenta il giocatore attuale

Message:

Classe usata per inviare messaggi al giocatore,ha 4 proprietà principali:

- **action:**
  Stringa contenente l'azione da eseguire

- **error:**
  Booleano con valore vero se vi è un errore

- **msg:**
  Stringa con il messaggio da passare al giocatore

- **player:**
  Oggetto designante il giocatore interessato

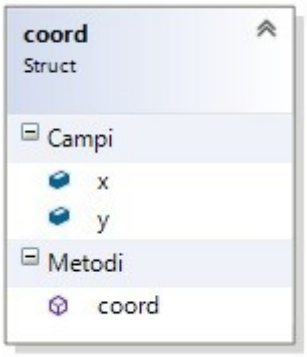
La classe ha i seguenti metodi:

- **format:**
  Metodo contenente i messaggi da inviare.nel caso base viene inviato un messaggio con scritto il colore del giocatore di turno,in caso di giocatore sotto scacco o sotto scacco matto quest'ultimo viene avvertito con un messaggio e della sua sconfitta nel secondo caso.

- **Message:**
  Metodo costruttore della classe.

- **Print:**
  Metodo che scrive nella console se vi è un errore(campo error impostato a true,altrimenti a false) e stampa un eventuale messaggio msg.

Coord:



Oggetto contenente le coordinate x(orizzontali) ed y(verticali) di una casella della scacchiera.
La classe ha i seguenti campi:

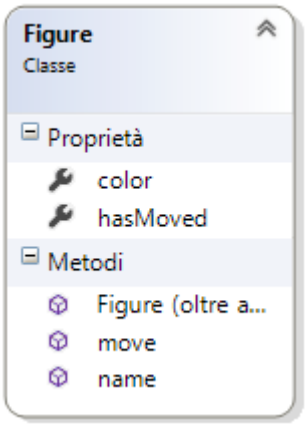
- **x:**
  Rappresenta la coordinata orizzontale,variabile da 0 a 7
- **y:**
  Rappresenta la coordinata verticale, variabile da 0 a 7

La classe ha i seguenti metodi:

- **coord:**
  Metodo costruttore della classe assegnante ad x il valore dell'attuale x e ad y il valore dell'attuale y.

Di seguito la descrizione delle classi relative alle pedine:

Figure:



Classe padre di tutte le pedine.
Di seguito vengono descritte le proprietà:

- **color**:
  Stringa memorizzante il colore della pedina

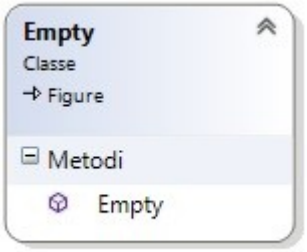
- **hasMoved**:
  Booleano per vedere se la pedina si è mossa

La classe ha i seguenti metodi:

- **Figure(overload)**:
  Metodo costruttore della classe Figure. Ne abbiamo due: il primo che ha per argomento la stringa che identifica il colore della pedina ed il secondo che non ha argomenti.

- **Move**
  Metodo per il movimento di una pedina,ha in input un oggetto di tipo Board e due oggetti di tipo Coord rappresentanti le coordinate iniziali e quelle finali del movimento
- **name**
  Metodo che ha come output il nome della pedina

Empty:



Classe usata per un oggetto vuoto,ossia un oggetto usato per riempire le caselle vuote della scacchiera
La classe ha un solo metodo:

- **Empty**:
  Metodo costruttore della classe rappresentante una casella vuota

Bishop:

Oggetto per simulare un alfiere.
La classe ha i seguenti metodi:

- **Bishop**:
  Metodo costruttore della classe rappresentante l'alfiere

- **move:**
  Metodo che incorpora le regole di movimento dell'alfiere:movimento in diagonale di un numero variabile di caselle

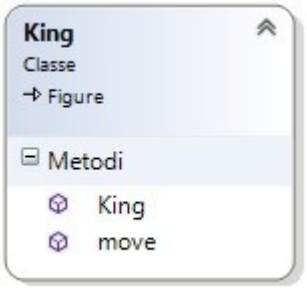
Knight:

Oggetto per simulare un cavallo.
La classe ha i seguenti metodi:

- **Knight**:
  Metodo costruttore della classe rappresentante il cavallo

- **move:**
  Metodo che incorpora le regole del tipico movimento “ad elle” del cavallo

King:

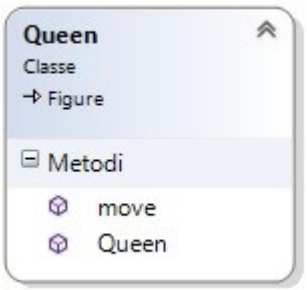


Oggetto per simulare un re.
La classe ha i seguenti metodi:

- **King**:
  Metodo costruttore della classe rappresentante il re

- **move:**
  Metodo di movimento del re:una casella in direzione radiale

Queen:

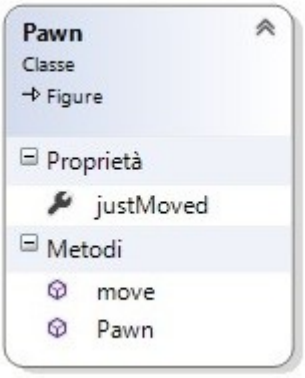


Oggetto per simulare una regina.
La classe ha i seguenti metodi:

- **Queen**:
  Metodo costruttore della classe rappresentante la regina

- **move:**
  Metodo che incorpora il movimento della regina,è una combinazione del metodo di movimento della torre e di quello dell'alfiere

Pawn:



Oggetto per simulare un pedone.
Di seguito vengono descritte le proprietà:

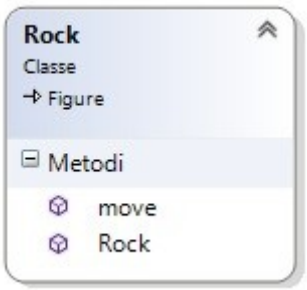
- **justMoved**:
  Valore booleano che cambia se il pedone è stato mosso, serve per il movimento iniziale di due caselle effettuabile solo se il pedone non è stato ancora mosso.

La classe ha i seguenti metodi:

- **Pawn**:
  Metodo costruttore della classe rappresentante il pedone.

- **move:**
  Metodo per il movimento del pedone:di una casella verso il bordo avversario,vi è la possibilità di muoverlo di due caselle se il pedone non è mai stato mosso.

Rock:



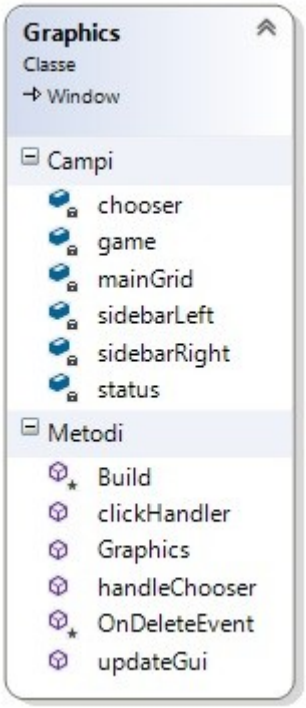
Oggetto per simulare la torre.
La classe ha i seguenti metodi:

- **Rock**:
  Metodo costruttore dell'oggetto rappresentante la torre

- **move:**
  Metodo incorporante il movimento della torre di un numero qualsiasi di caselle orizzontalmente o verticalmente

Vediamo ora le classi usate per l'interfaccia grafica:

Graphics:



Classe principale dell'interfaccia grafica. Si fa uso di un delegato (Cb).
La classe ha i seguenti campi:

- **chooser:**
  Attributo privato di tipo popup.

- **game:**
  Oggetto di tipo game presente nella parte logica del gioco.

- **mainGrid:**
  Oggetto per la griglia principale

- **sidebarLeft:**
  Barra sinistra per le pedine rimosse.

- **sidebarRight:**
  Barra destra per le pedine rimosse.

- **status:**
  Oggetto di tipo label che mostra i messaggi all'utente.

La classe ha i seguenti metodi:

- **Build:**
  Metodo protected e virtual che inizializza l'interfaccia grafica e la mostra all'utente.

- **clickHandler:**
  Metodo che gestisce i click.se non vi è un messaggio d'errore aggiorna l'interfaccia grafica con il nuovo messaggio.

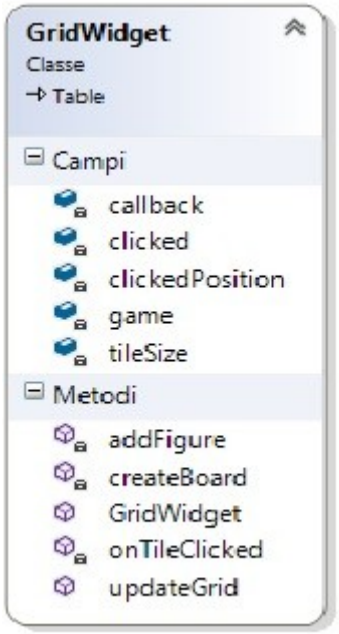
- **Graphics:**
  Metodo costruttore della classe grafica.

- **handleChooser:**
  Metodo che aggiorna l'interfaccia grafica in base ad un nuovo payload.

- **OnDeleteEvent:**
  Metodo per cancellare un evento.

- **updateGui:**
  Metodo che aggiorna le barre laterali,la griglia e gli eventuali messaggi.

GridWidget:



Classe a griglia usata per la scacchiera.
La classe ha i seguenti campi:

- **callback:**
  Attributo contenente la funzione delegata usata quando occorre un evento.

- **clicked:**
  Attributo booleano contenente l'informazione sul click di una casella.

- **clickedPosition:**
  Oggetto coord segnante la posizione cliccata.

- **Game:**
  Istanza di un oggetto di tipo Game.

- **tileSize:**
  Oggetto di tipo coord contenente la grandezza di una casella.

La classe ha i seguenti metodi:

- **addfigure:**
  Metodo per aggiungere una figura nella scacchiera. Invoca il metodo attach della classe di libreria table.

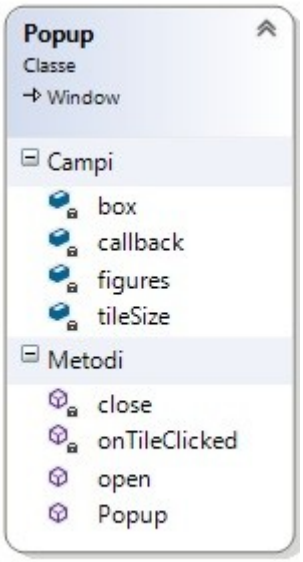
- **createBoard:**
  Metodo per creare la scacchiera,ottiene la grandezza della scacchiera e alterna caselle bianche a caselle grigie.

- **GridWidget:**
  Metodo costruttore,imposta i valori di clicked su false,callback sul callback attuale, game sull'istanza attuale di game e tilesize su un nuovo oggetto coord.

- **onTileClicked:**
  Descrive il comportamento su una casella quando è cliccata,se è presente una pedina appare una circonferenza che la inscrive e scompare se il giocatore clicca di nuovo sulla stessa pedina.

- **UpdateGrid:**
  Metodo che distrugge i widget figli e riinvoca il metodo createBoard.Questo metodo viene invocato dopo ogni movimento.

Popup:



Classe che viene usata durante la scelta della pedina nella promozione del pedone,è una finestra che appare e da la possibilità all'utente di scegliere in quale pedina promuoverlo tramite click del mouse.

La classe ha i seguenti campi:

- **box:**
  Istanza della classe Hbox della libreria Gtk.

- **callback:**
  Delegato di tipo azione che incapsula un metodo con due parametri(in questo caso un oggetto Figure ed uno coord) e che non ritorna un valore.

- **figures:**
  Array di oggetti Figure.

- **tileSize:**
  Oggetto di tipo coord usato per la dimensione del popup su schermo.

La classe ha i seguenti metodi:

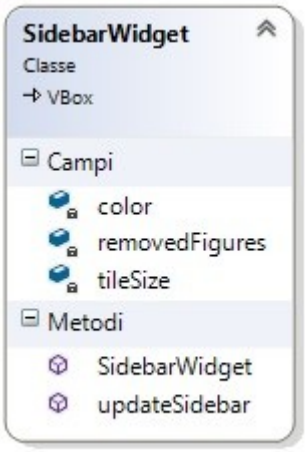
- **close:**
  Metodo per chiudere il popup, distrugge ogni widget figlio ed invoca il metodo Hide per nascondersi.

- **onTileClicked:**
  Metodo che descrive il comportamento della casella su cui si clicca.

- **open:**
  Metodo di apertura del popup.

- **PopUp:**
  Metodo costruttore della classe.

SidebarWidget:



Classe grafica parziale figlia della classe Vbox delle librerie gtk usata per rappresentare le sidebar in cui vengono poste le pedine dopo che sono state mangiate.
La classe ha i seguenti campi:

- **color:**
  Attributo per il colore delle pedine che possono essere contenute nella sidebar.

- **removedFigures:**
  Lista delle figure rimosse.

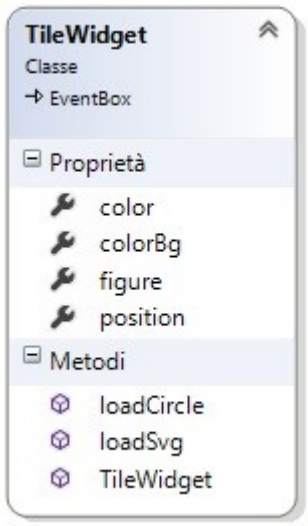
- **tileSize:**
  Oggetto di tipo coord definente la grandezza della casella.

La classe ha i seguenti metodi:

- **SidebarWidget:**
  Metodo costruttore della classe con input dato dalla lista delle figure rimosse,dal loro colore e da un intero usato per definire la sua grandezza su schermo.
  La sidebar ottiene la pedina rimossa e il colore della pedina rimossa(che manterrà),poi viene invocato il metodo updateSidebar..Infine vengono impostate larghezza e grandezza delle caselle.

- **updateSidebar:**
  Metodo per aggiornare la sidebar. Vengono distrutti i widget figli e per ogni figura in removedFigures viene creato un tileWidget inserendolo nel colore adeguato.

TileWidget:



Widget rappresentante una casella. Contiene i metodi per caricare gli sprites delle pedine ee del cerchio di selezione.
Di seguito vengono descritte le proprietà:

- **color:**
  Attributo per il colore

- **figure:**
  Stringa con il nome della pedina nella casella.

- **position:**
  Attributo di tipo coord che definisce la posizione della casella.

La classe ha i seguenti metodi:

- **loadCircle:**
  Metodo per caricare il cerchio di selezione.

- **loadSvg:**
  Metodo per caricare l'immagine di una pedina.

- **TileWidget:**
  Metodo costruttore della classe. Se un oggetto rimosso è vuoto non viene considerato

Program:

Classe per creare il programma,invoca il metodo init e crea una nuova istanza dell'oggetto Game ed infine crea un oggetto Graphics con l'istanza dell'oggetto game come argomento.
La classe termina con l'invocazione del metodo run.

Implementazione

Implementazione:

Di seguito viene riportato il codice del programma:

Classi del nucleo del programma:

Classe board:

```
using System;
using System.Collections.Generic;

namespace Chess
{
  // Class for the board
  public class Board
  {
    //starting layout for the game
    //lower case are black figures,upper case are white figures
    //X or any char different than rnbqkp is considered empty
    const string layout = "RNBKQBNR\n" +
                          "PPPPPPPP\n" +
                          "XXXXXXXX\n" +
                          "XXXXXXXX\n" +
                          "XXXXXXXX\n" +
                          "XXXXXXXX\n" +
                          "pppppppp\n" +
                          "rnbkqbnr";
    /*const string layout = "RNBKQBNR\n" +
                          "PPPPPPPP\n" +
                          "XXXXXXXX\n" +
                          "XXXXXXXX\n" +
                          "XXXXXXXX\n" +
                          "XXXXXXXX\n" +
                          "ppppQppp\n" +
                          "rnbkXXXr";
*/

    //chooseable figures for the pawn promotion
    const string chooseLayout = "RNBQ";
    // Attribute for a tile of the board
    private Figure[,] fields;

    public List<Figure> removedFigures { get; set; }
    // array of chooseable figures
    public Figure[] chooseableFigures { get; set; }
    // attribute for size of board
    public coord size { get; set; }

    // Constructor method
    public Board ()
    {
      removedFigures = new List<Figure> ();
      int lineNumber = 1;  // is 1 because the last line dosen't have a \n
      int lineLength = 0;
      for (int i = 0; i < layout.Length; i++) {
        if (layout [i] == '\n') {
          lineNumber++;
        } else
          lineLength++;
      }// End for
      lineLength /= lineNumber;

      //create array with the right size
```

```
      this.fields = new Figure[lineLength, lineNumber];
      //save size
      this.size = new coord (lineLength, lineNumber);

      //fill up the array with the right figures
      int c = 0;
      for (int y = 0; y < lineNumber; y++) {
        for (int x = 0; x < lineLength; x++) {
          if (layout [c] == '\n')
            c++;
          this.fields [x, y] = figureLookup (layout [c]);
          c++;
        }// End second for
      }// End first for
      this.chooseableFigures = createChooseableFigures ();
    }// End board class

    //create array of the chooseable figures without any color
    public Figure[] createChooseableFigures ()
    {
      Figure[] chooseableFigures = new Figure[chooseLayout.Length];
      for (int i = 0; i < chooseLayout.Length; i++) {
        chooseableFigures [i] = figureLookup (chooseLayout [i]);
        chooseableFigures [i].color = "";
      }
      return chooseableFigures;
    }

    // Method for switching figures on the board
    public Message switchFigures (Figure figure, coord position)
    {
      Player player = new Player (getFieldFigureColor (position));
      Message result = new Message (false, "", "", player);
      this.fields [position.x, position.y] = figure;
      if (isCheck (player.next ())) {
        result.msg = "check";
        if (isCheckMate (player.next (), new coord (0, 0))) {
          result.msg = "checkmate";
        }
      }
      return result;

    }

    // Method that connect lower char with white figures and letters RNBQKP with relative
figures
    private Figure figureLookup (char c)
    {
      Figure result;
      string color = "white";
      if (Char.IsLower (c)) {
        c = Char.ToUpper (c);
        color = "black";
      }
      switch (c) {
      case 'R':
        result = new Rock (color);
        break;
      case 'N':
        result = new Knight (color);
        break;
      case 'B':
        result = new Bishop (color);
        break;
      case 'Q':
```

```
        result = new Queen (color);
        break;
      case 'K':
        result = new King (color);
        break;
      case 'P':
        result = new Pawn (color);
        break;
      default:
        result = new Empty ();
        break;
      }
      return result;
    }

    // Method that return a message
    public Message Move (Player player, coord start)
    {
      return new Message (this.getFieldFigureName (start) == "Empty" ||
this.getFieldFigureColor (start) != player.ToString (), "firstClick", "", player);
    }

    // Method that return some message to player,for example if it is under check or under
checkmate
    public Message Move (Player player, coord start, coord end)
    {
      Message result = new Message (false, "", "", player);

      if (doCastling (player, start, end)) {
        return result;
      } else if (doEnPassant (player, start, end)) {
        return result;
      } else if (checkMove (player, start, end) && !isCheck (player, start, end)) {
        if (this.fields [end.x, end.y].GetType ().Name != "Empty") {
          this.removedFigures.Add (this.fields [end.x, end.y]);
        }
        this.fields [end.x, end.y] = this.fields [start.x, start.y];
        this.fields [start.x, start.y] = new Empty ();

        //check if with this move the other player will be in check or even checkmate
        if (isCheck (player.next ())) {
          result.msg = "check";
          if (isCheckMate (player.next (), new coord (0, 0))) {
            result.msg = "checkmate";
          }
        }
        if (this.getFieldFigureName (end) == "Pawn") {
          ((Pawn)this.fields [end.x, end.y]).justMoved = true;
        }

        //remove all justMoved
        for (int x = 0; x < this.size.x; x++) {
          for (int y = 0; y < this.size.y; y++) {
            if (this.fields [x, y].color == player.ToString () && getFieldFigureName (new
coord (x, y)) == "Pawn") {
              ((Pawn)this.fields [x, y]).justMoved = false;
            }
          }
        }

        //set moved Pawn's justMoved to true;
        if (this.getFieldFigureName (end) == "Pawn" && (start.y + 2 == end.y || start.y - 2
== end.y)) {
          ((Pawn)this.fields [end.x, end.y]).justMoved = true;
        }
```

```
        if (this.getFieldFigureName (end) == "Pawn" && (end.y == 0 || end.y == 7)) {
          result.action = "chooser";
        }
        this.fields [end.x, end.y].hasMoved = true;
        isDraw (player.next (), new coord (0, 0));
        return result;
      }
      result.error = true;
      return result;
    }

    //returns true if the move is possible
    //retruns false if the move is not possible
    private bool checkMove (Player player, coord start, coord end)
    {
      if ((this.fields [start.x, start.y].GetType ().Name == "Empty") ||
        (this.fields [start.x, start.y].color != player.ToString ()) ||
        (this.fields [end.x, end.y].color == player.ToString ()))
        return false;

      return this.fields [start.x, start.y].move (this, start, end);
    }

    // Method that check if the player is under check
    private bool isCheck (Player player)
    {
      return isCheck (player, new coord (), new coord ());
    }
    //check if the king will be in check after this move and don't allow it if it is
    private bool isCheck (Player player, coord start, coord end)
    {
      coord king = new coord ();
      bool noKing = true;
      Figure removedObj = new Figure ();
      bool res = false;
      //move the figure to the new position if the new positon is not the same as the old
      if (!start.Equals (end)) {
        removedObj = this.fields [end.x, end.y];
        this.fields [end.x, end.y] = this.fields [start.x, start.y];
        this.fields [start.x, start.y] = new Empty ();
      }
      //search for the king on the board
      for (int x = 0; x < this.size.x; x++) {
        for (int y = 0; y < this.size.y; y++) {
          if (this.fields [x, y].color == player.ToString () && getFieldFigureName (new
coord (x, y)) == "King") {
            king = new coord (x, y);
            noKing = false;
          }
        }
      }
      //if a king was found, try to move all figures to the position of the king,if that is
possibile it means that the king is in check
      if (!noKing) {
        for (int x = 0; x < this.size.x && !res; x++) {
          for (int y = 0; y < this.size.y && !res; y++) {
            //if a move is posiblie means the king is checked
            if (checkMove (player.next (), new coord (x, y), king))
              res = true;
          }
        }
      }
      //revert the move so that the state of the board is the same as at the start of this
function
      if (!start.Equals (end)) {
```

```
        this.fields [start.x, start.y] = this.fields [end.x, end.y];
        this.fields [end.x, end.y] = removedObj;
      }
      return res;
    }

    // Method that check if the player is in checkmate
    private bool isCheckMate (Player player, coord start)
    {
      bool res = true;
      for (int x = 0; x < this.size.x && res; x++) {
        for (int y = 0; y < this.size.y && res; y++) {
          coord end = new coord (x, y);
          if (checkMove (player, start, end)) {
            res = isCheck (player, start, end);
            //if res == false means that there is a possible move
          }
        }
      }
      if (res) {
        if (start.x < this.size.x - 1) {
          res = isCheckMate (player, new coord (start.x + 1, start.y));
        } else if (start.y < this.size.y - 1) {
          res = isCheckMate (player, new coord (0, start.y + 1));
        }
      }
      return res;
    }

    // Method that move figures in search of possible move
    private bool isDraw (Player player, coord start)
    {
      coord end = new coord ();
      if (this.getFieldFigureColor (start) == player.ToString ()) {
        for (end.x = 0; end.x < this.size.x; end.x++) {
          for (end.y = 0; end.y < this.size.y; end.y++) {
            //move eache figure to each field
            if ((doEnPassant (player, start, end, true)) || (checkMove (player, start, end)
&& !isCheck (player, start, end))) {
              //Found possible move
              return false;
            }
          }
        }
      }
      if (start.x < this.size.x - 1) {
        return isDraw(player, new coord(start.x + 1, start.y));
      } else if (start.y < this.size.y - 1) {
        return isDraw(player, new coord(0, start.y + 1));
      }
      return true;
    }

    //special move: castling
    private bool doCastling (Player player, coord start, coord end)
    {
      //castling is only possible if the figure on the start position is a king witch hasn'
t moved
      if (this.getFieldFigureName (start) == "King" && this.fields [start.x, start.y].color
== player.ToString () && this.fields [start.x, start.y].hasMoved == false) {
        //left side castling
        if (start.x + 2 == end.x && this.getFieldFigureName (new coord (start.x + 1,
start.y)) == "Empty" && !Move (player, start, new coord (start.x + 1, start.y)).error) {
          //move the king a secound time
          if (!Move (player, new coord (start.x + 1, start.y), new coord (start.x + 2,
```

```
start.y)).error) {
            //save kings position to allow the rock to move
            Figure tmpKingPosition = this.fields [end.x, end.y];
            this.fields [end.x, end.y] = new Empty ();
            //move rock
            if (!Move (player, new coord (7, start.y), new coord (4, start.y)).error) {
              this.fields [end.x, end.y] = tmpKingPosition;
              return true;
            } else {
              tmpKingPosition.hasMoved = false;
              this.fields [start.x, start.y] = tmpKingPosition;
            }
          } else {
            this.fields [start.x, start.y] = this.fields [start.x + 1, start.y];
            this.fields [start.x, start.y].hasMoved = false;
            this.fields [start.x + 1, start.y] = new Empty ();
          }
        }
        //right side castling
        else if (start.x - 2 == end.x && this.getFieldFigureName (new coord (start.x - 1,
start.y)) == "Empty" && !Move (player, start, new coord (start.x - 1, start.y)).error) {
          //move the king a secound time
          if (!Move (player, new coord (start.x - 1, start.y), new coord (start.x - 2,
start.y)).error) {
            //save kings position to allow the rock to move
            Figure tmpKingPosition = this.fields [end.x, end.y];
            this.fields [end.x, end.y] = new Empty ();
            //move rock
            if (!Move (player, new coord (0, start.y), new coord (2, start.y)).error) {
              this.fields [end.x, end.y] = tmpKingPosition;
              return true;
            } else {
              tmpKingPosition.hasMoved = false;
              this.fields [start.x, start.y] = tmpKingPosition;
            }
          } else {
            this.fields [start.x, start.y] = this.fields [start.x - 1, start.y];
            this.fields [start.x, start.y].hasMoved = false;
            this.fields [start.x - 1, start.y] = new Empty ();
          }
        }
      }
      return false;
    }


    private bool doEnPassant (Player player, coord start, coord end)
    {
      return doEnPassant (player, start, end, false);
    }

    // Method for En Passant
    private bool doEnPassant (Player player, coord start, coord end, bool tryOnly)
    {
      int direction;
      if (player.ToString () == "white") {
        direction = 1;
      } else {
        direction = -1;
      }
      if (this.getFieldFigureName (start) == "Pawn" &&
        this.fields [start.x, start.y].color == player.ToString () &&
        (start.x + 1 == end.x || start.x - 1 == end.x) &&
        start.y + direction == end.y &&
        this.getFieldFigureName (end.x, start.y) == "Pawn" &&
```

```
        ((Pawn)this.fields [end.x, start.y]).justMoved &&
        (this.getFieldFigureName (end.x, end.y) == "Empty")) {
        if (!tryOnly) {
          //do actual move
          this.removedFigures.Add (this.fields [end.x, end.y - direction]);
          this.fields [end.x, end.y - direction] = new Empty ();
          this.fields [end.x, end.y] = this.fields [start.x, start.y];
          this.fields [start.x, start.y] = new Empty ();
        }
        return true;
      }
      return false;
    }

    //Method for getting the name of the figure,have in input a coord object
    public string getFieldFigureName (coord c)
    {
      return getField (c).name ();
    }

    // Method for getting name of figure,having in input two int
    public string getFieldFigureName (int x, int y)
    {
      return getFieldFigureName (new coord (x, y));
    }

    // Method that get the color of the figure
    public string getFieldFigureColor (coord c)
    {
      return getField (c).color;
    }

    // Method that get the fields x and y from a coord object
    public Figure getField (coord c)
    {
      return this.fields [c.x, c.y];
    }
  }
}
```

Classe Game:

```
using System;
using System.Collections.Generic;

namespace Chess
{
  public class Game
  {
    private Board board;    // Istance of a board object
    private Player player;  // Istance of a player object

    // Constructor
    public Game ()
    {
      this.board = new Board ();
      this.player = new Player ("white");
    }

    //interaction function for the user interface to the game
    public Message call (Payload data) {
      Message result = new Message (false, "", "", null);
      if (!data.error) {
        switch (data.action) {
        case "switchFigures":
          result = this.board.switchFigures (data.figure, data.startPos);
          break;
        case "move":
          //need to block the game when waiting for click on the chooser
          result = this.board.Move (this.player, data.startPos, data.endPos);
          if (!result.error) {
            this.player = this.player.next ();
          }
          break;
        case "checkSelection":
          result = this.board.Move (this.player, data.startPos);
          break;
        }
      }
      return result;
    }

    // Get method that returns all chooseable figures
    public Figure[] getChooseableFigures() {
      return this.board.chooseableFigures;
    }

    // Get method that return size
    public coord getSize() {
      return this.board.size;
    }

    // List of removed figures
    public List<Figure> getRemovedFigures ()
    {
      return this.board.removedFigures;
    }

    // Method for getting the name of the figure
    public string getFieldFigureName (coord coord)
    {
      return this.board.getFieldFigureName (coord);
    }
```

```
    // Method for getting the color of the figure
    public string getFieldFigureColor (coord coord)
    {
      return this.board.getFieldFigureColor (coord);
    }

    // Message for the initial state
    public Message initialState() {
      return new Message (false, "", "", player.next());
    }
  }
}
```

Classe Player:

```
using System;

namespace Chess
{
  /* Class for the player */
  public class Player
  {
    private string currentPlayer;         /* String identifiyng color of actual player */

    /* Constructor of the player object */
    public Player (string player)
    {
      this.currentPlayer = player;
    }

    /* Override of 'ToString' method */
    public override string ToString ()
    {
      return currentPlayer;                /* It returns the string that represent color of
actual player */
    }

    /* Method for the alternance of players */
    public Player next ()
    {
      /* If current player is white next will be black and vice-versa */
      return new Player ((this.currentPlayer == "white") ? "black" : "white");
    }
  }
}
```

Classe Message:

```
using System;

namespace Chess
{
  public class Message
  {
    public bool error { get; set;}      // If it is true an error has occurred
    public string msg { get; set;}      // String for the message
    public Player player { get; set;}   // Player object
    public string action { get; set;}   // Action to do

    // Constructor
    public Message (bool error, string msg, string action, Player player)
    {
      this.error = error;
      this.msg = msg;
      this.player = player;
      this.action = action;
    }

    // Virtual method that print true  on the terminal if there is an error and eventually
print a message
    public virtual void print ()
    {
      Console.WriteLine ("Error: " + this.error.ToString().ToLower() + ", Message: " +
this.msg);
    }

    // method for printing message to user
    public virtual string format () {
      string returnMsg = "It's " + this.player.next().ToString() + "'s turn.";

      switch (this.msg) {
      case "check":
        returnMsg += " | " + this.player.next().ToString() + " is in check";
        break;
      case "checkmate":
        returnMsg = this.player.next().ToString() + " is checkmate | " +
this.player.ToString() + " wins";
        break;
      }
      return returnMsg;
    }
  }
}
```

Classe Payload:

```
using System;

namespace Chess
{
  // Class for transport of information
  public class Payload
  {
    public bool error { get; set; }      // If it is true there is an error
    public string action { get; set; }   // Action to do
    public coord startPos { get; set; }  // Coord of starting position
    public coord endPos { get; set; }    // Coord of ending position
    public Figure figure { get; set; }   // Istance of a figure

    //payload for chooser response
    public Payload (bool error, string action, Figure f, coord pos)
    {
      this.startPos = pos;
      this.action = action;
      this.error = error;
      this.figure = f;
    }
    //payload for move command
    public Payload (bool error, string action, coord start, coord end)
    {
      this.startPos = start;
      this.endPos = end;
      this.action = action;
      this.error = error;
    }

    //payload for check selection
    public Payload (bool error, string action, coord start)
    {
      this.startPos = start;
      this.action = action;
      this.error = error;
    }
  }
}
```

Classe Coord:

```
using System;

namespace Chess
{
  // Definition of the struct used for coordinates
  public struct coord
  {
    public int x;
    public int y;

   // Constructor of the class coord
    public coord (int x, int y) {
      this.x = x;
      this.y = y;
    }
  }
}
```

Classi per le pedine:

Classe figure:

```
using System;

namespace Chess
{
  /* Generic class for a figure */
  public class Figure
  {
    public string color { get; set; }   /* String for the color offigures */
    public bool hasMoved { get; set; }  /* Flag for checking if the figure has moved */

    public Figure (string color)
    {
      this.color = color;
      this.hasMoved = false;
    }

    public Figure ()
    {
      this.color = "";
      this.hasMoved = false;
    }

    /* Method for the movement of the figure on the board */
    public virtual bool move (Board board, coord start, coord end)
    {
      return true;
    }

    public string name () {
      return this.GetType().Name;
    }
  }
}
```

Classe Empty:

```
using System;

namespace Chess
{
  /* Class for an empty object on the board */
  public class Empty : Figure
       {
    /* Constructor of an empty object*/
    public Empty ()
    {
    }
       }
}
```

Classe Bishop:

```
using System;

namespace Chess
{
  /* Class for the bishop figure */
  class Bishop : Figure
  {
    /* Constructor of the figure bishop */
    public Bishop (string color) : base (color)
    {
    }
    /* Method for the movement of the bishop*/
    public override bool move (Board board, coord start, coord end)
    {
      if (Math.Abs (start.x - end.x) == Math.Abs (start.y - end.y)) {
          coord tmp = start;
          while (!tmp.Equals (end)) {         // Until tmp isn't equal to end coordinates:
          if (board.getFieldFigureName (tmp) != "Empty" && !tmp.Equals(start))
// If there is an object along trajectory

            return false;                              // movement isn't permitted
           if (tmp.x > end.x && tmp.y > end.y) {
// If temp.x and temp.y are major than their end

            tmp.x--;                                   //Decrement both
            tmp.y--;
           }
           if (tmp.x > end.x && tmp.y < end.y) {     // or temp.x is major and temp.x is
minor
            tmp.x--;                                   // decrement x
            tmp.y++;                                   // increment y
           }
           if (tmp.x < end.x && tmp.y > end.y) {// or temp.x is minor and temp.y is major
            tmp.x++;                                   // increment x
            tmp.y--;                                   // decrement y
           }
           if (tmp.x < end.x && tmp.y < end.y) {  // or temp.x is minor and temp.y is minor
            tmp.x++;                                   // Increment both
            tmp.y++;
           }
         }
          return true;    // movement is allowed if there isn't an object along trajectory
      }
      return false;         // If the movement isn't diagonal it isn't permitted
    }
  }
}
```

Classe King:

```
using System;

namespace Chess
{
  /* Class for the king figure*/
  public class King : Figure
  {

    /* Constructor of the figure king */
    public King (string color) : base (color)
    {
    }

    /* Method for the movement of king */
    public override bool move (Board board, coord start, coord end)
    {
      for (int x = (start.x == 0) ? 0 : start.x - 1; x <= start.x + 1; x++) {
/* If the movement on x is between -1 and 1(included) */

        for (int y = (start.y == 0) ? 0 : start.y - 1; y <= start.y + 1; y++) {
 /* if the movement on y is between -1 and 1(included) */

          if (end.Equals (new coord (x, y)))    /* and the end coordinates are updated */
            return true;                        /* this method return a true value */
        }
      }
      return false;
/* If the value of x and y are not corresponding to the rule the movement isn't allowed  */
    }
  }
}
```

Classe Knight:

```
using System;

namespace Chess
{
  /* Class for the knight figure */
  public class Knight : Figure
  {

    /* Constructor of the object knight */
    public Knight (string color) : base (color)
    {
    }

    /* Method for the movement of knight */
    public override bool move (Board board, coord start, coord end)
    {
      coord rule = new coord (1, 2);
      int x = rule.x;                     // x is setted to 1
      int y = rule.y;                     // y is setted to 2

      for (int i = 0; i < 2; i++) {
        if (i == 1) {                     // When i is equal to 1
          x = rule.y;                     // x is setted to 2
          y = rule.x;                     // and y is setted to 1
        }
        if (start.x + x == end.x) {
  // If the movement on x is equal to starting position plus x
          if (start.y + y == end.y) {
  // and the movement on y is equal to starting position plus y
            return true;                  // movement is permitted
          }
        }
        if (start.x - x == end.x) {
// If the movement on x is equal to starting position minus x

          if (start.y - y == end.y) {
// and the movement on y is equal to starting position minus y

            return true;                  // movement is permitted
          }
        }
        if (start.x - x == end.x) {
// If the movement on x is equal to starting position minus x

          if (start.y + y == end.y) {
 // and the movement on y is equal to starting position plus y

            return true;                  // movement is permitted
          }
        }
        if (start.x + x == end.x) {
// If the movement on x is equal to starting position plus x

          if (start.y - y == end.y) {
// and the movement on y is equal to starting position minus y

            return true;                  // movement is permitted
          }
        }
      }
      return false;                       // Other movement aren't permitted
    }
  }
}
```

Classe Pawn:

```
using System;

namespace Chess
{

  /* Class for the pawn figure */
  public class Pawn : Figure
  {
    public bool justMoved { get; set; }

    /* Constructor of the object pawn*/
    public Pawn (string color) : base (color)
    {
      justMoved = false;
/* Definition of the boolean used for the checking if the pawn has moved */
    }

    /* Method for the movement of pawn */
    public override bool move (Board board, coord start, coord end)
    {
      int direction;        /* Int value for direction of movement */

  /* If the color is white we have an increment of y(positive direction) else a decrement */
      if (this.color == "white") {
        direction = 1;
      } else {
        direction = -1;
      }
      if (!this.hasMoved &&
          start.x == end.x &&
          start.y + (2 * direction) == end.y &&
          board.getFieldFigureName (start.x, start.y + direction) == "Empty" &&
          board.getFieldFigureName (start.x, start.y + (2 * direction)) == "Empty")
        return true;
      if (start.x == end.x &&
          start.y + direction == end.y &&
          board.getFieldFigureName (start.x, start.y + direction) == "Empty")
        return true;
      if ((start.x + 1 == end.x || start.x - 1 == end.x) &&
          start.y + direction == end.y &&
          board.getFieldFigureName (end) != "Empty") {
        return true;
      }

      return false;
/* If the value of x and y are not corresponding to the rule the movement isn't allowed  */
    }
  }
}
```

Classe Queen:

```
using System;

namespace Chess
{

  /* Class for the queen figure */
  public class Queen : Figure
  {

    /* Constructor of the object queen */
    public Queen (string color) : base (color)
    {
    }

    /* Method for the movement of the queen */
    public override bool move (Board board, coord start, coord end)
    {
      // Diagonal movement
      if (Math.Abs (start.x - end.x) == Math.Abs (start.y - end.y)) {
        coord tmp = start;
        while (!tmp.Equals (end)) {             // Until tmp isn't equal to end coordinates:
          if (board.getFieldFigureName (tmp) != "Empty" && !tmp.Equals(start))
// If there is an object along trajectory

            return false;                        // movement isn't permitted
          if (tmp.x > end.x && tmp.y > end.y) {
// If temp.x and temp.y are major than their end

            tmp.x--;                             //Decrement both
            tmp.y--;
          }
          if (tmp.x > end.x && tmp.y < end.y) {  // or temp.x is major and temp.x is minor
            tmp.x--;                             // decrement x
            tmp.y++;                             // increment y
          }
          if (tmp.x < end.x && tmp.y > end.y) {  // or temp.x is minor and temp.y is major
            tmp.x++;                             // increment x
            tmp.y--;                             // decrement y
          }
          if (tmp.x < end.x && tmp.y < end.y) {  // or temp.x is minor and temp.y is minor
            tmp.x++;                             // Increment both
            tmp.y++;
          }
        }
        return true;       // movement is allowed if there isn't an object along trajectory
      }

      // Vertical and horizontal movement
    if (start.x == end.x) {                    // Check for the vertical movement
      if (start.y < end.y) {
 // If the end position is major than the starting position

        for (int i = start.y + 1; i < end.y; i++) {// Check for an object in this trajectory
          if (board.getFieldFigureName (start.x, i) != "Empty") {
            return false;        // If there is one (not empty) movement isn't allowed
          }
        }
      } else {                  // If the end position is minor than the starting position
        for (int i = start.y - 1; i > end.y; i--) {// Check for an object in this trajectory
            if (board.getFieldFigureName (start.x, i) != "Empty") {
              return false;     // If there is one (not empty) movement isn't allowed
            }
```

```
        }
      }
      return true;                    // If there aren't object movement is permitted

    } else if (start.y == end.y) {     // Check for the horizontal movement
      if (start.x < end.x) {  // If the end position is major than the starting position
        for (int i = start.x + 1; i < end.x; i++) {// Check for an object in this trajectory
            if (board.getFieldFigureName (i, start.y) != "Empty") {
              return false;           // If there is one (not empty) movement isn't allowed
          }
        }
      } else {                  // If the end position is minor than the starting position
        for (int i = start.x - 1; i > end.x; i--) {// Check for an object in this trajectory
            if (board.getFieldFigureName (i, start.y) != "Empty") {
              return false;    // If there is one (not empty) movement isn't allowed
            }
          }
        }
        return true;        // If there aren't object along trajectory movement is permitted
      }
      return false;
// Any other case different from movements horizontal,diagonal or vertical aren't permitted
    }
  }
}
```

Classe Rock:

```
using System;

namespace Chess
{
  // Class for the rock figure public class Rock : Figure
{

    // Constructor of the rock figure
  public Rock (string color) : base (color)
  {
  }

    // Method for the movement of rock
  public override bool move (Board board, coord start, coord end
  {
      //  int x = this.rule.x;
      // int y = this.rule.y;

    if (start.x == end.x) {    // Check for the vertical movement
      if (start.y < end.y) {   // If the end position is major than the starting position
        for (int i = start.y + 1; i < end.y; i++) {// Check for an object in this trajectory
            if (board.getFieldFigureName (start.x, i) != "Empty") {
              return false;    // If there is one (not empty) movement isn't allowed
          }
        }
      } else {                // If the end position is minor than the starting position
        for (int i = start.y - 1; i > end.y; i--) {// Check for an object in this trajectory
            if (board.getFieldFigureName (start.x, i) != "Empty") {
              return false;   // If there is one (not empty) movement isn't allowed
            }
          }
        }
        return true;         // If there aren't object movement is permitted
```

```
    } else if (start.y == end.y) { // Check for the horizontal movement
      if (start.x < end.x) {    // If the end position is major than the starting position
        for (int i = start.x + 1; i < end.x; i++) {// Check for an object in this trajectory
            if (board.getFieldFigureName (i, start.y) != "Empty") {
              return false;     // If there is one (not empty) movement isn't allowed
            }
          }
      } else {               // If the end position is minor than the starting position
        for (int i = start.x - 1; i > end.x; i--) {// Check for an object in this trajectory
            if (board.getFieldFigureName (i, start.y) != "Empty") {
              return false;  // If there is one (not empty) movement isn't allowed
            }
          }
        }
        return true;        // If there aren't object along trajectory movement is permitted
      }
      return false;       // If the movement isn't vertical or horizontal it isn't permitted
    }
  }
}
```

Classi per l'interfaccia grafica:

Classe Graphics:

```
using System;
using Gtk;

namespace Chess
{
  public delegate bool Cb (coord start, coord end);
  public class Graphics : Gtk.Window
  {
    private GridWidget mainGrid;
    private SidebarWidget sidebarRight;
    private SidebarWidget sidebarLeft;
    private Popup chooser;
    private Label status;
    private Game game;

    // constructor that show the graphic interface to users
    public Graphics (Game g) : base (Gtk.WindowType.Toplevel)
    {
      Build ();
      int scale = 10;
      if (Screen.Height < 1000)
        scale = 5;
      this.game = g;
      VBox gridWrapper = new VBox ();
      HBox box = new HBox ();
      this.status = new Label ("");
      this.chooser = new Popup (this, this.game.getChooseableFigures(), handleChooser,
scale);
      this.mainGrid = new GridWidget (this.game, clickHandler, scale);
      gridWrapper.PackStart (status, false, false, 0);
      gridWrapper.PackStart (this.mainGrid, false, false, 0);
      this.sidebarLeft = new SidebarWidget (g.getRemovedFigures (), "black", scale);
      box.PackStart (new HBox ());
      box.PackStart (this.sidebarLeft);
      box.PackStart (gridWrapper, false, false, 0);
      this.sidebarRight = new SidebarWidget (g.getRemovedFigures (), "white", scale);
```

```
      box.PackEnd (this.sidebarRight);
      box.PackStart (new HBox ());
      box.ShowAll ();
      this.Add (box);
      updateGui (this.game.initialState ());
      this.Show ();
    }

    // method for updating of graphic interface
    public void updateGui (Message msg)
    {
      this.sidebarLeft.updateSidebar ();
      this.sidebarRight.updateSidebar ();
      this.mainGrid.updateGrid ();
      if (msg.error == false && msg.format () != "")
        this.status.Text = msg.format ();
      msg.print ();

    }

    // Method for delete an event
    protected void OnDeleteEvent (object sender, DeleteEventArgs a)
    {
      Application.Quit ();
      a.RetVal = true;
    }

    // Method that build the graphics of the game
    protected virtual void Build ()
    {
      global::Stetic.Gui.Initialize (this);
      // Widget Chess.MainWindow
      this.Name = "Chess.MainWindow";
      this.Title = "Chess";
      this.WindowPosition = ((global::Gtk.WindowPosition)(3));
      this.DeleteEvent += new global::Gtk.DeleteEventHandler (this.OnDeleteEvent);
    }

    // Method for handling the click
    public bool clickHandler (coord start, coord end)
    {
      Message msg;
      if (start.Equals (end)) {
        msg = this.game.call(new Payload (false, "checkSelection", start));
      } else {
        msg = this.game.call(new Payload(false, "move", start, end));
        if (msg.action == "chooser") {
          this.chooser.open (msg.player, end);
        }
        updateGui (msg);
      }
      return !msg.error;
    }

    // Method for handling the chooser
    public void handleChooser (Figure figure, coord position)
    {
      updateGui (this.game.call (new Payload (false, "switchFigures", figure, position)));
    }

  }
}
```

Classe GridWidget:

```
using System;
using Gtk;

namespace Chess
{
  public partial class GridWidget : Table
  {
    //public delegate bool Cb(coord start, coord end);
    private Game game;                // Istance of game class
    private Cb callback;              // Delegate method
    private coord clickedPosition;    // coord of clicked position
    private bool clicked;             // true if a tile is clicked
    private coord tileSize;           // Size of a tile

    // Constructor
    public GridWidget (Game game, Cb callback, int scale) : base ((uint)game.getSize ().x,
(uint)game.getSize ().y, true)
    {
      this.callback = callback;
      this.clicked = false;
      this.game = game;
      this.tileSize = new coord (10 * scale, 10 * scale);
    }

    // Method for adding a figure to game
    private void addFigure (Widget w, coord pos)
    {
      this.Attach (w, (uint)pos.x, (uint)pos.x + 1, (uint)pos.y, (uint)pos.y + 1);
    }

    // Method that draw the board
    private void createBoard ()
    {
      //alternating background color for the grid
      string tileBackground = "white";
      for (int x = 0; x < this.game.getSize ().x; x++) {
        tileBackground = (tileBackground == "white") ? "gray" : "white";
        for (int y = 0; y < this.game.getSize ().y; y++) {
          string type = this.game.getFieldFigureName (new coord (x, y));
          string playerColor = this.game.getFieldFigureColor (new coord (x, y));
          tileBackground = (tileBackground == "white") ? "gray" : "white";

          type = (type == "Empty") ? "" : type;
          TileWidget tile = new TileWidget (tileBackground, type, playerColor,
this.tileSize);
          tile.position = new coord (x, y);
          tile.ButtonPressEvent += onTileClicked;
          this.addFigure (tile, new coord (x, y));
        }
      }
      this.ShowAll ();
    }

    // Method for updating the grid
    public void updateGrid ()
    {
      //remove all children
      foreach (Widget child in this.Children) {
        child.Destroy ();
      }
      //add all children
      createBoard ();
```

```
    }

    // Method that describe what happen when a tile is clicked
    private void onTileClicked (object obj, ButtonPressEventArgs args)
    {
      if (((Gdk.EventButton)args.Event).Type == Gdk.EventType.ButtonPress) {
        TileWidget tile = (TileWidget)obj;
        //Console.WriteLine (tile.position.x + ", " + tile.position.y + " " + tile.color + "
" + tile.figure);
        if (!this.clicked) {
          this.clickedPosition = new coord (tile.position.x, tile.position.y);

          if (this.callback (this.clickedPosition, this.clickedPosition)) {
            this.clicked = true;
            Fixed f = (Fixed)(tile.Child);
            Image circle = tile.loadCircle (this.tileSize);
            f.Add (circle);
            f.ShowAll ();
          }
        } else {
          //remove the ring whenn the user clicks agen on the same tile
          if (this.callback (this.clickedPosition, tile.position) &&
              this.clickedPosition.Equals (tile.position)) {
            Fixed f = (Fixed)(tile.Child);
            if (f.Children.Length > 1)
              f.Remove (f.Children [1]);
          }
          this.clicked = false;
        }
      }
    }
  }
}
```

ClassePopup:

```
using System;
using Gtk;
```

```
namespace Chess
{
  // class for the popup when the pawn have to be promoted
  public class Popup: Gtk.Window
  {
    private HBox box;
    private Figure[] figures;
    private Action<Figure, coord> callback;
    private coord tileSize;

    // constructor
    public Popup (Window parent, Figure[] f, Action<Figure, coord> callback, int scale) :
base (Gtk.WindowType.Toplevel)
    {
      this.TransientFor = parent;
      this.SetPosition (Gtk.WindowPosition.CenterOnParent);
      this.Decorated = false;

      this.tileSize = new coord (10 * scale, 10 * scale);
      this.callback = callback;
      this.Title = "choose";
      this.figures = f;
      this.box = new HBox ();
      this.Add (this.box);
    }

    // Method for the opening of popup
    public void open (Player player, coord position)
    {
      close ();
      foreach (Figure fig in this.figures) {
        TileWidget tile = new TileWidget ("", fig.name (), player.ToString (),
this.tileSize);
        tile.position = position;
        fig.color = player.ToString ();
        box.PackStart (tile);
        tile.ButtonPressEvent += onTileClicked;
      }
      this.ShowAll ();
    }

    // Method for the closing of the popup
    private void close ()
    {
      foreach (Widget child in this.box.Children) {
        child.Destroy ();
      }
      this.Hide ();
    }

    // Method that describe the reaction of a tile when it is clicked
    private void onTileClicked (object obj, ButtonPressEventArgs args)
    {
      TileWidget tile = (TileWidget)obj;
      bool found = false;
      for (int i = 0; i < ((HBox)tile.Parent).Children.Length && !found; i++) {
        if (((HBox)tile.Parent).Children [i].Equals (tile)) {
          found = true;
          this.callback (this.figures [i], tile.position);
        }
      }
      close ();
    }
  }
}
```

Classe SidebarWidget:

```
using System;
using Gtk;
using System.Collections.Generic;

namespace Chess
{
  public partial class SidebarWidget : Gtk.VBox
  {
    private List<Figure> removedFigures;   // List of removed figures
    private string color;                  // Color of the sidebar's admitted figures
    private coord tileSize;                // Size of tile

    // Constructor
    public SidebarWidget (List<Figure> removedFigures, string color, int scale)
    {
      this.removedFigures = removedFigures;
      this.color = color;                       // Color of the sidebar's admitted figures

      updateSidebar ();

      this.WidthRequest = 5 * scale;
      this.tileSize = new coord (5 * scale, 5 * scale);
    }

    // Method for updating the sidebar
    public void updateSidebar ()
    {
      foreach (Widget child in this.Children) {
        child.Destroy ();
      }
      foreach (Figure f in removedFigures) {
        if (f.color == this.color) {
          this.PackStart (new TileWidget ("", f.GetType ().Name, f.color, this.tileSize),
false, false, 0);
        }
      }
      this.ShowAll ();
    }
  }
}
```

Classe TileWidget:

```
using System;
using System.IO;
using System.Reflection;
using System.Runtime.InteropServices;
using Gtk;

namespace Chess
{
  public partial class TileWidget : Gtk.EventBox
  {
    public coord position { get; set; }   // Position

    public string color { get; set; }     // Color of a tile

    public string figure { get; set; }    // Instance of a figure

    // constructor
    public TileWidget (string colorBg, string figure, string color, coord size)
    {
      string imgName;
      if (color != "" && figure != "" && figure.ToLower () != "empty") {
        if (figure.ToLower () != "knight") {
          imgName = color.ToLower () [0].ToString () + figure.ToUpper () [0].ToString ();
        } else {
          imgName = color.ToLower () [0].ToString () + "N";
        }
        Image img = loadSvg (imgName, size);
        Fixed f = new Fixed ();
        f.Add (img);
        f.ShowAll ();
        this.Add (f);
        this.Show ();
      }
      if (colorBg != "") {
        Gdk.Color col = new Gdk.Color ();
        Gdk.Color.Parse (colorBg, ref col);
        this.ModifyBg (StateType.Normal, col);
      }
      this.figure = figure;
      this.color = color;
    }

    // Method that load images of figures
    public Gtk.Image loadSvg (string file, coord size)
    {
      Gdk.Pixbuf display;
      string basePath = System.IO.Path.GetDirectoryName
(System.Reflection.Assembly.GetExecutingAssembly ().GetName ().CodeBase).Substring (5);
      try {
        display = new Gdk.Pixbuf (basePath + @"/img/cburnett/" + file + ".svg", size.x,
size.y);
      } catch (GLib.GException e) {
        Console.WriteLine (e);
        display = null;
      }
      return (new Gtk.Image (display));

    }

    // Method for loading of the circle that inscribe the figure when it is clicked
    public Gtk.Image loadCircle (coord size)
```

```
    {
      string basePath = System.IO.Path.GetDirectoryName
(System.Reflection.Assembly.GetExecutingAssembly ().GetName ().CodeBase).Substring (5);
      Gdk.Pixbuf display = new Gdk.Pixbuf (basePath + @"/img/circle.svg", size.x, size.y);
      return (new Gtk.Image (display));
    }
  }
}
```

Infine ecco il codice della classe program:

```
using System;
using Gtk;

namespace Chess
{
  class MainClass
  {
    public static void Main (string[] args)
    {
      Application.Init ();
      //Create game
      Game g = new Game ();
       //Create the graphic interface for the game
      new Graphics (g);
      Application.Run ();
    }
  }
}
```

Test del programma

Test del programma:

È stato scelto di effettuare sul programma dei test di tipo black box,ossia dei test sul funzionamento del programma senza considerare il codice(come se fosse,per l'appunto,chiuso in una scatola nera). La maggior parte dei test sono stati effettuati in base ai casi d'uso descritti nel relativo paragrafo.

I seguenti test sono stati effettuati su Acer vn7-792g con ubuntu 15.04 emulato tramite VMware Workstation 12 Player usando monodevelop con le seguenti specifiche:
-***Ram*** *3256 MB*
-***Cores*** *4*
-***Risoluzione*** *1920x1080*

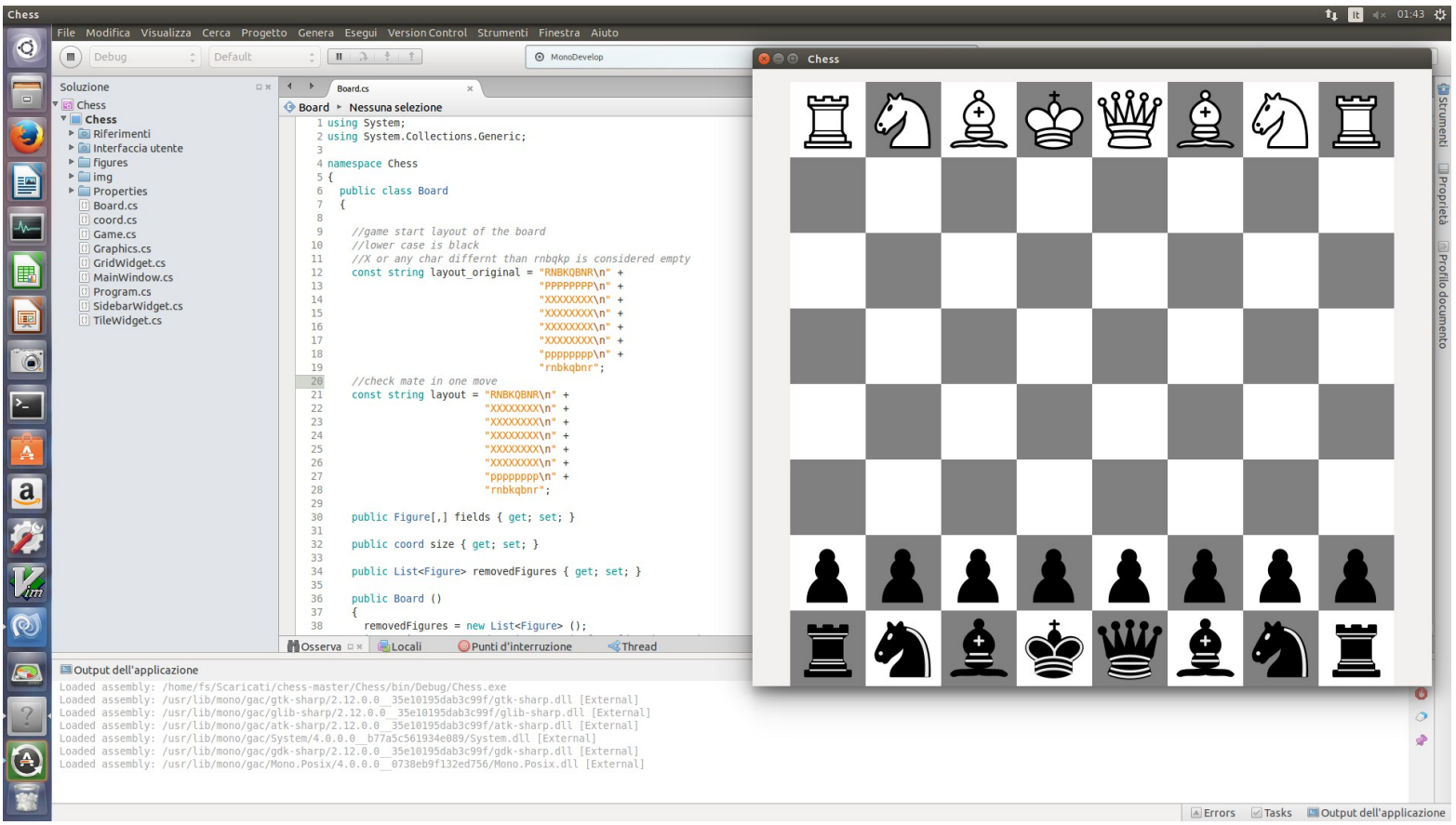
Inoltre sono stati effettuati vari test su Arch Linux(release del 01-05-2016, Kernel 4.5.1) tramite Xamarin studio con le seguenti specifiche:
-***Ram*** *16 GB*
-***Cores*** *8(4 fisici con Hypertreading)*
-***Risoluzione*** *1920x1080*

Su un Lenovo thinkpad T400 con risoluzione 1280x800 e sistema operativo Parabola GNU/Linux 2015.05.01 tramite monodevelop con:

-***Ram*** *2gb*

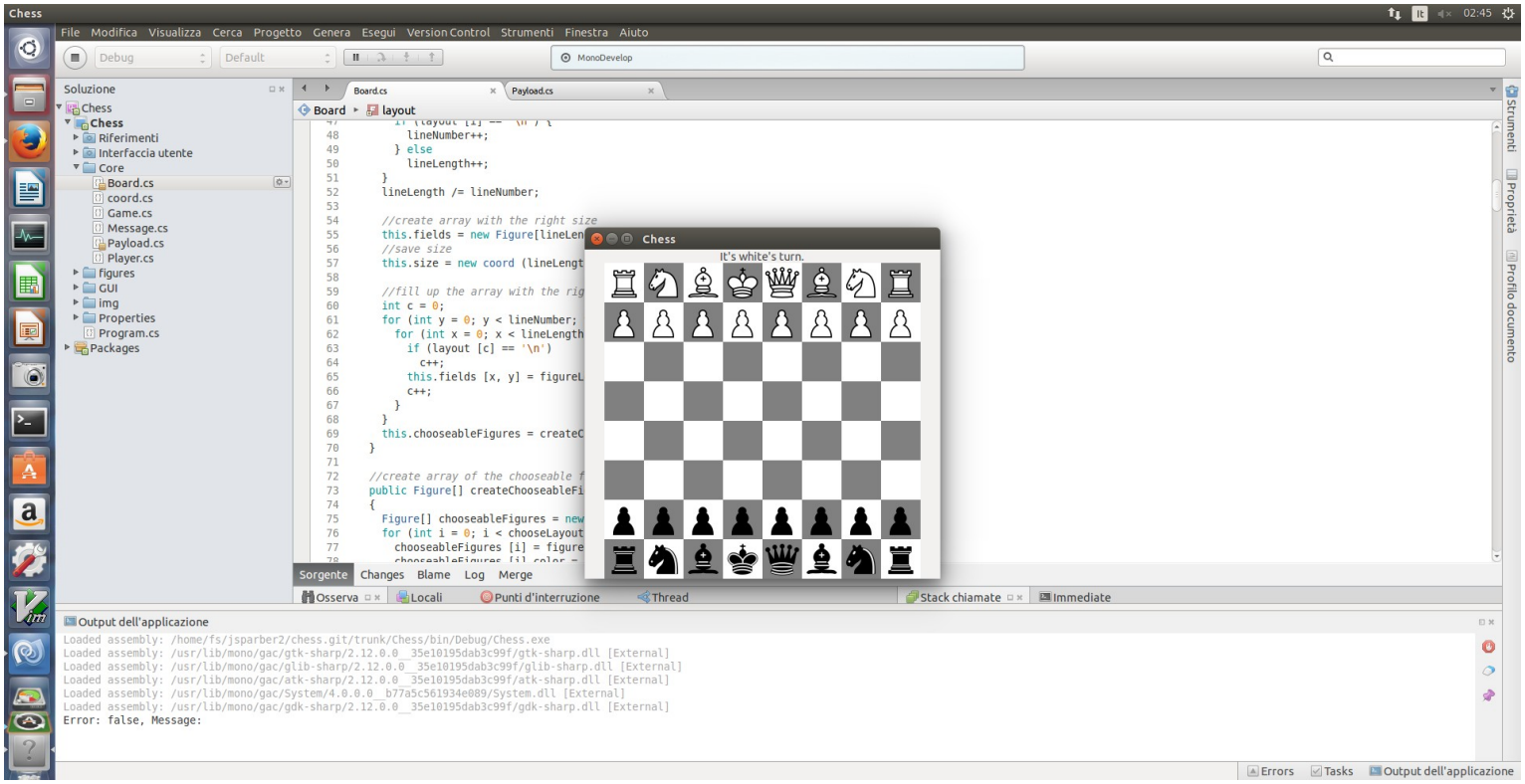
E su un macbook air con Arch Linux tramite monodevelop e risoluzione 1440x900

Il primissimo test è stato il test del caricamento corretto delle librerie e dell'avvio del programma



Per questo test,per semplicità di alcuni test di movimento di pedine successivi,è stato modificata la disposizione delle pedine rispetto quella originale andando ad eliminare i pedoni bianchi.

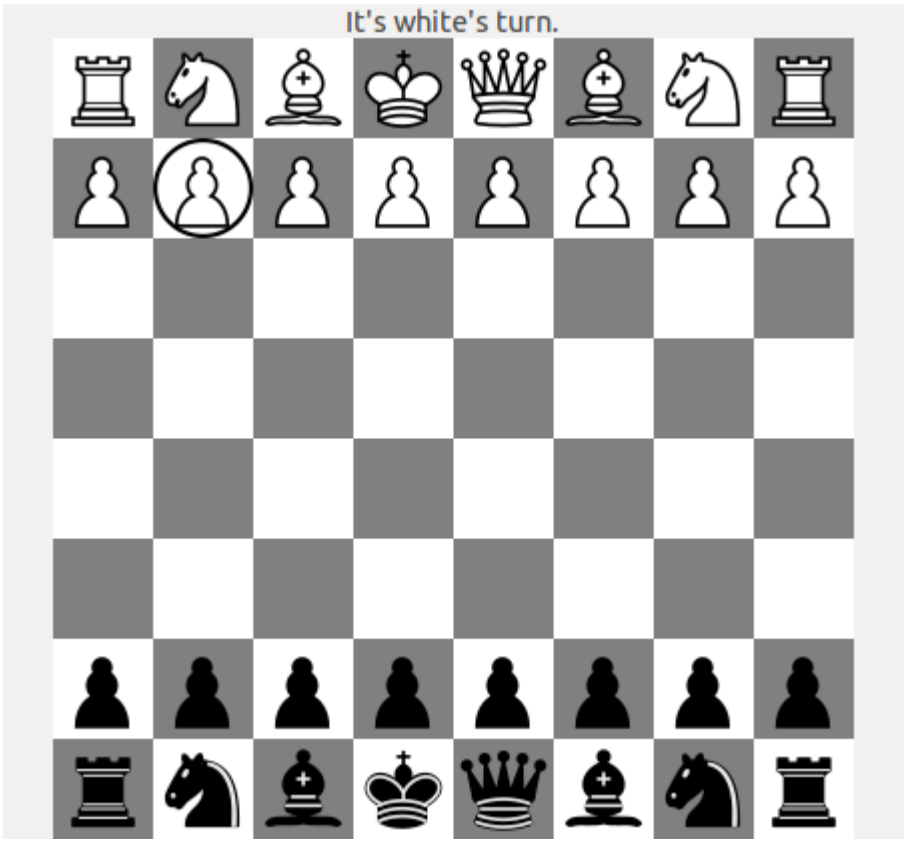
Quello precedente è stato uno dei primi test,mancava ancora il riquadro apposito per i messaggi all'utente. Di seguito sono stati effettuati altri test d'avvio,tra cui viene riportato il più recente:



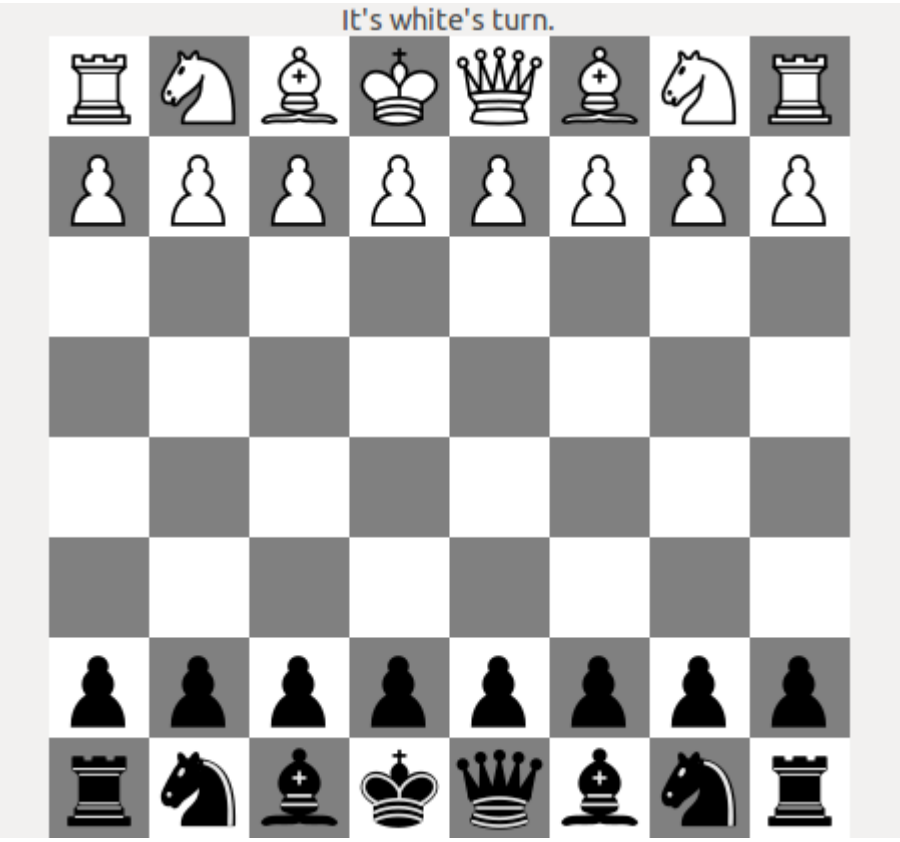
Si può notare,oltre al ridimensionamento su schermo(effettuato in seguito ad un bug in cui l'immagine non si adattava su uno schermo di risoluzione 1366 x 768 e non era ridimensionabile) il caricamento dello spazio riservato ai messaggi situato sopra la scacchiera e il messaggio su terminale composto in primo luogo da un valore error:se fosse stato uguale a true esso avrebbe indicato la presenza di un errore.A destra di quest'ultimo viene stampato un eventuale messaggio di scacco o scacco matto.

Tra parentesi tonde accanto al nome del test verrà riportato il numero del caso d'uso di riferimento

Test di selezione di una pedina(0)



Test di deselezione pedina(1)



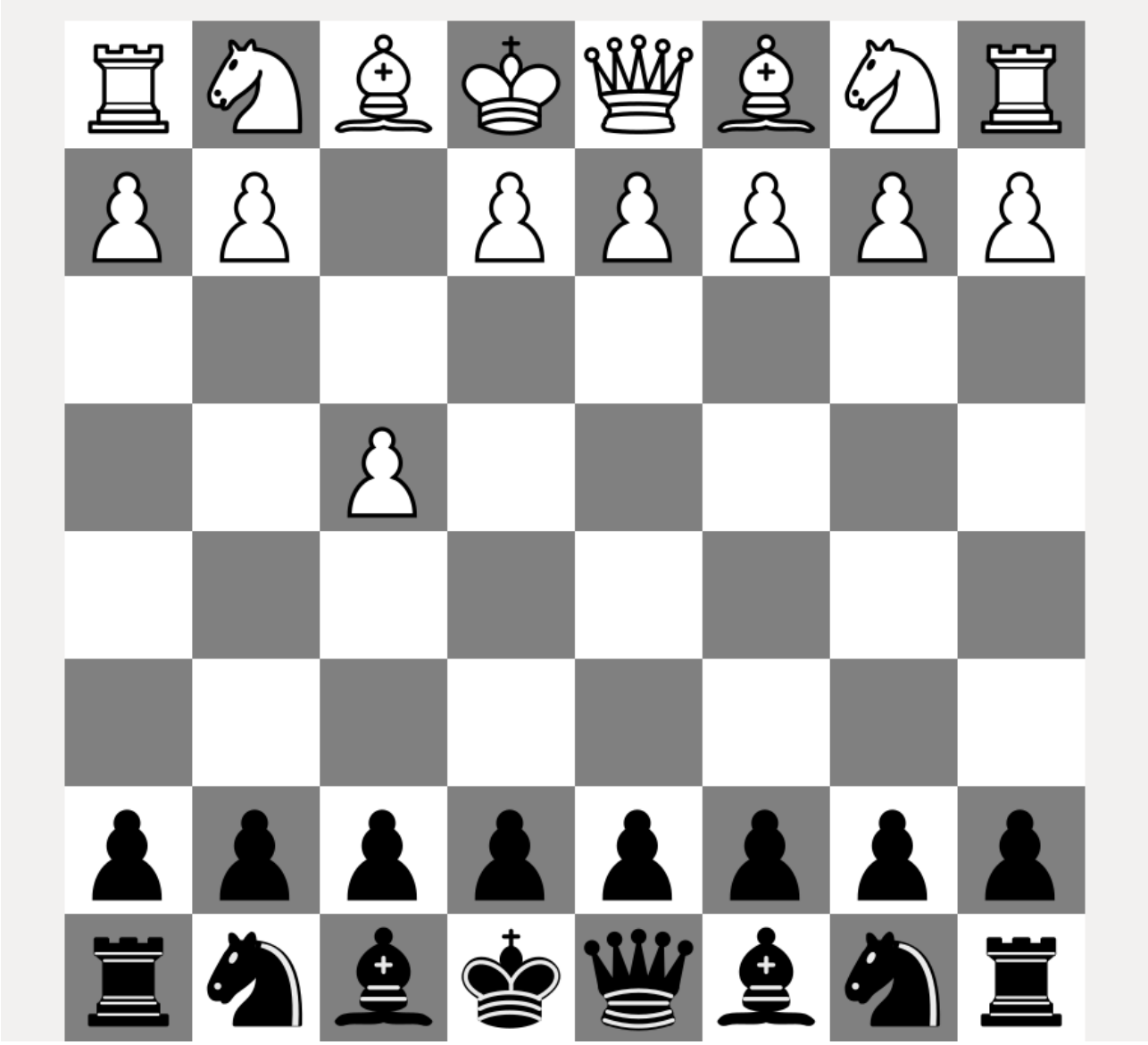
Test di movimento della pedina selezionata e di movimento del primo turno di un pedone(2-3)

Si noti che viene mostrato correttamente il messaggio di inizio turno del giocatore nero.

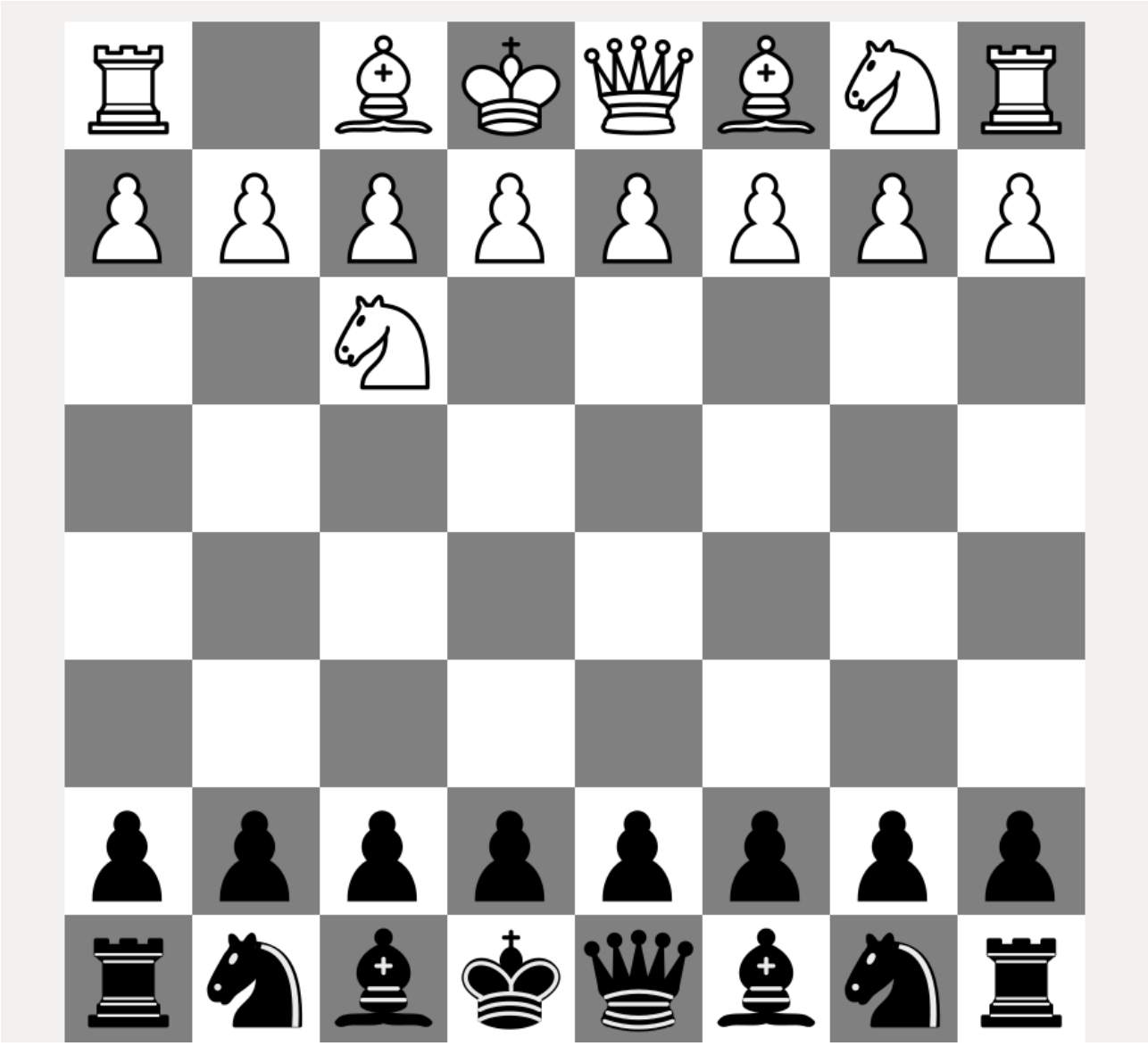
Test di movimento non consentito(2bis):
Il test è stato effettuato tentando di muovere il precedente pedone in una casella diagonale anziché in quella frontale.Il risultato è stato l'output su terminale della variabile error impostata a vero

```
Loaded assembly: /home/fs/jsparb
Loaded assembly: /usr/lib/mono/g
Loaded assembly: /usr/lib/mono/g
Loaded assembly: /usr/lib/mono/g
Loaded assembly: /usr/lib/mono/g
Loaded assembly: /usr/lib/mono/g
Error: false, Message:
Error: true, Message:
```

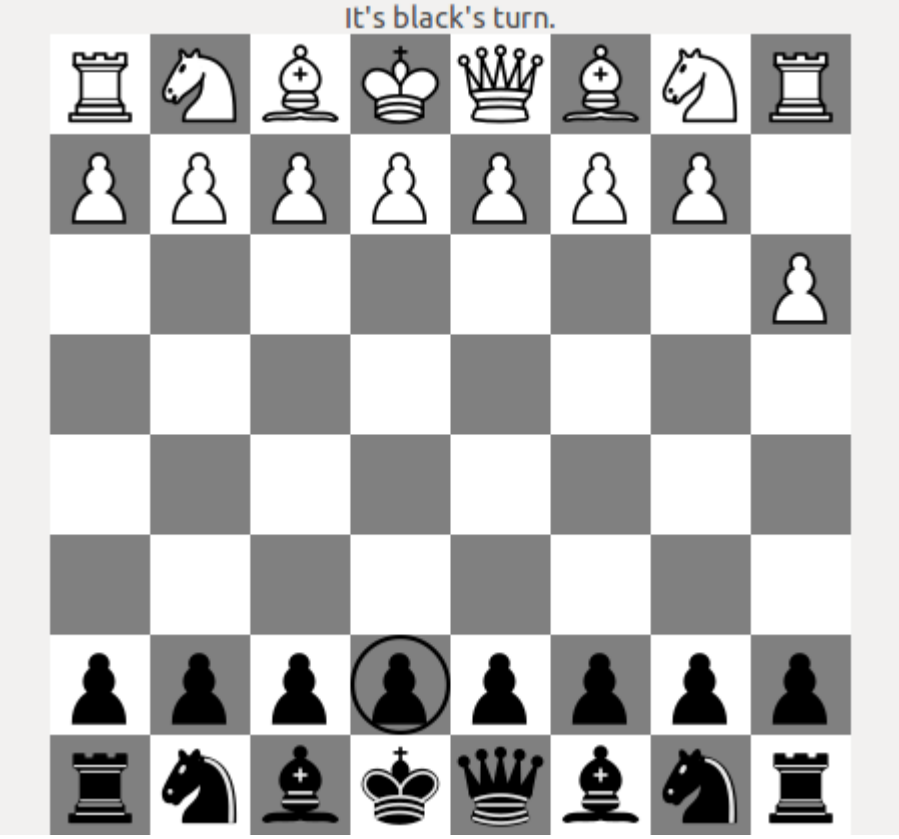
Test di movimento di due caselle del pedone(4):

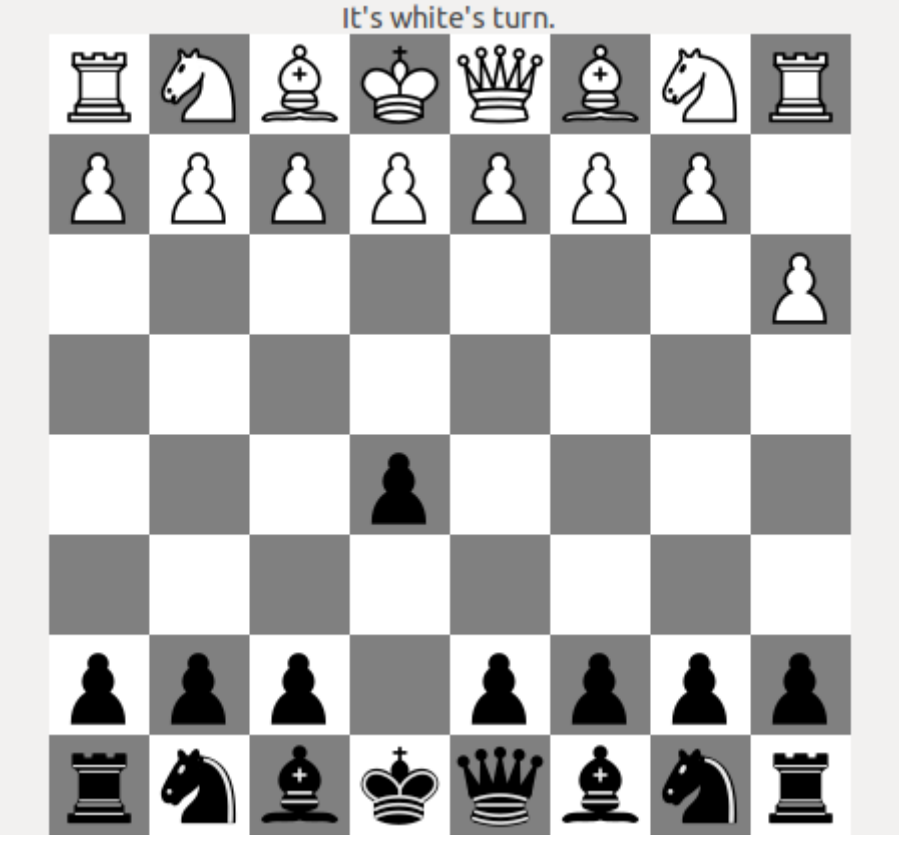


Test di movimento del cavallo(primo turno)(5)

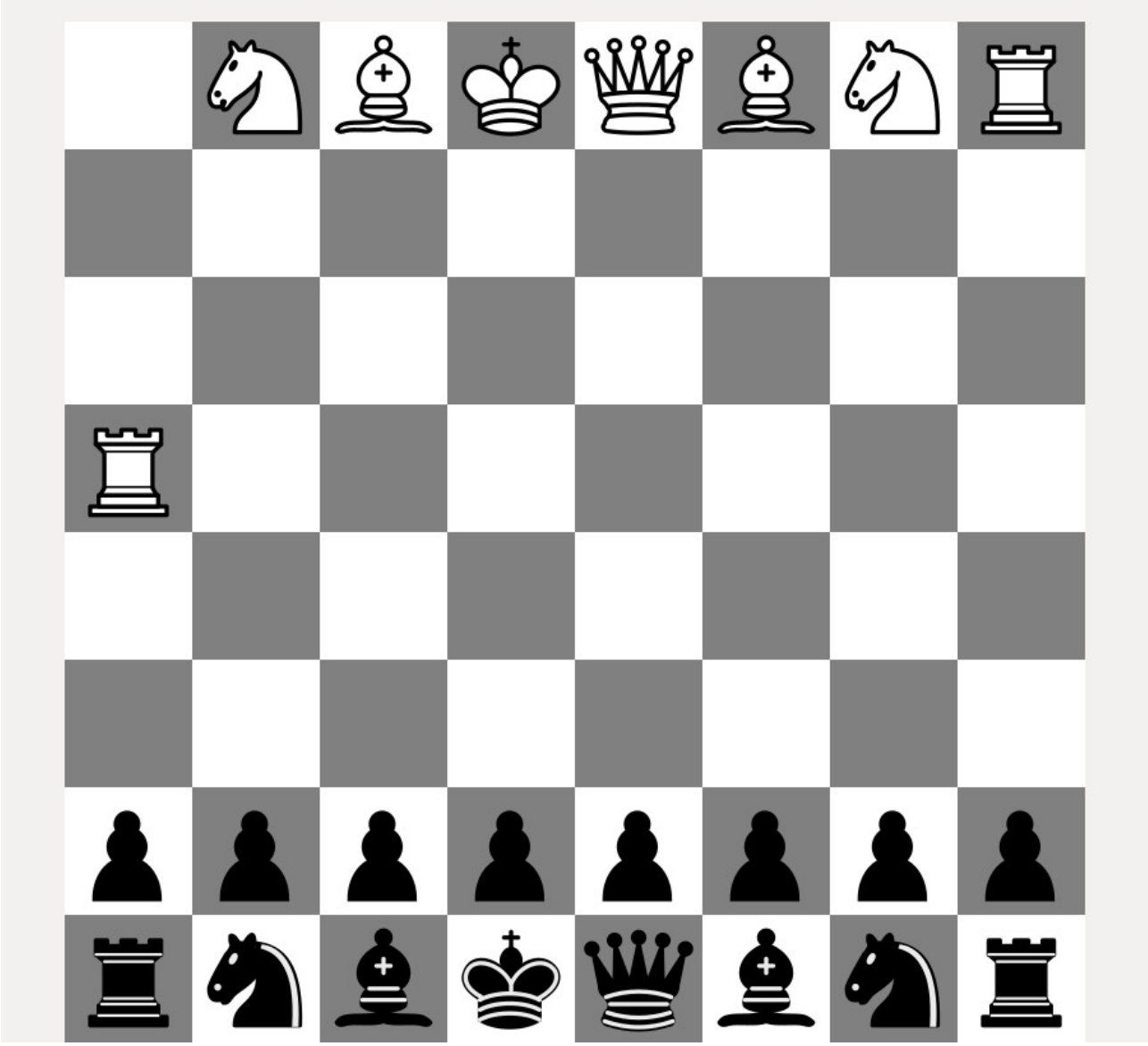


Test di selezione e movimento di due caselle del pedone nero(0-4):
Per completezza riportiamo anche un test sulla selezione del pedone nero e sul suo movimento:

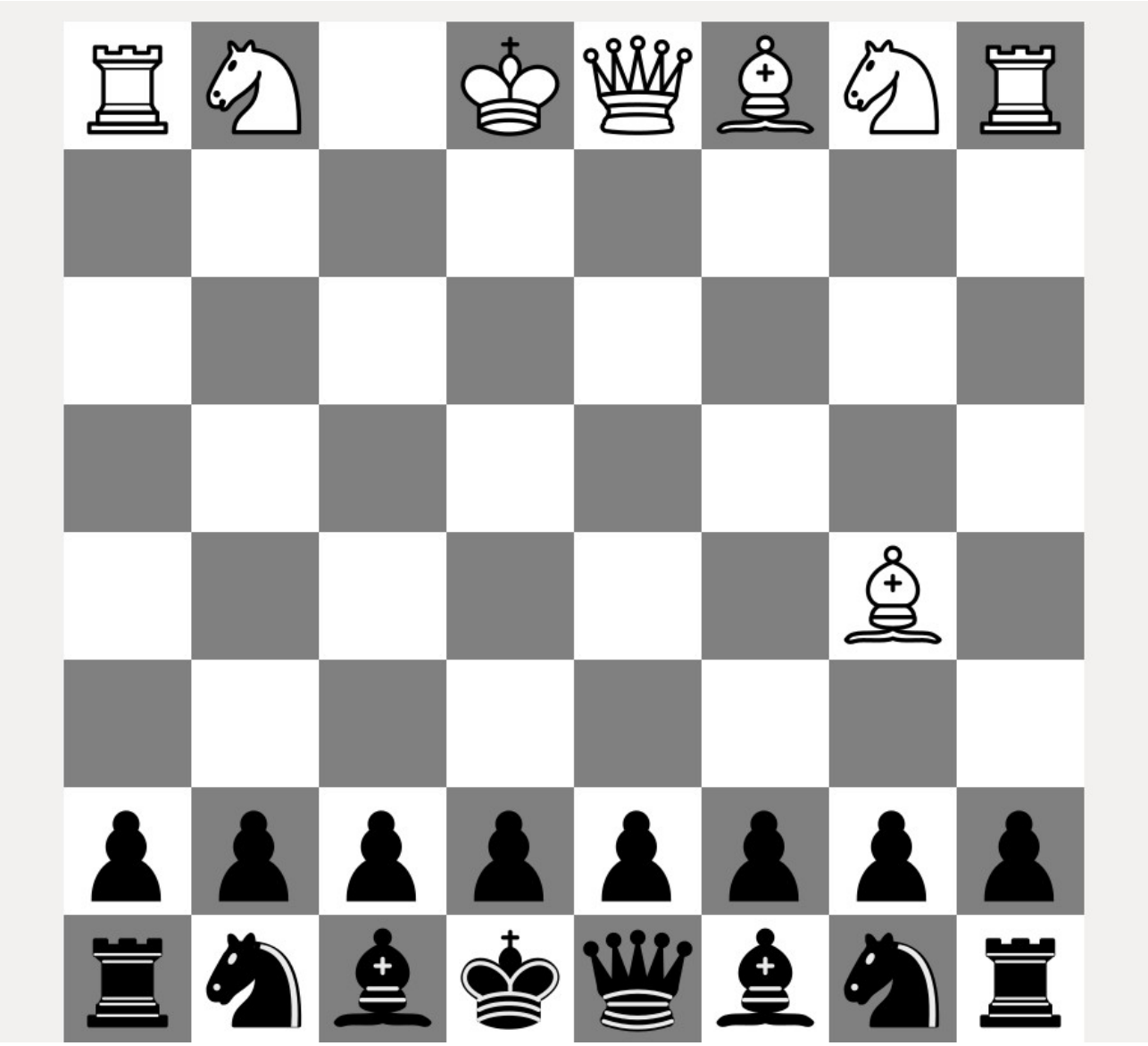




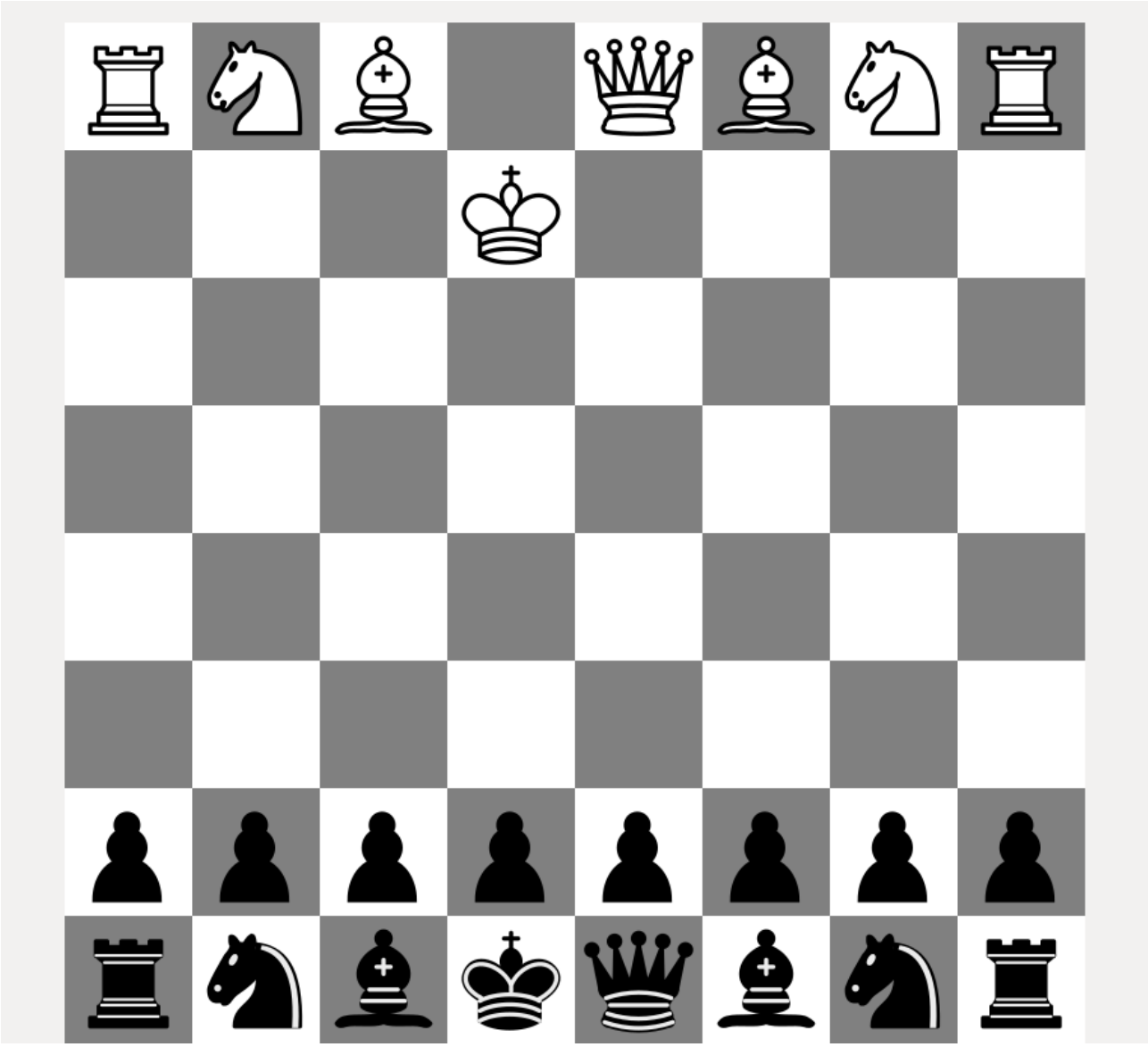
Test del movimento della torre(6):
Da qui fino al test di cattura dell'alfiere per comodità è stata rimossa la fila di pedoni bianchi



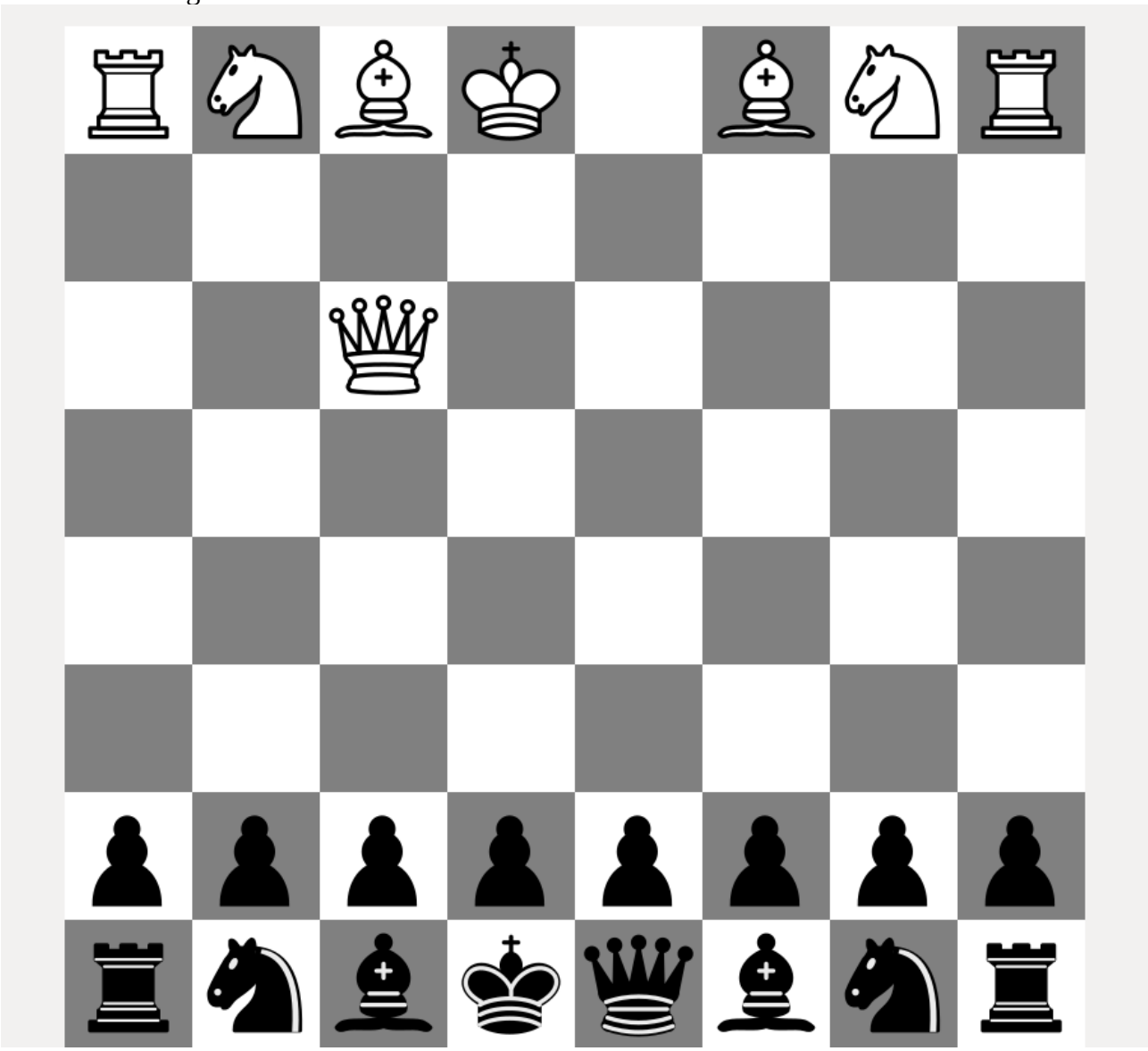
Test di movimento dell'alfiere(7):



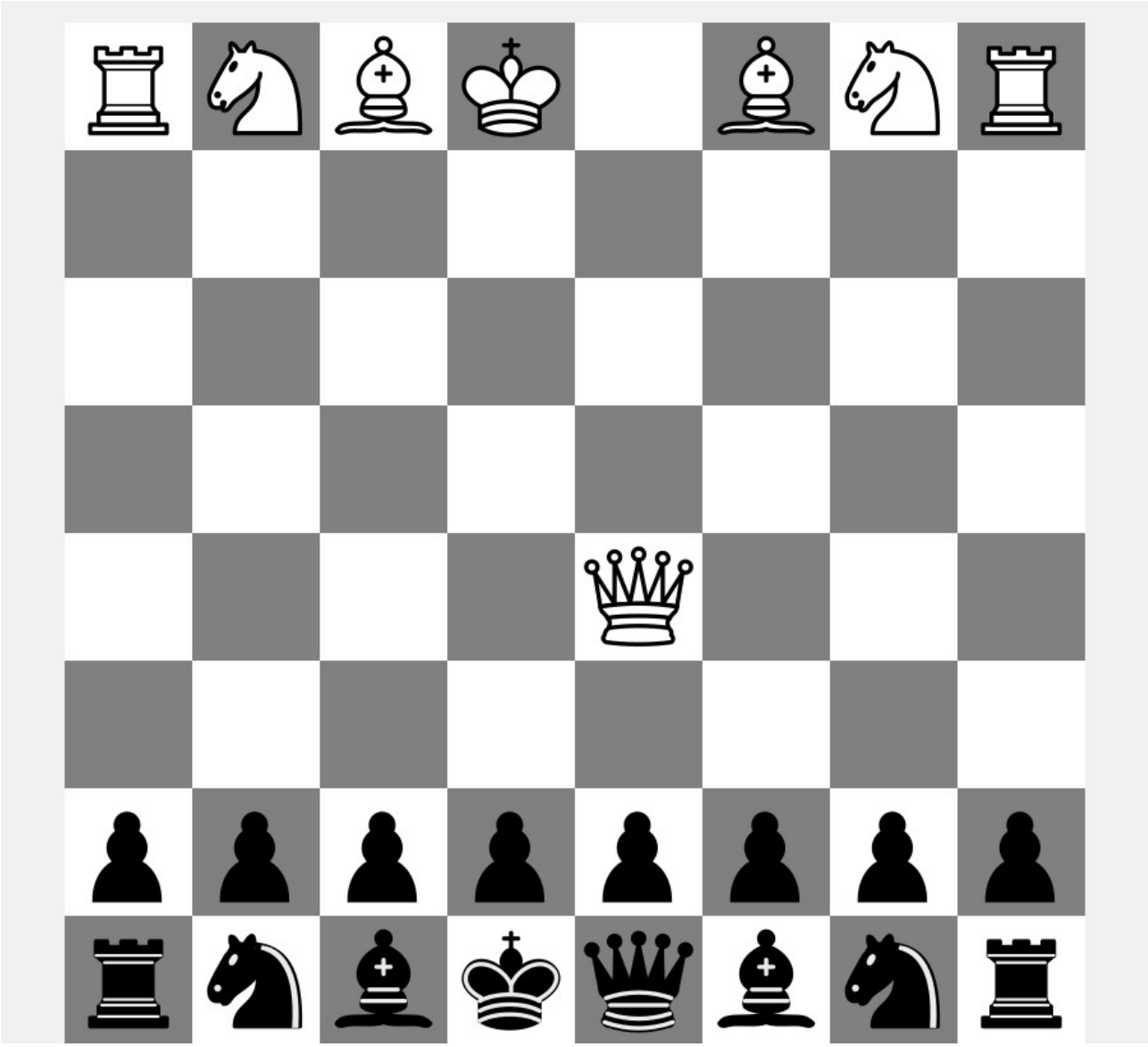
Test di movimento del re(8):



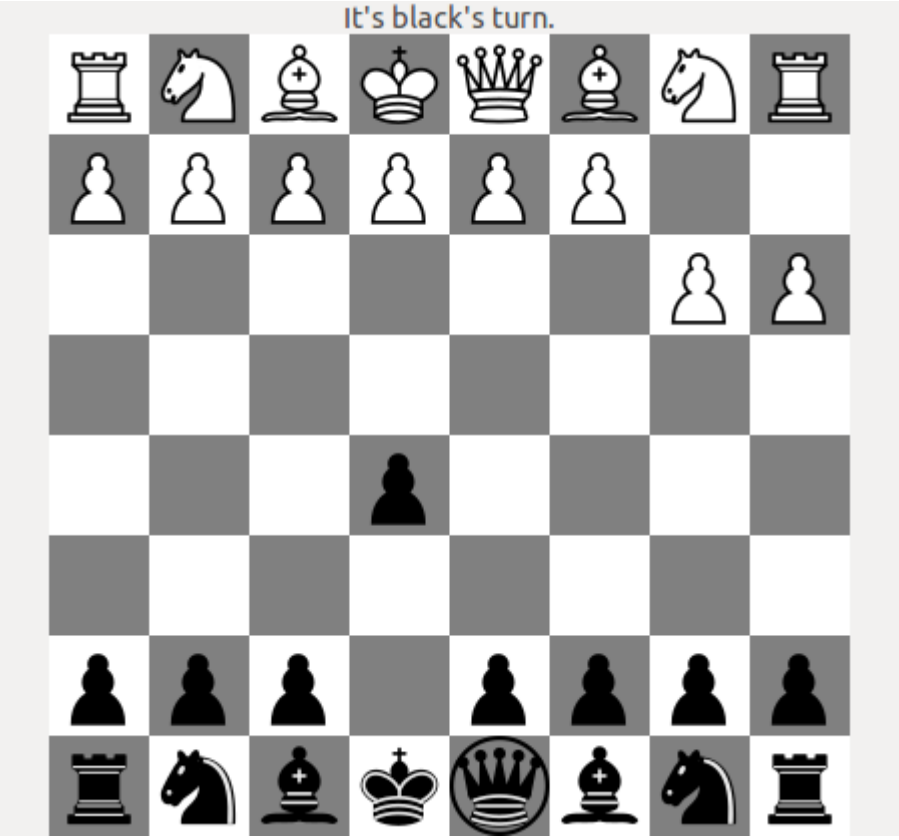
Test di movimento della regina(verticale e diagonale)(9):
-Movimento diagonale



-Movimento verticale:



Test di selezione e movimento della regina nera(0-9):
Si riporta di seguito un test sullla selezione e sul movimento della regina nera:



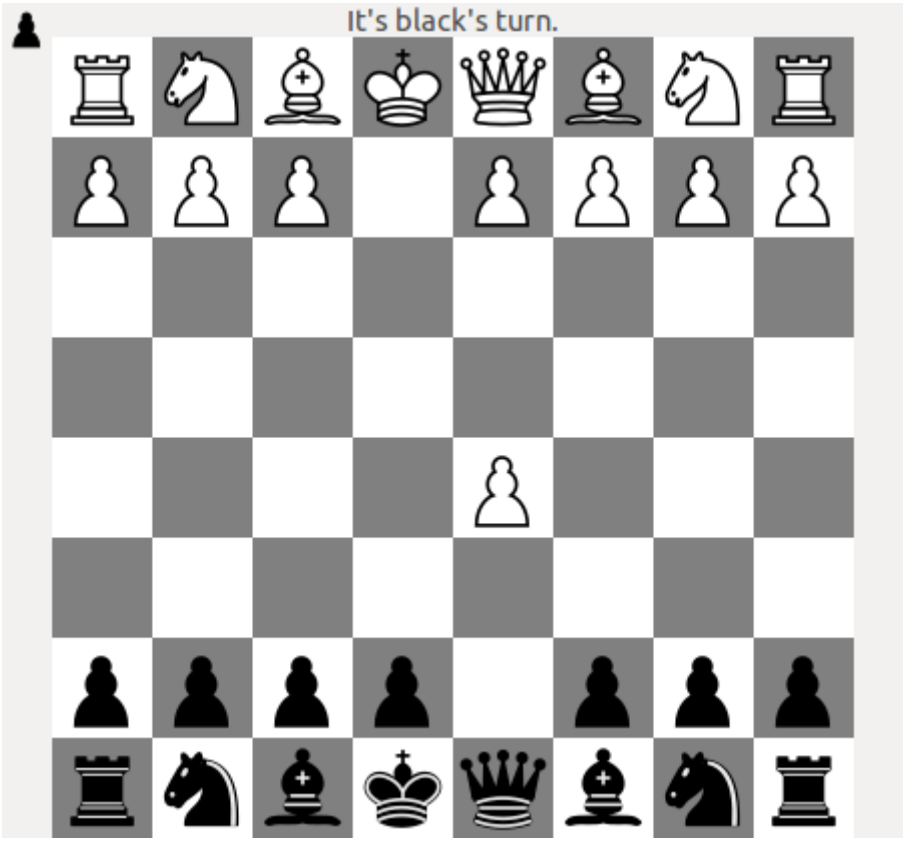
Test di movimento in un momento qualsiasi di pedone e cavallo(10-11-12):
-Scacchiera durante una partita in corso prima dei movimenti:

-Scacchiera dopo il movimento di una casella dei pedoni sulla parte destra:

-Scacchiera dopo il movimento di due caselle dei penultimi pedoni a destra:

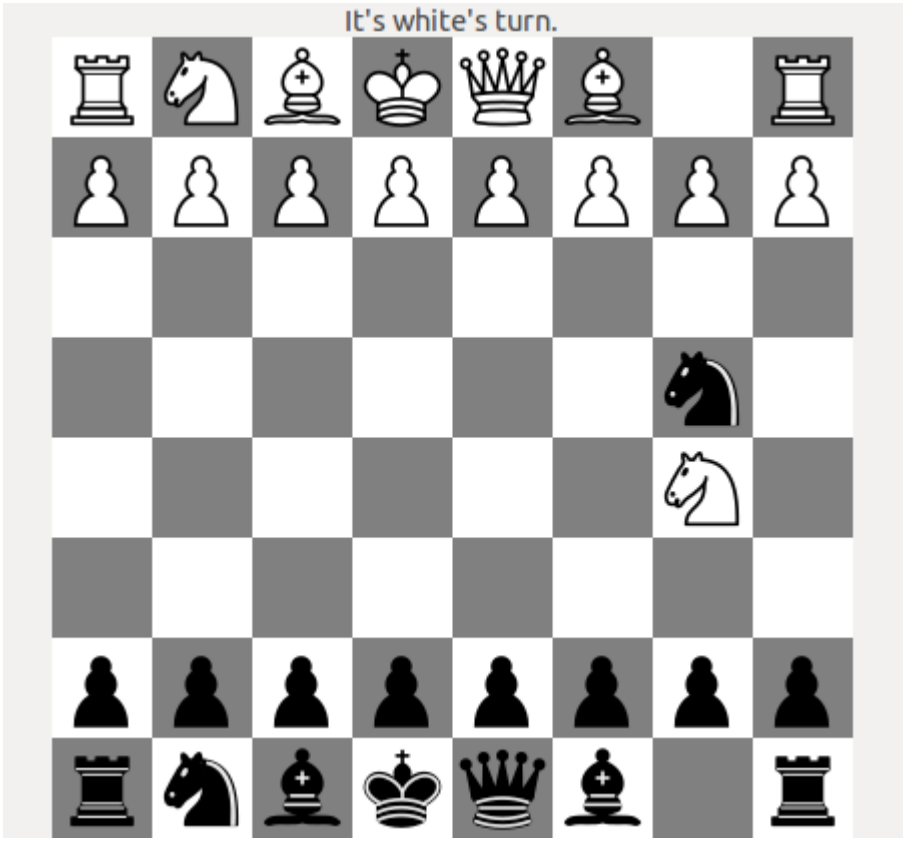
-Scacchiera dopo il movimento dei cavalli situati a destra della scacchiera:

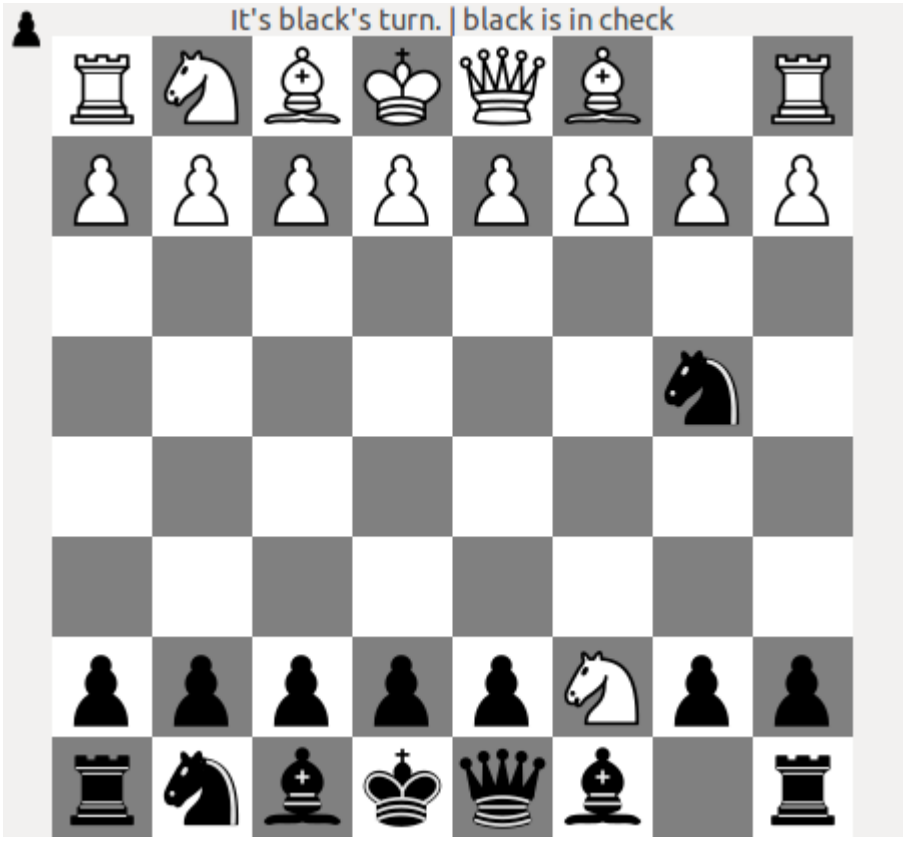
Test di cattura effettuata dal pedone(13):



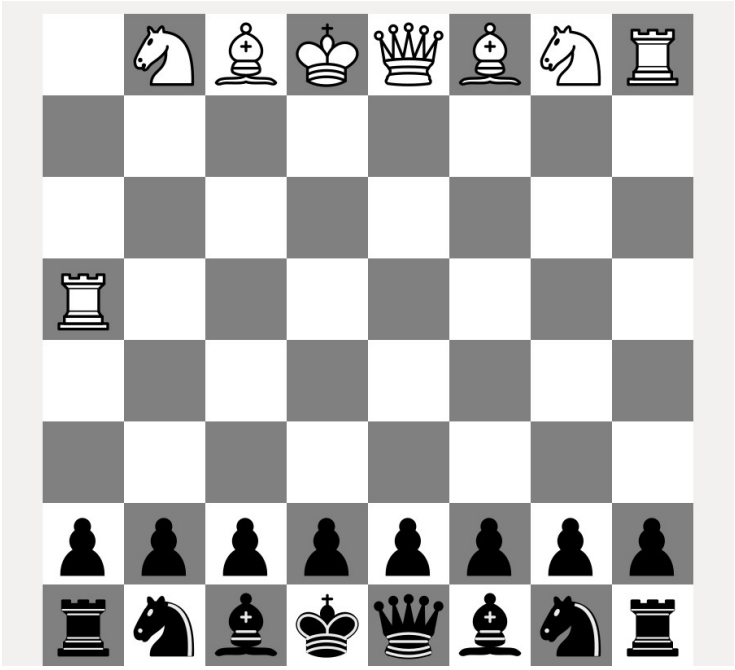
Qui viene anche testato il funzionamento della sidebar:il pedone catturato viene correttamente rimosso e inserito nella sidebar sinistra.

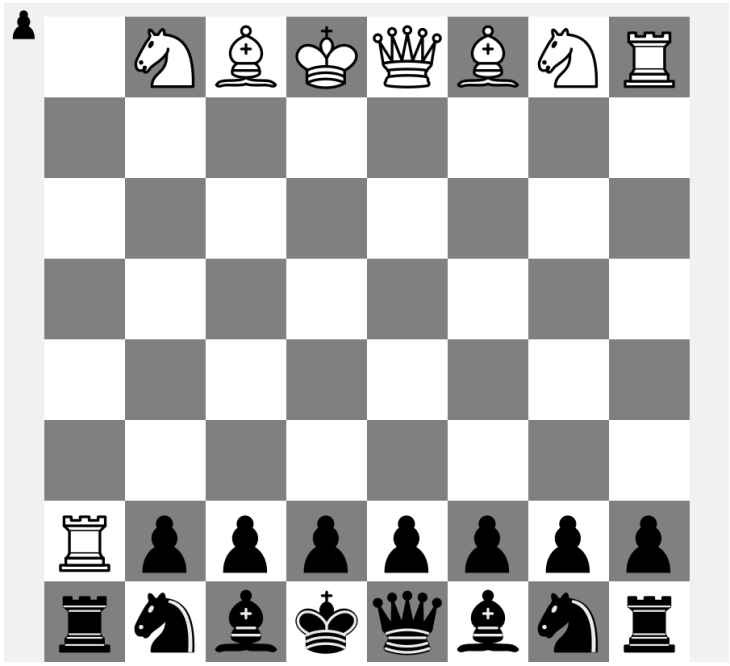
Test di cattura effettuata dal cavallo e test di scacco(14-22):



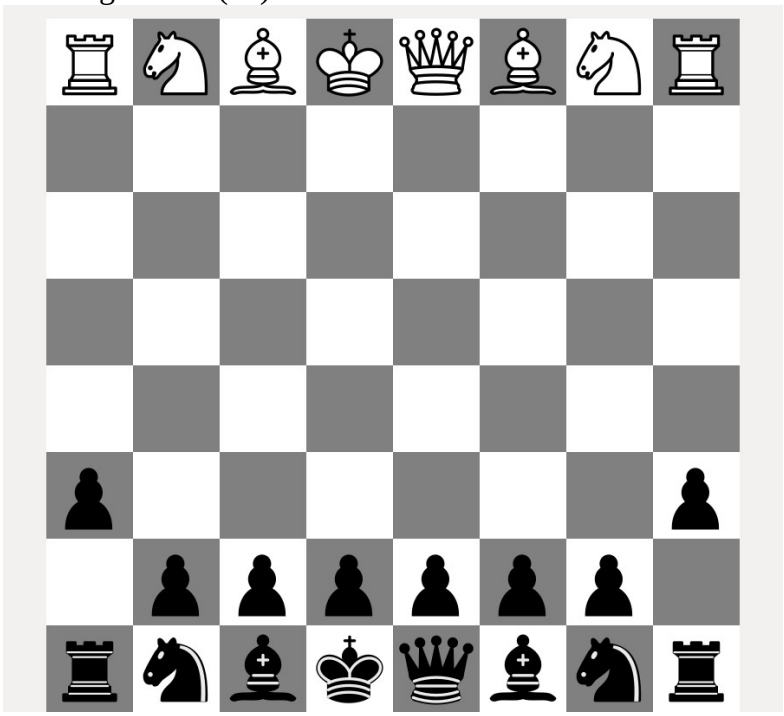


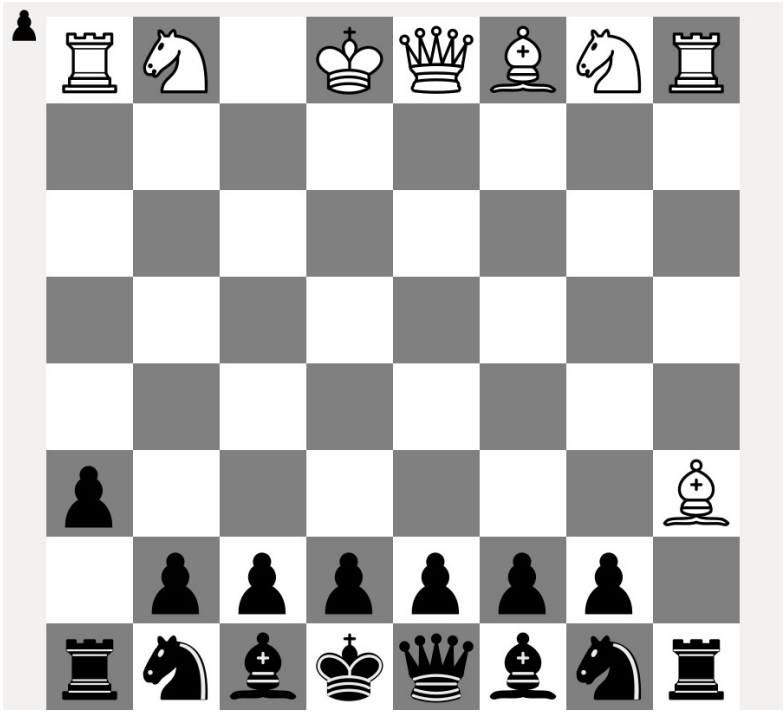
Test della cattura effettuata dalla torre(15):

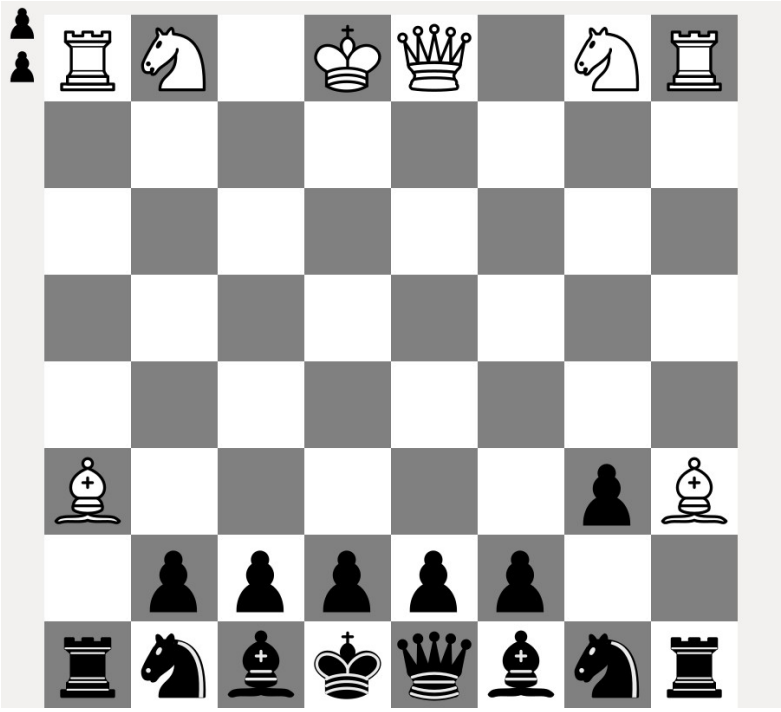




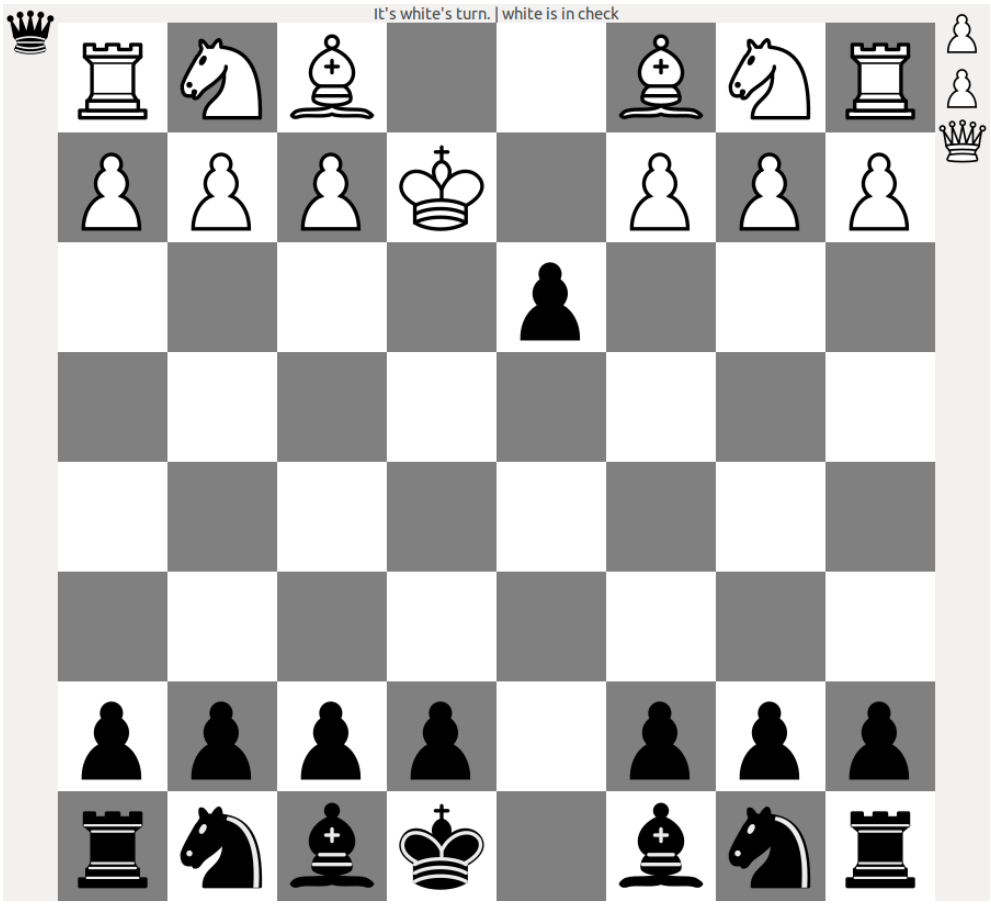
Test di cattura effettuato dagli alfieri(16):

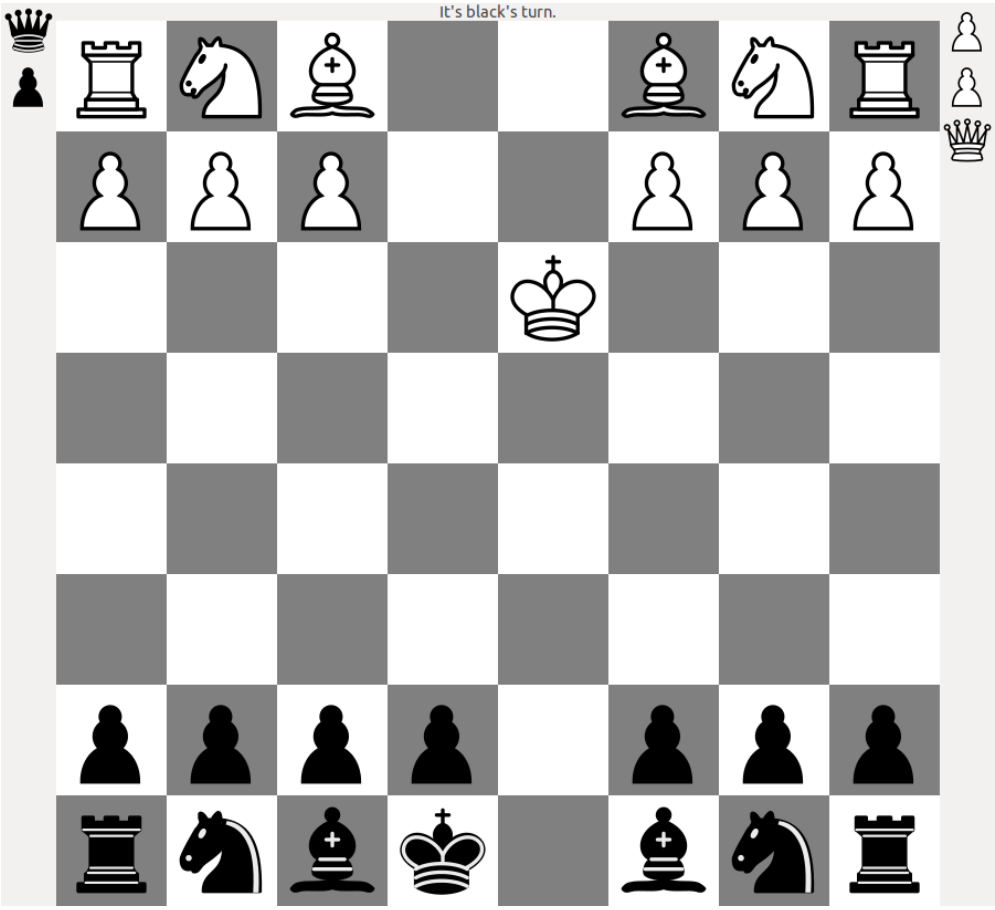






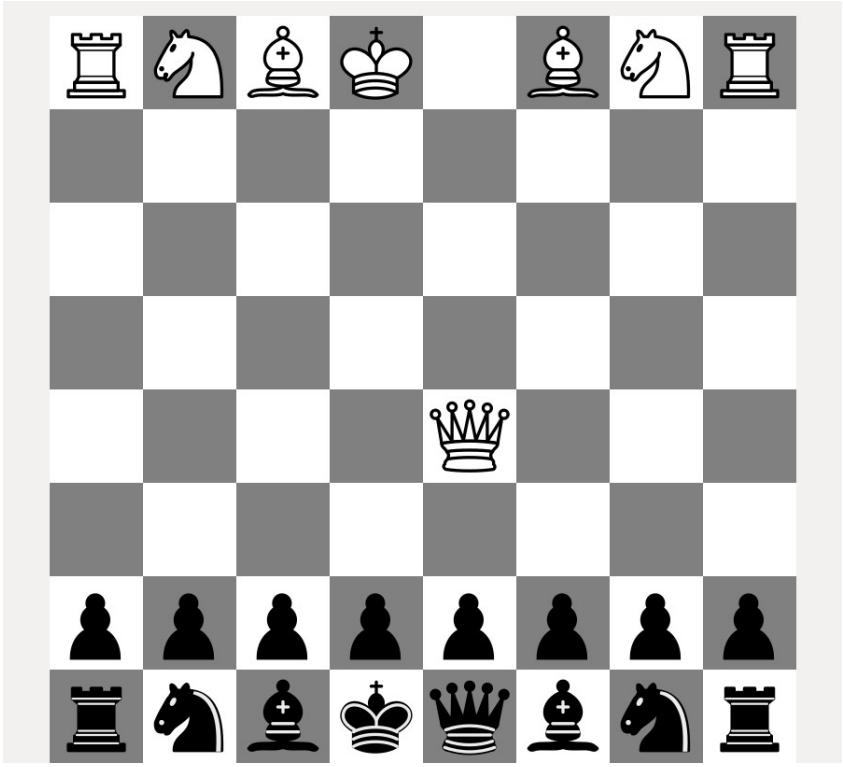
Test cattura effettuata dal re(17):

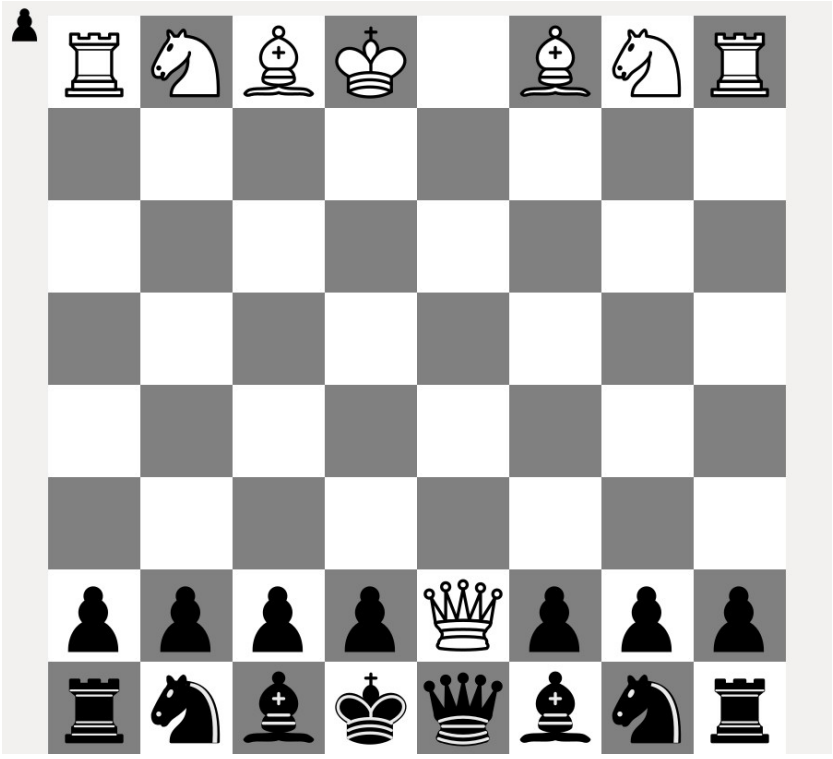




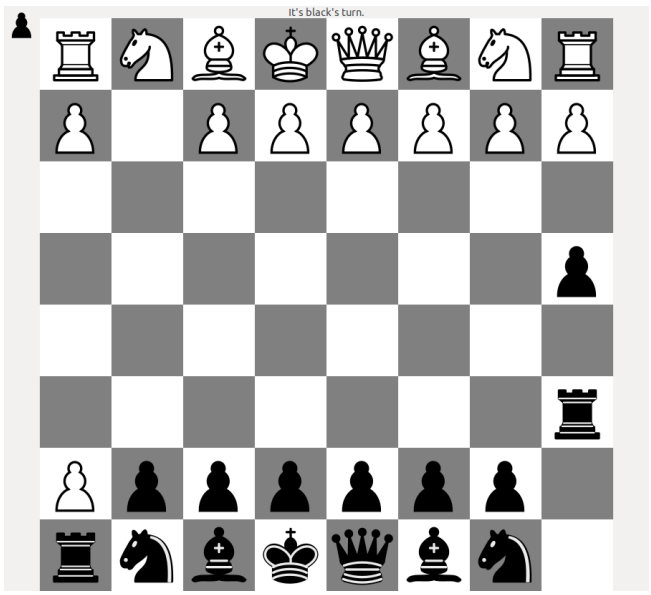
Dopo la cattura effettuata dal re viene rimosso lo stato di check.

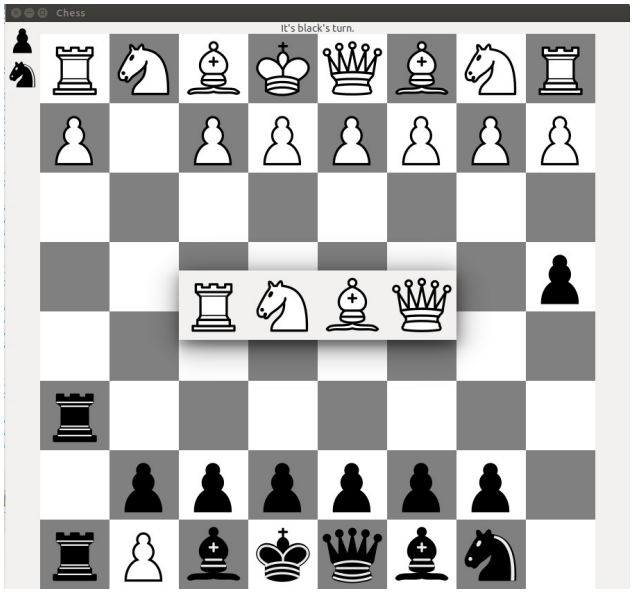
Test della cattura effettuata dalla regina(18)

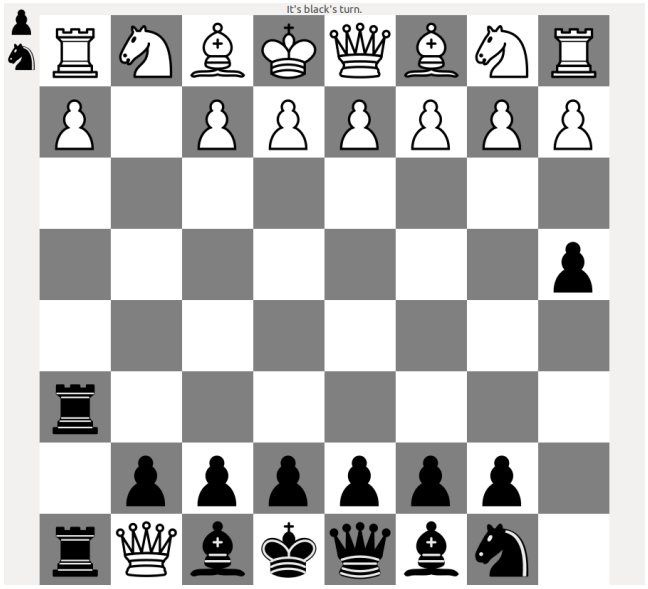




Test della promozione del pedone(19):

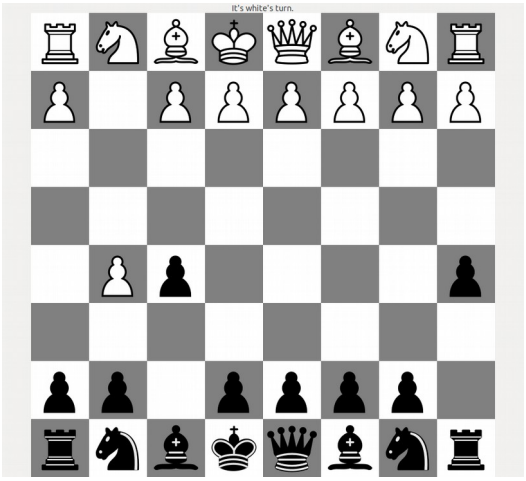


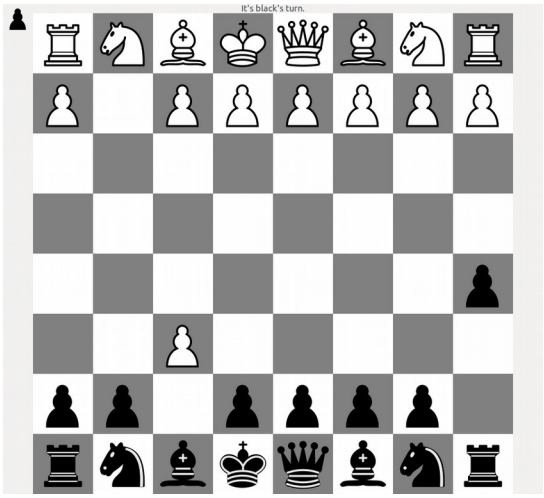




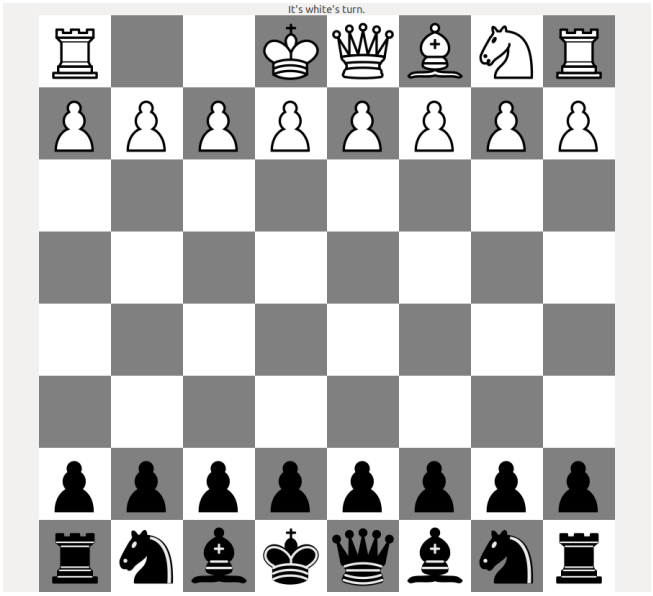
Dai test effettuati è visibile la corretta apertura ed il corretto funzionamento del popup contenente le pedine in cui il pedone viene promosso.

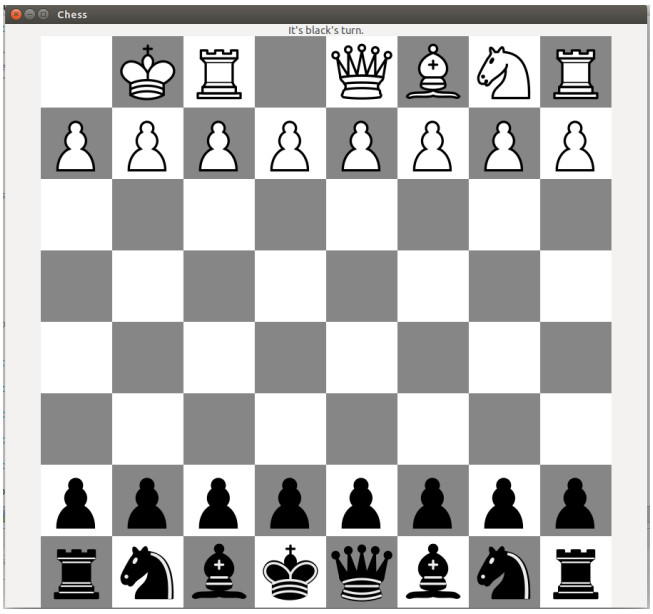
Test En passant(20):

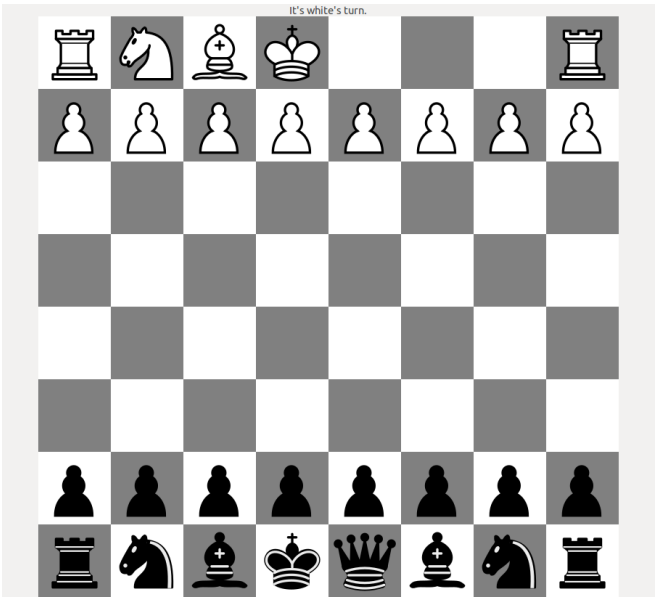


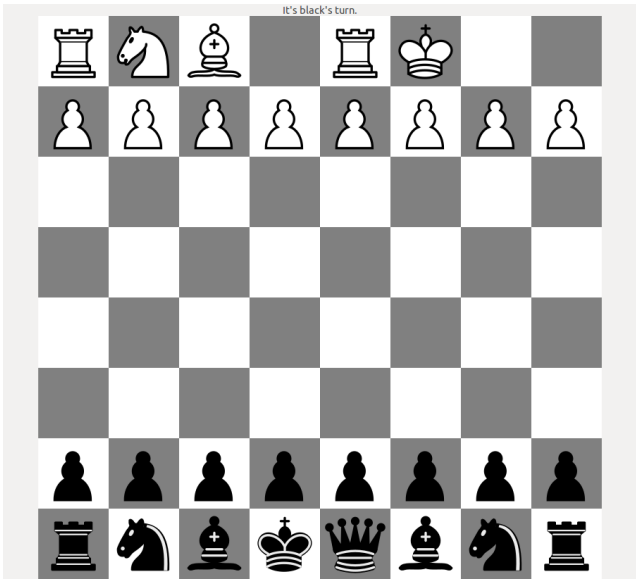


Test sul funzionamento dell'arrocco(21):
Sono stati testati i casi dell'arrocco corto(torre che viene spostata di due caselle) e di quello lungo(torre che viene spostata di 3 caselle)

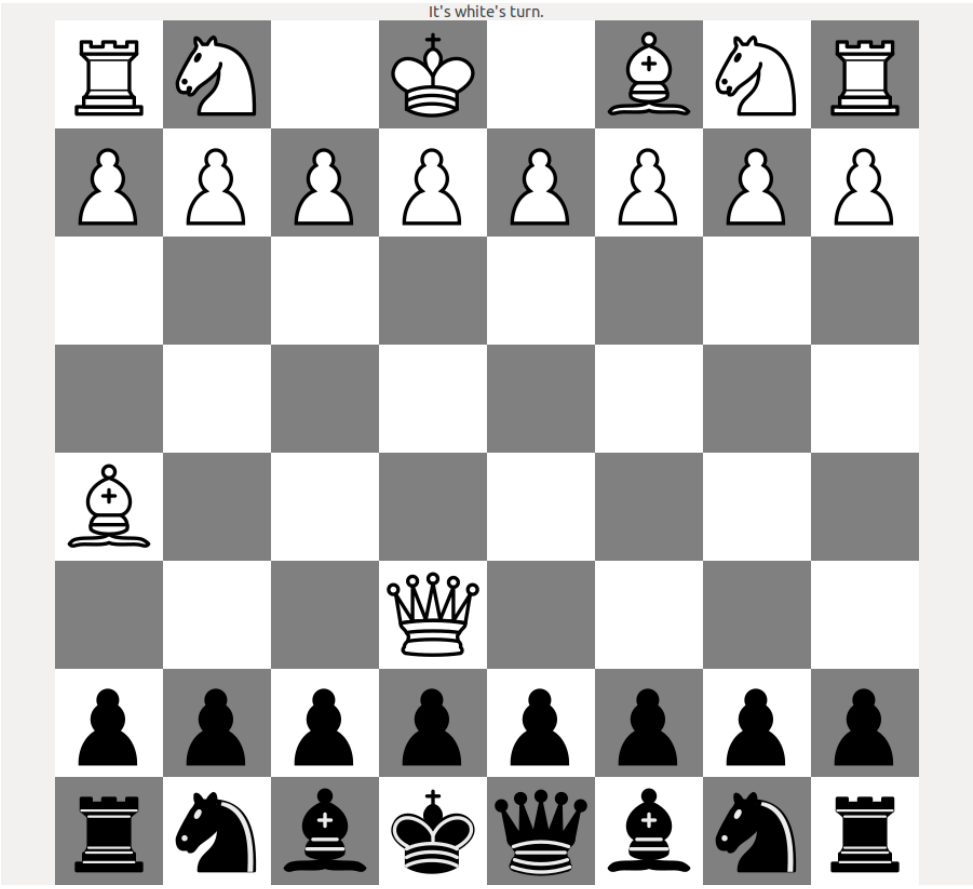


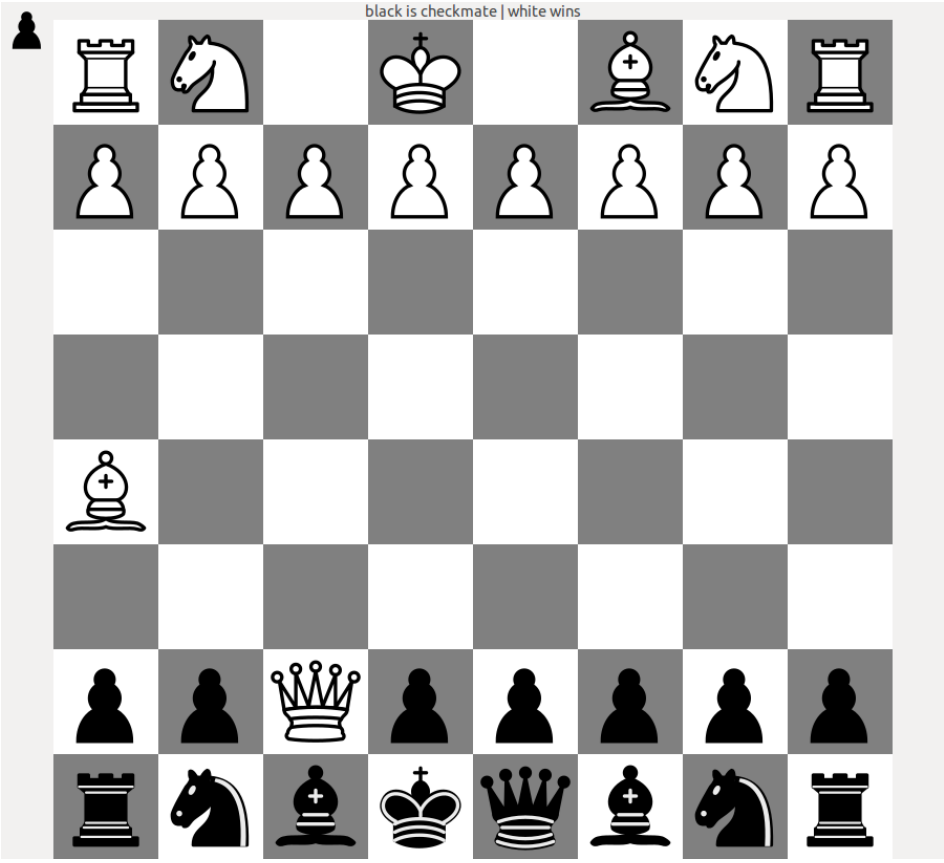






Test sul funzionamento delloscacco matto(23):





La partita termina correttamente e il giocatore viene avvertito di aver vinto la suddetta.

Si ringraziano i tester, specialmente Flavio Mascetti e Marco Tamagno.