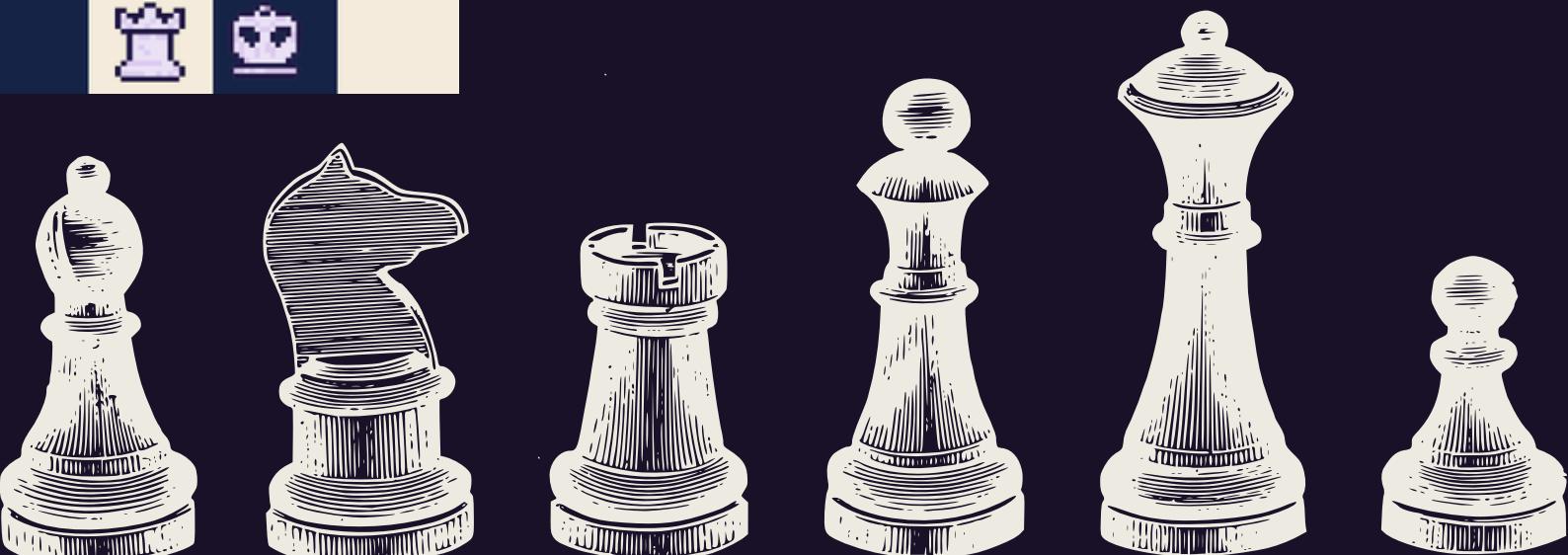


TESINA DEL PROGETTO SOFTWARE
“JAVA CHESS”



java
chess



SOMMARIO

1. INTRODUZIONE E PANORAMICA DELL'APPLICAZIONE

2. MODALITA' DI TESTING

2.1 REQUISITI FUNZIONALI E NON FUNZIONALI

2.2 BLACKBOX

3. MODELLAZIONE UML

3.1 USE CASE DIAGRAM (INTERAZIONE UTENTE)

3.2 CLASS DIAGRAM (STRUTTURA DEL SISTEMA)

3.3 SEQUENCE DIAGRAM (ESECUZIONE DI UNA MOSSA)

3.4 ACTIVITY DIAGRAM (LOGICA DEI PROCESSI)

3.5 STATE MACHINE DIAGRAM (STATI DELLA PARTITA)

4. ARCHITETTURA SOFTWARE

4.1 I DESIGN PATTERN

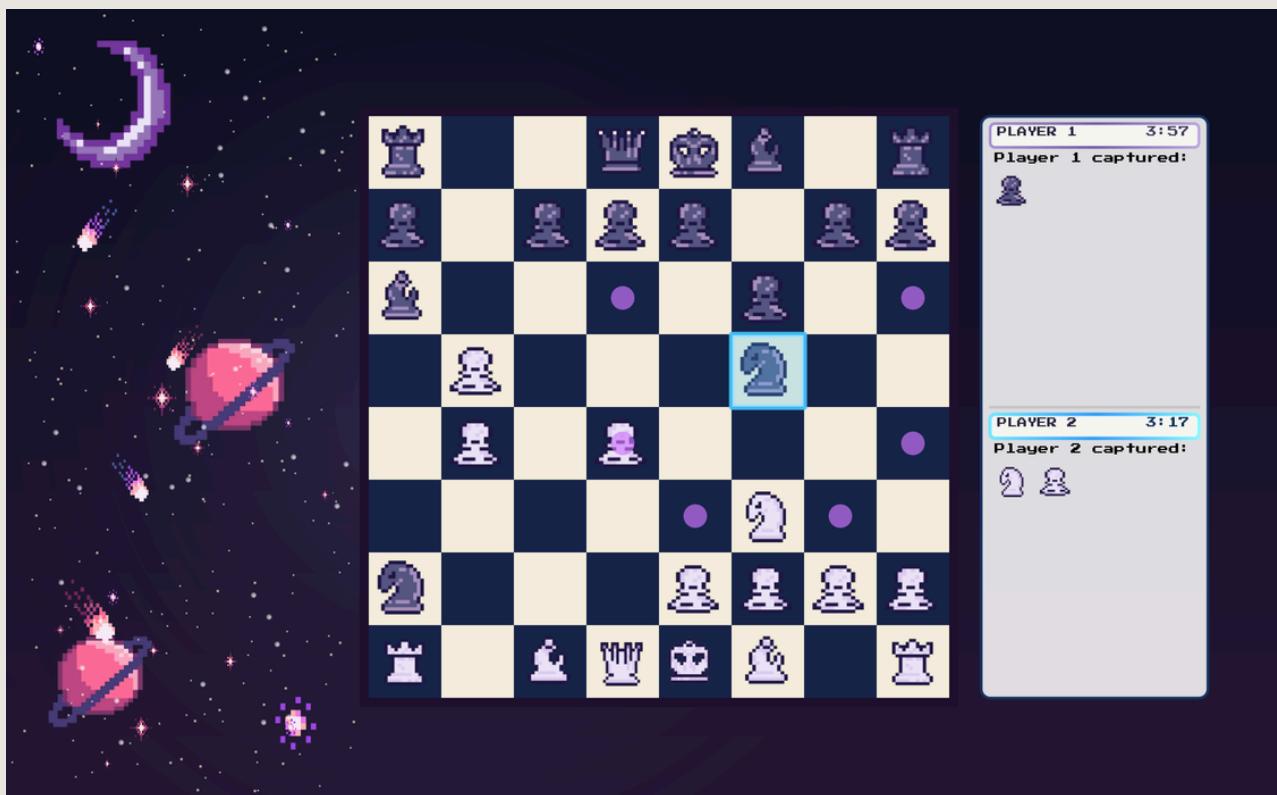
5. CONCLUSIONI

INTRODUZIONE E PANORAMICA DELL'APPLICAZIONE

Il progetto “**Java-Chess**” nasce con l’obiettivo di realizzare una versione digitale del gioco degli scacchi, sfruttando i paradigmi della Programmazione a Oggetti offerti da Java e dalla libreria grafica Swing.

A differenza di una semplice scacchiera statica, l’applicazione punta a offrire un’esperienza utente coinvolgente attraverso un’interfaccia grafica curata, la cui estetica è volutamente ispirata allo stile pixel-art delle console **ATARI**.

Il motore logico implementato gestisce rigorosamente le regole del gioco, incluse le casistiche più complesse come l’Arrocco e la cattura En Passant. Sebbene concepito inizialmente come caso di studio per il corso di Programmazione a Oggetti, lo sviluppo si è evoluto in un progetto personale approfondito, che mi ha visto curare autonomamente l’intera realizzazione degli asset grafici e dello stile.



INTRODUZIONE E PANORAMICA DELL'APPLICAZIONE

Una volta avviato il programma, si presenta la schermata iniziale con la possibilità di scegliere i nomi.

Grazie alla classe **GameGUI**, due giocatori possono sfidarsi facilmente sullo stesso computer.

Il sistema facilita la visualizzazione delle mosse valide sulla scacchiera, applica tutte le regole ufficiali e monitora il tempo e i pezzi catturati in un elegante pannello laterale.

Non appena la partita termina per scacco matto o patta, è possibile ricominciare a giocare con un click.

MODALITA' DI TESTING

2.1 REQUISITI FUNZIONALI E NON FUNZIONALI

REQUISITI FUNZIONALI

Il sistema garantisce le seguenti funzionalità essenziali per una partita corretta:

- **Movimento pezzi:** spostamento sulla scacchiera secondo le regole standard (ad esempio: diagonale per l'Alfiere, ad "L" per il Cavallo...).
- **Mosse speciali:** rilevamento e gestione automatica dell'Arrocco (corto/lungo) e della cattura En Passant.
- **Controllo partita:** verifica automatica di Scacco, Scacco Matto e condizioni di Patta (stallo o materiale insufficiente).
- **Interfaccia utente:** evidenziazione visiva del pezzo selezionato e delle mosse possibili; visualizzazione dei pezzi catturati.
- **Timer:** gestione di due orologi indipendenti con sconfitta immediata all'esaurimento del tempo.

REQUISITI NON FUNZIONALI

- **UX (Esperienza Utente):** grafica pixel-art ad alto contrasto che unisce estetica accattivante e massima leggibilità per l'utente.
- **Performance:** fluidità costante garantita dall'uso della cache, che carica le immagini in memoria una sola volta evitando rallentamenti.
- **Robustezza:** il gioco ignora automaticamente input errati o click fuori dalla scacchiera, prevenendo errori critici o blocchi.
- **Portabilità:** eseguibile su qualsiasi computer grazie a Java e facilmente scaricabile tramite il mio GitHub personale.

MODALITA' DI TESTING

2.2 BLACKBOX

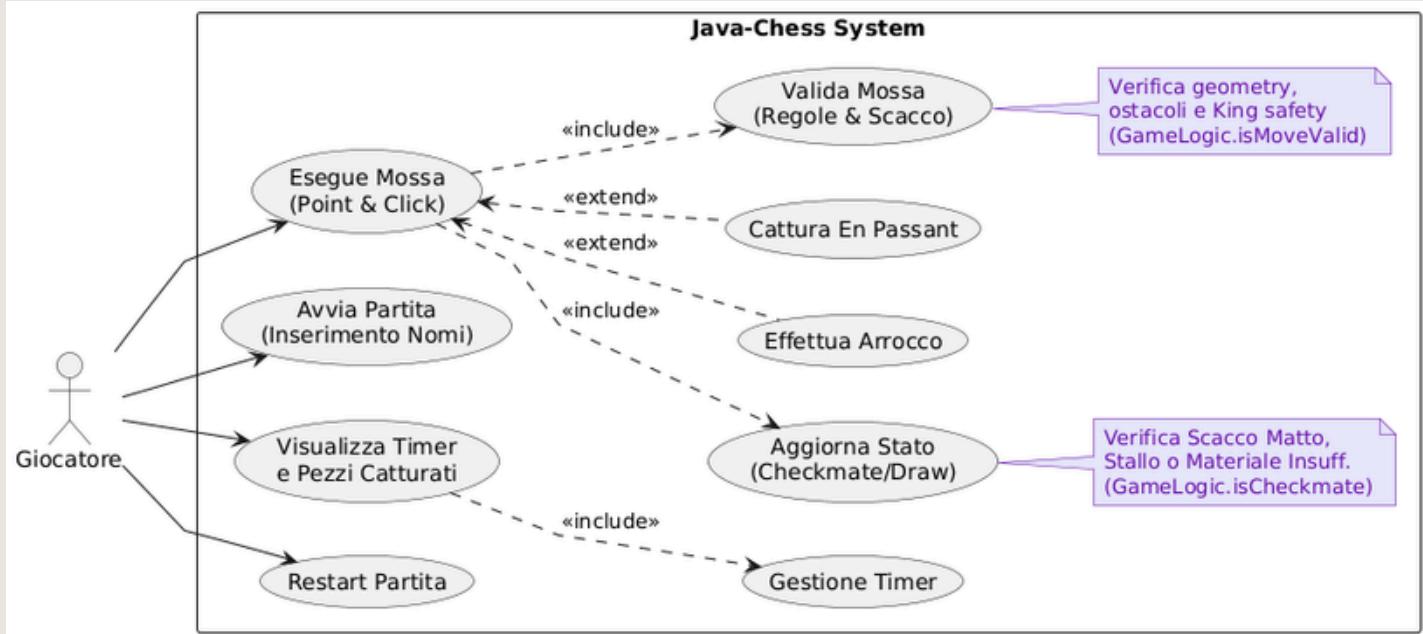
Ho testato il software comportandomi esattamente come farebbe un utente finale, ovvero ignorando completamente il codice che ho scritto. Mi sono concentrata esclusivamente su due aspetti:

- **Input: l'azione che compio.**
- **Output: la reazione del gioco.**

Ho utilizzato due delle tecniche più frequenti nel Blackbox:

- **Equivalence Partitioning:** invece di testare ogni singola mossa possibile (che sarebbe impossibile), ho raggruppato le mosse in "classi". Ad esempio, ho assunto che se il movimento base di un pedone funziona, funzionerà per tutti i pedoni. Allo stesso modo, ho testato una mossa palesemente errata (come muovere la torre in diagonale) per assicurarmi che il sistema rifiutasse quella categoria di errori.
- **Boundary Value Analysis:** ho testato i limiti della scacchiera, provando a muovere pezzi dalla prima o dall'ultima riga verso l'esterno.

MODELLAZIONE UML: 3.1 USE CASE



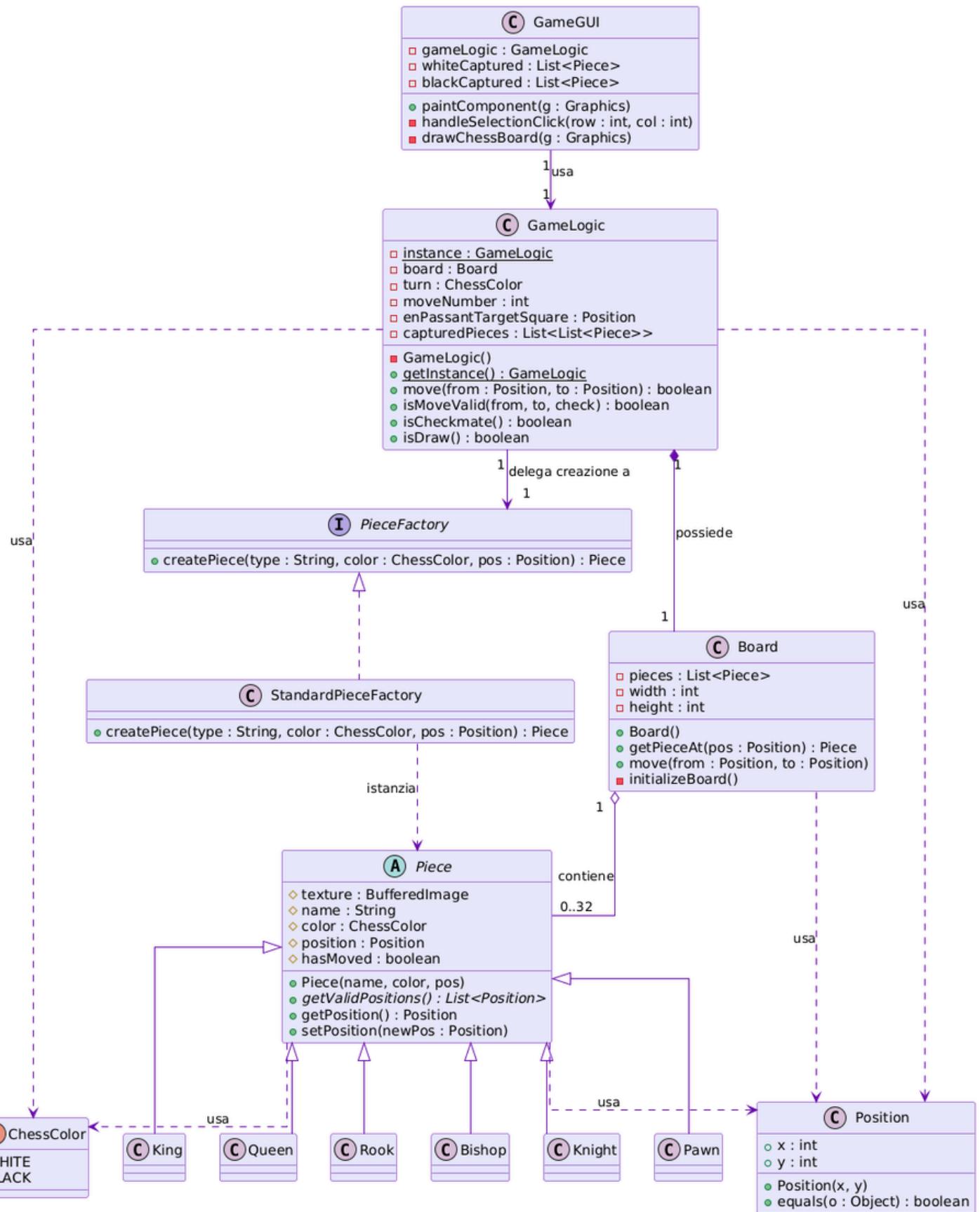
Il diagramma modella le interazioni tra l'attore Giocatore e il sistema Java-Chess.

Le funzionalità principali attivate dall'utente sono:

- 1. Avvia Partita:** Configurazione iniziale e inserimento dei nomi.
- 2. Esegue Mossa:** Azione centrale che innesca obbligatoriamente (tramite <<include>>) la Validazione (verifica geometrica mosse dei pezzi e sicurezza del Re) e il Controllo Fine Partita (Scacco Matto/Patta).
- 3. Visualizza Stato:** Monitoraggio real-time di timer e pezzi catturati.
- 4. Restart Partita:** Riavvio immediato della sessione.

Le mosse speciali Arrocco ed En Passant sono modellate come relazioni di estensione (<<extend>>), attivandosi solo al verificarsi di specifiche condizioni di gioco.

MODELLAZIONE UML: 3.2 CLASS DIAGRAM



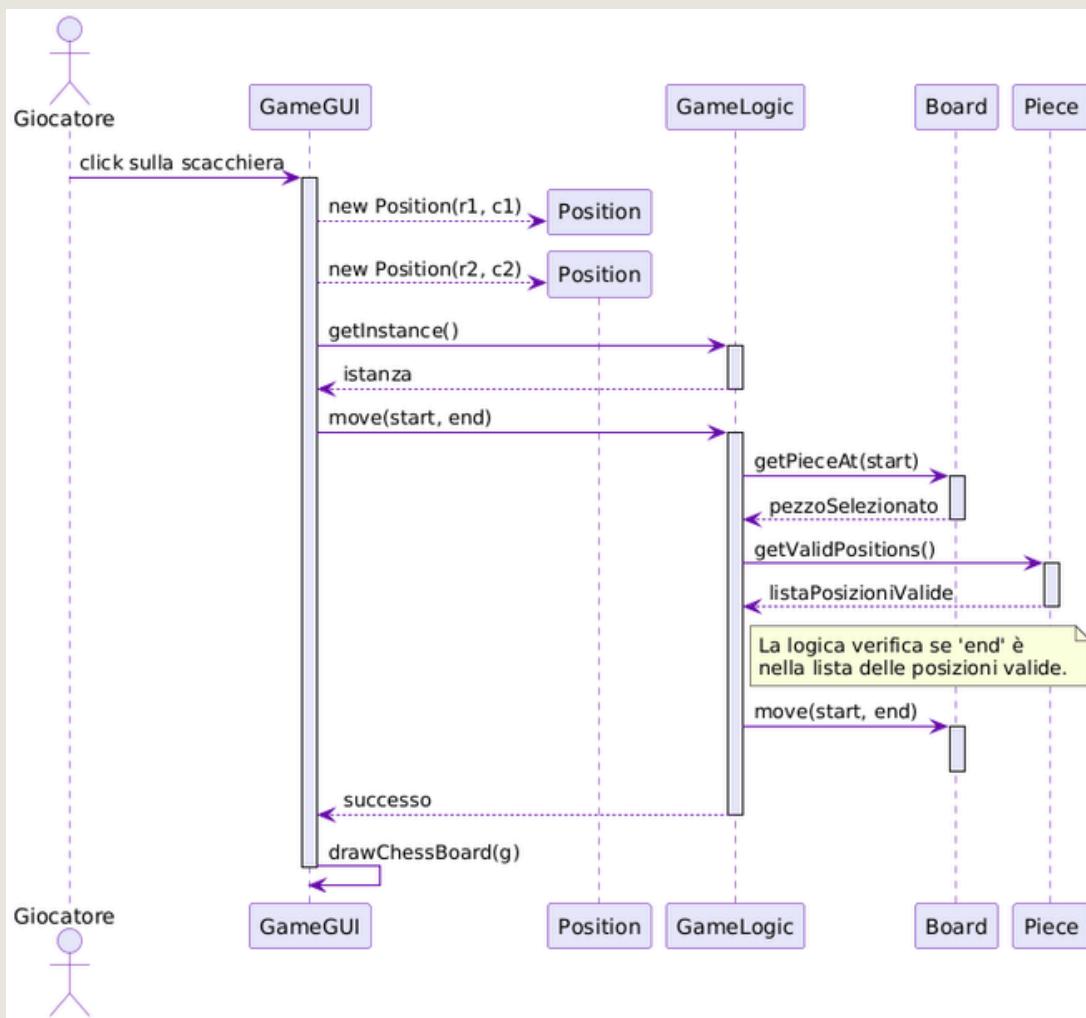
MODELLAZIONE UML: 3.2 CLASS DIAGRAM

Il class diagram proposto mostra l'architettura logica del software, evidenziando come la classe **GameLogic** gestisca le regole della partita e garantisca l'unicità del motore di gioco attraverso il pattern Singleton.

La creazione dei diversi pezzi è delegata a **PieceFactory** e **StandardPieceFactory**, che permettono di generare gli oggetti in modo separato dalla logica principale per mantenere il codice pulito e modulare.

Ogni pezzo specifico eredita dalla classe base **Piece**, definendo le proprie regole di movimento, mentre la classe **Board** ha il compito di memorizzare la disposizione fisica dei pezzi sulla griglia di gioco. Infine, la **GameGUI** gestisce l'interfaccia grafica e l'interazione con l'utente, mentre la classe **Position** e l'enumerazione **ChessColor** servono a rappresentare rispettivamente le coordinate spaziali e i due schieramenti, assicurando una comunicazione corretta tra tutti i componenti.

MODELLAZIONE UML: 3.3 SEQUENCE DIAGRAM



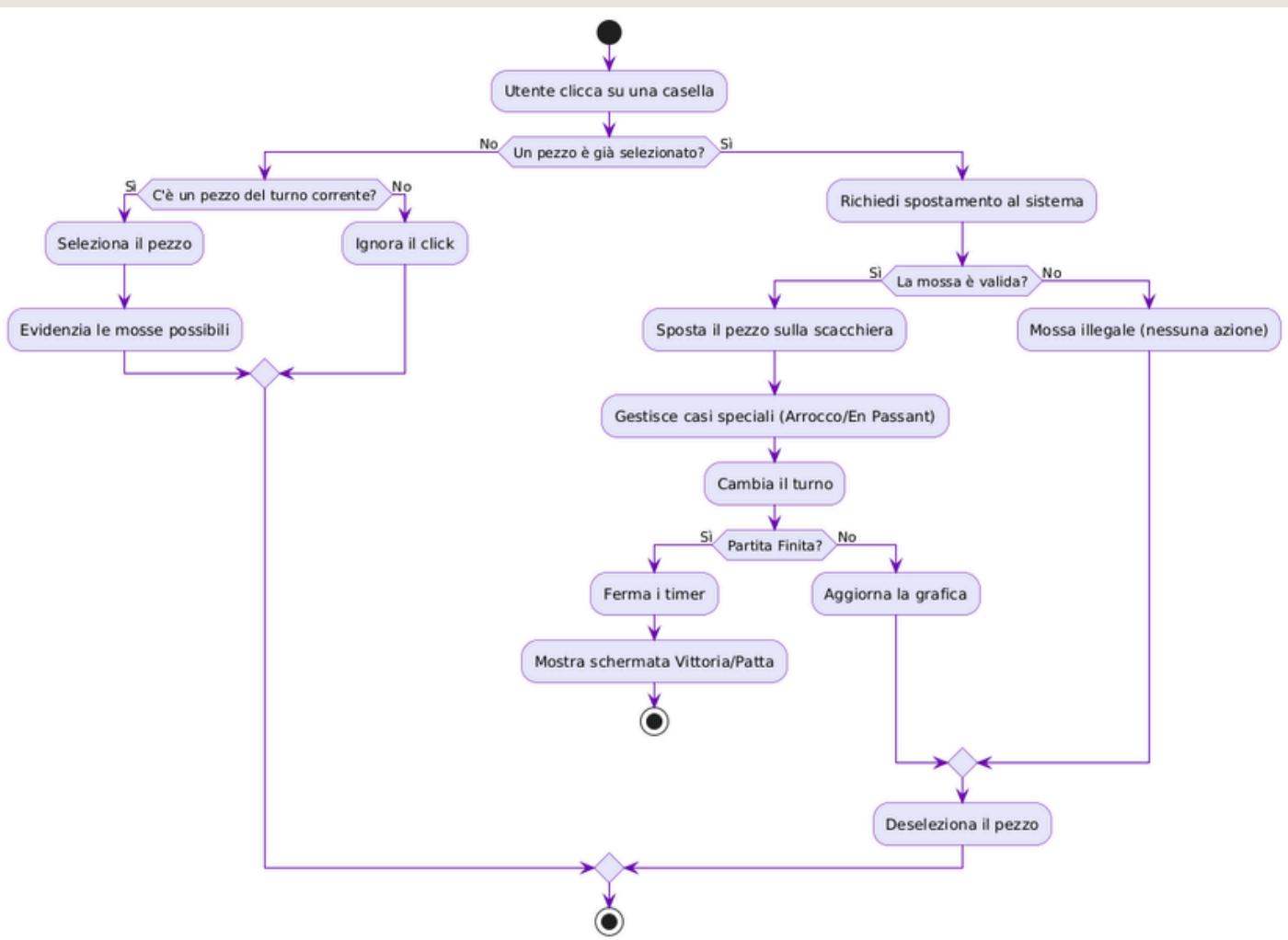
Il sequence proposto illustra esclusivamente lo svolgimento di una mossa normale, descrivendo l'interazione tra i componenti attivi dal momento del click fino all'aggiornamento visivo della scacchiera.

Tutto ha inizio quando **il giocatore interagisce con la GameGUI**, la quale crea due oggetti Position per definire precisamente le coordinate di partenza e di destinazione del pezzo.

Successivamente, **l'interfaccia interroga l'unica istanza della GameLogic per elaborare il movimento**, spingendo la logica a recuperare il pezzo selezionato dalla Board e a verificarne la validità consultando la lista delle posizioni permesse restituita dal metodo getValidPositions del pezzo stesso.

Una volta confermata la validità della destinazione, **la GameLogic comanda alla Board di spostare il pezzo** e invia un feedback positivo alla GameGUI. Infine, **l'interfaccia conclude il ciclo** invocando **drawChessBoard** per ridisegnare la scacchiera aggiornata e mostrare il nuovo stato del gioco all'utente.

MODELLAZIONE UML: 3.4 ACTIVITY DIAGRAM

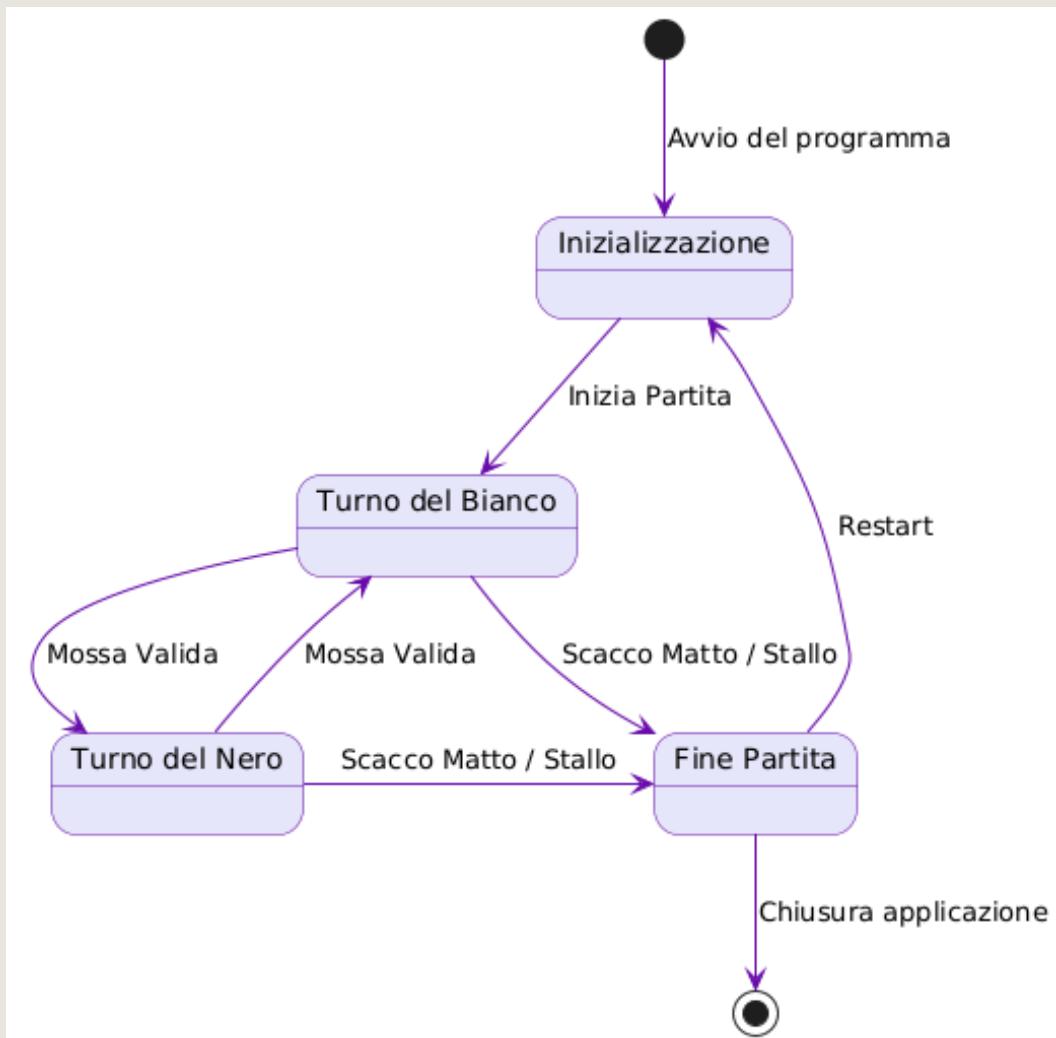


L'activity mostrato modella il flusso decisionale che si verifica a ogni interazione del Giocatore con la scacchiera.

Il processo è diviso in due macro-fasi:

- Selezione:** Il sistema verifica se il click dell'utente mira a selezionare un pezzo del turno corrente. In caso positivo, le mosse legali vengono immediatamente evidenziate.
- Movimento e Validazione:** Se un pezzo è già attivo, il sistema richiede l'esecuzione della mossa. Qui avviene il controllo cruciale di Validità (conformità alle regole), la gestione delle mosse speciali (Arrocco/En Passant) e il passaggio del turno. Infine, viene eseguita la verifica di fine partita; in caso positivo, il sistema termina il gioco e ferma i timer.

MODELLAZIONE UML: 3.5 STATE MACHINE DIAGRAM



Lo state machine mostrato modella l'evoluzione temporale della partita attraverso quattro stati principali:

1. **Inizializzazione:** È lo stato di partenza in cui il sistema configura la scacchiera e l'interfaccia grafica.
2. **Ciclo dei Turni:** Il cuore del gioco è l'alternanza continua tra Turno del Bianco e Turno del Nero. Il passaggio da un turno all'altro avviene esclusivamente tramite l'evento "Mossa Valida".
3. **Fine Partita:** Il ciclo si interrompe quando una mossa determina Scacco Matto o Stallo, portando il sistema nello stato conclusivo di vittoria o patta.
4. **Restart:** Dallo stato finale, l'utente può ricominciare il gioco, avviando una nuova sessione.

ARCHITETTURA SOFTWARE: SINGLETON

Ho scelto di applicare il pattern Singleton alla classe GameLogic per garantire che in tutto il ciclo di vita dell'applicazione esista una e una sola istanza del motore di gioco.

Negli scacchi è fondamentale avere un unico arbitro che gestisca i turni e lo stato della scacchiera; la creazione di più istanze porterebbe a conflitti di dati e a una gestione incoerente della partita.

```
public class GameLogic {  
    private static GameLogic instance;  
    private Board board;  
    private boolean isWhiteTurn;  
  
    private GameLogic() {  
        this.board = new Board();  
        this.isWhiteTurn = true;  
        //...  
    }  
  
    public static GameLogic getInstance() {  
        if (instance == null) {  
            instance = new GameLogic();  
        }  
        return instance;  
    }  
    //...  
}
```

ARCHITETTURA SOFTWARE: FACTORY

Ho scelto di adottare il pattern Factory Method per la gestione dei pezzi poiché permette di separare la logica di creazione degli oggetti dalla logica principale del gioco.

In questo modo, la classe Board o GameLogic non deve conoscere i dettagli costruttivi di ogni singolo pezzo (come il caricamento delle texture), ma si limita a richiederne la creazione a una classe specializzata.

PieceFactory è l'interfaccia che stabilisce il metodo standard per la creazione di un pezzo, agendo come un contratto per il sistema.

StandardPieceFactory è la classe operativa che traduce questo contratto in pratica, utilizzando uno switch per istanziare fisicamente il pezzo corretto in base al nome ricevuto.

La classe astratta **Piece** definisce le fondamenta comuni a tutti gli oggetti della scacchiera, come il colore e la posizione, mentre una classe concreta come **Rook** rappresenta il singolo pezzo specifico, contenendo al suo interno le regole di movimento uniche che lo distinguono dagli altri.

```
public interface PieceFactory {
    Piece createPiece(String type, ChessColor color, Position pos);
}

public class StandardPieceFactory implements PieceFactory {
    @Override
    public Piece createPiece(String type, ChessColor color, Position pos) {
        return switch (type.toLowerCase()) {
            case "pawn" -> new Pawn(color, pos);
            case "rook" -> new Rook(color, pos);
            case "knight" -> new Knight(color, pos);
            case "bishop" -> new Bishop(color, pos);
            case "queen" -> new Queen(color, pos);
            case "king" -> new King(color, pos);
            default -> throw new IllegalArgumentException("Tipo non accettato");
        };
    }
}
```

ARCHITETTURA SOFTWARE: FACTORY

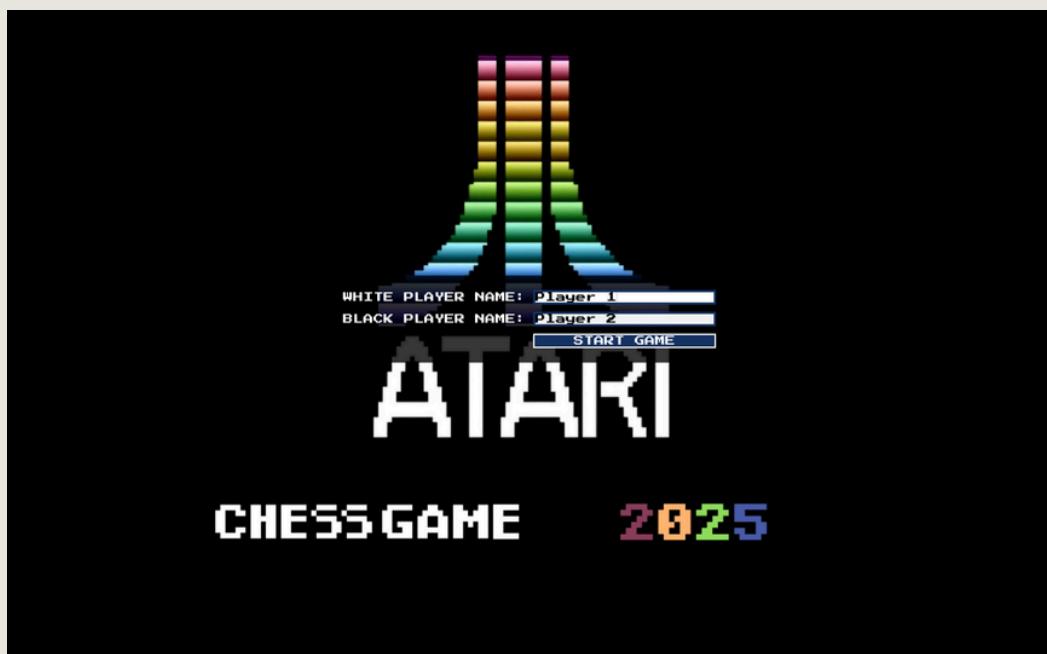
```
public abstract class Piece {  
    protected BufferedImage texture;  
    protected String name;  
    protected ChessColor color;  
    protected Position startPos;  
    protected boolean hasMoved;  
  
    public Piece(String name, ChessColor color, Position startPos) {  
        this.name = name;  
        this.color = color;  
        this.startPos = startPos;  
        this.hasMoved = false;  
    }  
  
    //questo è il metodo astratto che ogni pezzo lo implementerà in modo diverso  
    public abstract List<Position> getValidPositions();  
  
    public Position getPosition() { return position; }  
    public void setPosition(Position newPos) { this.position = newPos; }  
    public ChessColor getColor() { return color; }  
}
```

```
public class Rook extends Piece {  
  
    public Rook(ChessColor color, Position startPos) {  
        super("Rook", color, startPos);  
    }  
  
    @Override  
    public List<Position> getValidPositions() {  
        List<Position> validPositions = new ArrayList<>();  
  
        //... logica per calcolare le mosse in riga e colonna  
        //... restituita una lista di oggetti Position  
  
        return validPositions;  
    }  
}
```

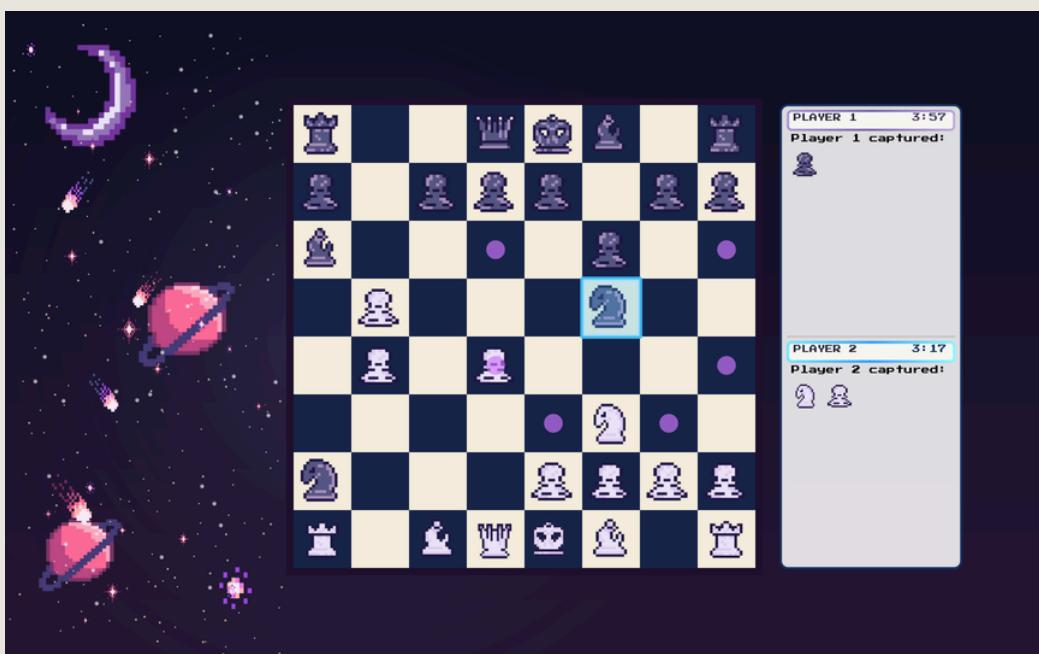
CONCLUSIONE

Il focus principale non è stato solo far funzionare il gioco, ma curarne l'architettura e la progettazione integrando l'uso dei diagrammi UML per modellare il sistema e dei design pattern.

Chiudo la tesina presentando alcune schermate del gioco, (gioco disponibile anche su GitHub), per offrire una visione più chiara.



SCHERMATA INIZIALE



PARTITA IN CORSO

CONCLUSIONE



SCHERMATA DI VINCITA



SCHERMATA DI PAREGGIO