Documento di Design

1 – Domain Class Model

Dopo una analisi degli Use Case è stata effettuata la loro mappatura in un insieme di classi che descrivono il mondo reale da modellare: il domain model del progetto.

Analizzando gli attributi e le specifiche richieste sono stati creati i seguenti oggetti allo scopo di rappresentare gli elementi e le componenti principali del Klotski.

A seguire una descrizione testuale degli oggetti mappati e delle relazioni presenti tra loro:

- **PLAYER:** l'oggetto *Player* ha lo scopo di modellare l'utente che gioca al gioco del Klotski. Esso può effettuare una o più partite all'interno delle quali può effettuare un insieme di mosse che modificano la struttura e la composizione della griglia di gioco. Queste mosse spostano tra loro i vari blocchi nelle posizioni libere a disposizione.

  (Questo oggetto è stato rimosso nella fase di definizione del class model)

- **MATCH:** la classe *Match* rappresenta una partita giocata dall'utente. Il suo scopo è quello di collezionare l'insieme delle mosse effettuate dal giocatore. Inoltre, una partita viene giocata su una griglia che contiene l'insieme dei blocchi da spostare e che caratterizzano il gioco Klotski (tali oggetti verranno approfonditi successivamente).

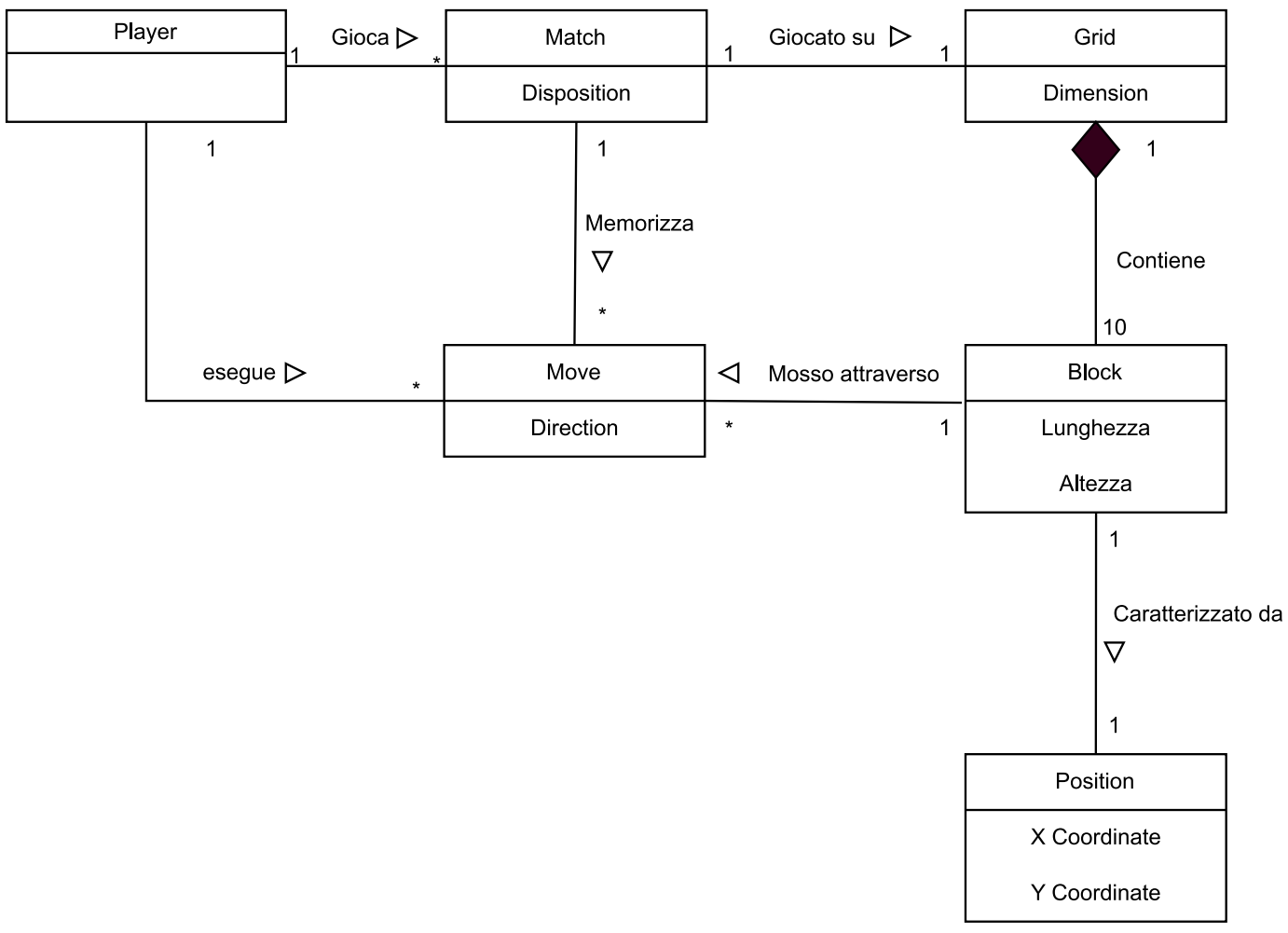
  Ogni partita inizia con una disposizione di partenza: rappresenta un posizionamento preciso dei blocchi all'interno della griglia con cui partire per giocare.

- **GRID:** tale classe è l'astrazione dell'oggetto reale che, al suo interno, contiene i blocchi che possono scorrere nelle posizioni libere della griglia. Il suo scopo è quello di mantenere lo stato del gioco; inoltre, permette di capire se le mosse effettuate del giocatore siano ammissibili (possibilità di spostare il blocco nella posizione desiderata) o non possibili (il blocco è vincolato dalla presenza di altri a lui limitrofi). Infine, la griglia fornisce informazioni sullo stato della partita che può essere in corso o vinta (il blocco target ha raggiunto la posizione della vittoria).

- **BLOCK:** i blocchi rappresentano gli elementi alla base del gioco del Klotski. Essi possono essere tra loro spostati allo scopo di far raggiungere al blocco target la posizione di uscita per la vittoria. I blocchi sono raffigurati da delle forme geometriche ben precise: rettangoli e quadrati, ognuno dei quali possiede delle specifiche dimensioni, un'altezza e una larghezza. Ogni blocco possiede una posizione che rappresenta le coordinate dell'angolo in alto a sinistra della forma geometrica.

- **POSITION:** è l'elemento di base che rappresenta la modellazione di una posizione di un oggetto in uno spazio bidimensionale. Esso è una rappresentazione di un punto in un sistema cartesiano avente a disposizione un'ascissa e una ordinata.

- **MOVE:** rappresentano le mosse effettuate da un giocatore. Una mossa ha lo scopo di cambiare la posizione di un blocco facendolo scorrere negli spazi liberi. La mossa possiede una direzione che congiunge la posizione precedente del blocco spostato con la attuale. L'insieme delle mosse definisce lo storico delle azioni effettuate dal giocatore.



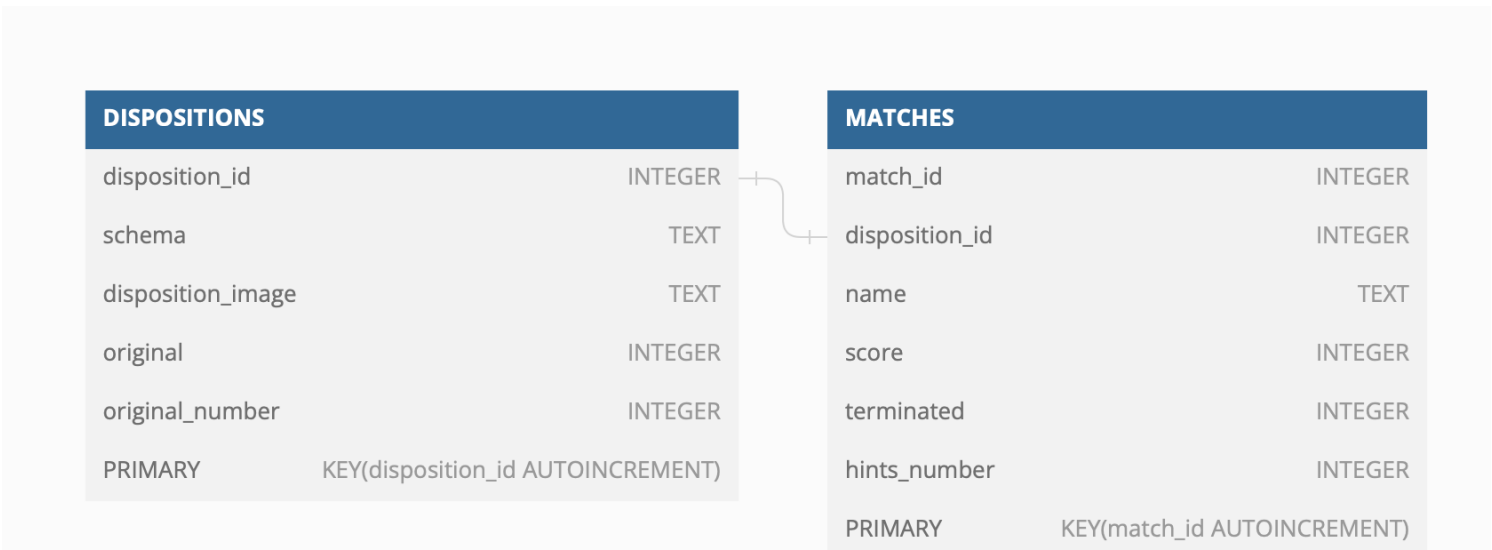
2 – Database Schema

La necessità di mantenere un salvataggio delle partite e l'insieme delle configurazioni di avvio del gioco ha portato alla decisione di utilizzare un semplice database che contenesse tutte le informazioni significative.

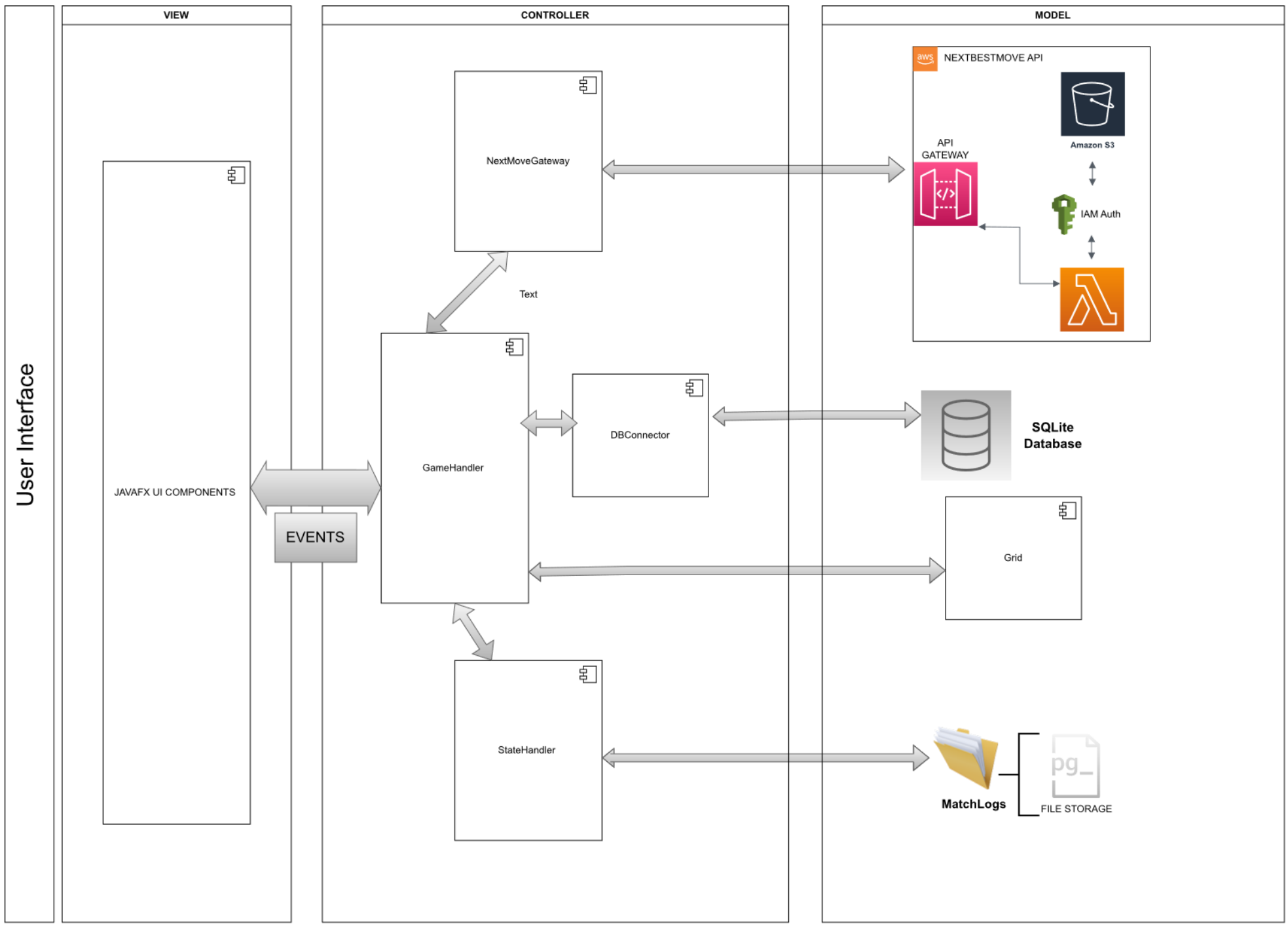
A tale scopo abbiamo scelto di utilizzare un database locale comune: SQLite abbinato all'uso di file di log delle partite per mantenere lo storico delle azioni (cioè le mosse) effettuate.

Il database schema costruito per il progetto è composto da due tabelle:

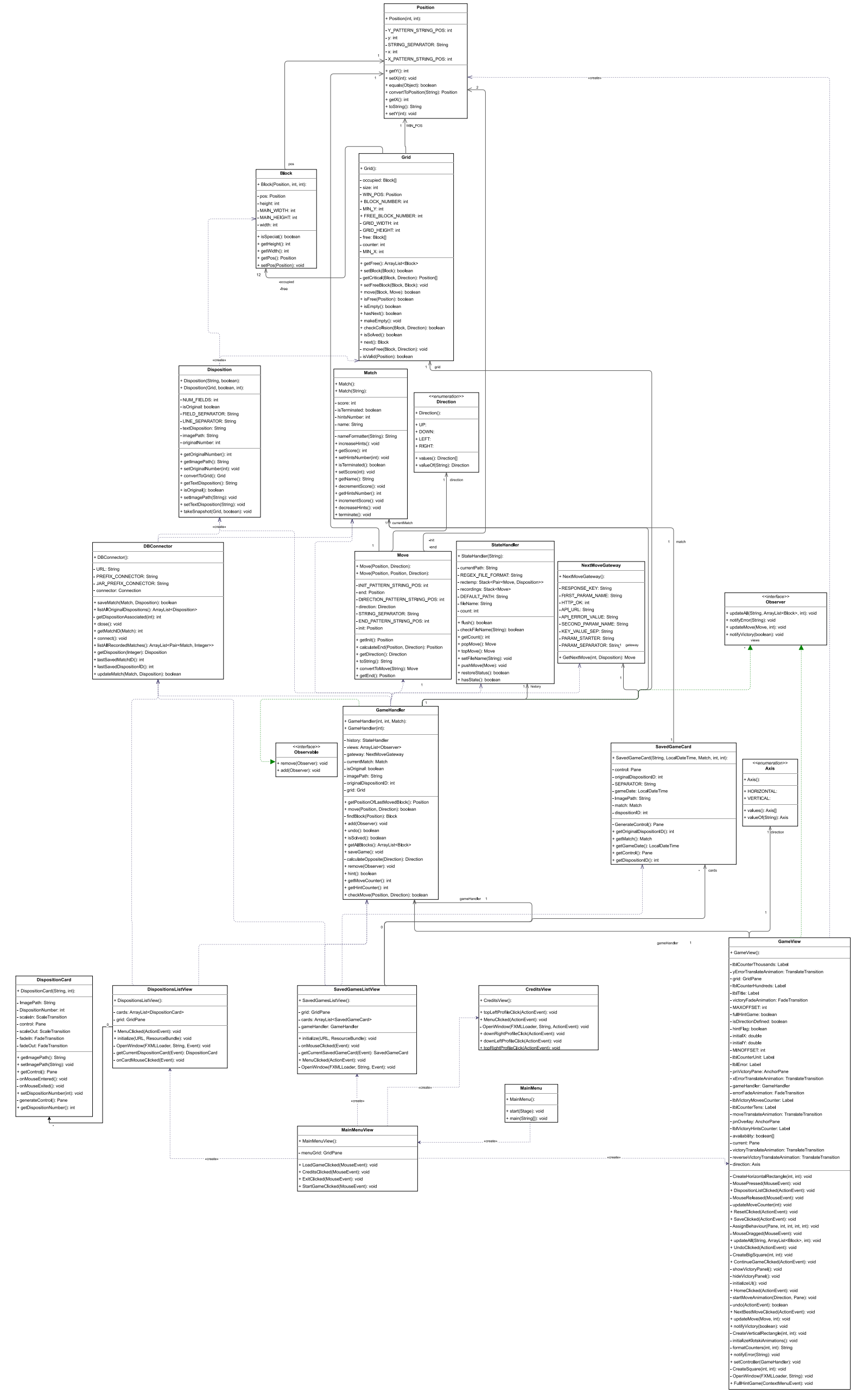
- **DISPOSITIONS:** questa tabella mantiene le informazioni delle configurazioni di gioco del Klotski. Possono essere di due tipi: **originali** se rappresentano delle configurazioni con cui è possibile iniziare a giocare o **non originali** se sono disposizioni create dall'utente durante una partita. A tale scopo è stato inserito il campo "*original*" che assume il valore 1 se del primo tipo altrimenti 0 *(non è stato possibile usare un booleano a causa dell'insieme di tipi di dato che SQLite mette a disposizione).*
  Il campo "*schema*" rappresenta la codifica su stringa della configurazione secondo il formato: **`block_heigth-block_width-block_position# .`**
  Il campo *"original_number"* contiene un valore che viene impostato se la disposizione non è originale, allo scopo di identificare la configurazione di partenza dalla quale si ha iniziato a giocare.
  Il campo "*disposition_image*" contiene il percorso relativo all'immagine che rappresenta graficamente le configurazioni, sia originali che salvate.
  Infine la chiave primaria della tabella è un valore intero auto incrementato gestito dal DBMS.
- **MATCHES:** è una tabella che ha lo scopo di mantenere informazioni legate ad una partita salvata. I campi "*name*", "*score*", "*terminated*" e "*hints_number*" rappresentano rispettivamente il nome della partita (che coincide con la data di salvataggio), il numero di mosse effettuate, un valore che indica se la partita è **in corso** o **terminata** e il numero di suggerimenti utilizzati.
  Il campo "*disposition_id*" è un riferimento alla tabella **DISPOSITIONS** e rappresenta l'ID della configurazione dell'ultima mossa effettuata prima del salvataggio.
  Anche in tal caso la chiave primaria della tabella è un valore intero auto incrementato gestito dal DBMS.



3 – Sequence Diagram



4 - Class Model



Documento di Design

ARCHITECTURAL PATTERN

In questo progetto è stato utilizzato il pattern architetturale **MVC** (*Model View Controller*). Questo pattern è ampiamente utilizzato per lo sviluppo di applicazioni desktop e applicazioni web poiché consente di aumentare **l'astrazione** presente tra i componenti dell'applicazione stessa.

Data la longevità di MVC (è uno dei primi design pattern ad essere stato ideato), del suddetto pattern sono nate varie "sfumature" che si differenziano principalmente per la presenza (e tipo) di comunicazioni tra *View* e *Model*. Difatti, il pattern MVC in sé prevede la presenza dei tre componenti da cui deve il nome, ma non standardizza in alcun modo la comunicazione che ci deve essere tra essi.
In questo progetto, la *View* e il *Model* non interagiscono mai direttamente, ma è il *Controller* a fare da intermediario.

Di seguito, i ruoli degli oggetti presentati nel Design Class Model:

- View: *MainMenuView, CreditView, DispositionsListView, SavedGamesListView, GameView*;
- Controller: *GameHandler, StateHandler, NextMoveGateway*;
- Model: Block, *Direction, Disposition, Grid, Match, Move, Position*;

GANG OF FOUR PATTERNS

- ***Factory***
  Il pattern *Factory* è stato applicato per la classe *Disposition*. *Disposition*, infatti, permette di creare una *Grid* a partire da una stringa che rappresenta la disposizione dei blocchi nella griglia. In questo modo ogni qualvolta sia necessario creare una *Grid* basta usare la classe *Disposition* che funge da *Factory* e crea la *Grid* desiderata. Inoltre, creando la *Grid* usando una classe dedicata, si nasconde la logica di inizializzazione a partire da una stringa.

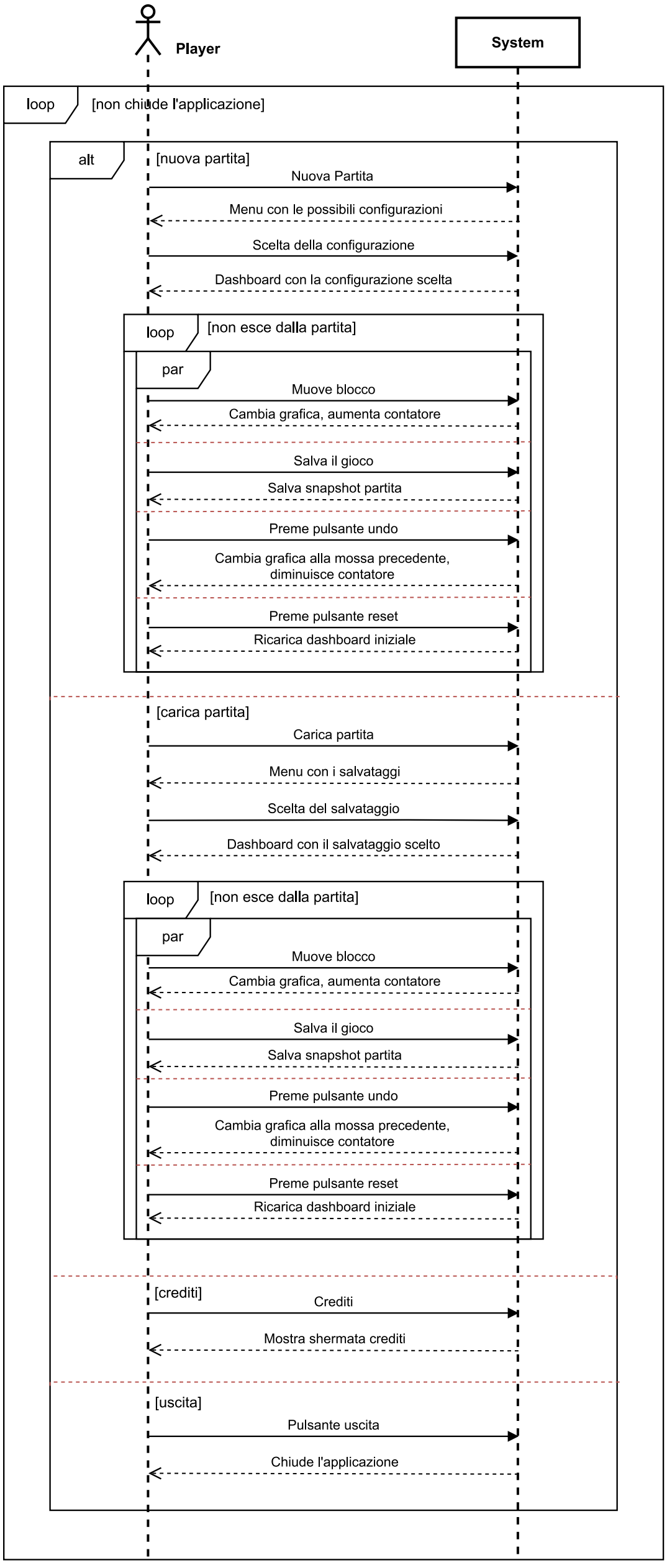
- ***Adapter***
  Il pattern *Adapter* è stato applicato per la classe *DBConnector*. Infatti questa classe offre un'interfaccia per interagire con il database, dove sono salvate le partite precedentemente effettuate. A differenza dei *Controller, l'Adapter* interagisce con un sistema che era già esistente e sfrutta un'interfaccia già esistente per eseguire le azioni richieste.

- ***Façade***
  Il pattern *Façade* è stato applicato per le classi *GameHandler, StateHandler* e *NextMoveGateway*. In particolare, poiché sono stati utilizzati più controller, il termine corretto per ognuno di questi è Use Case Controller. Queste classi rappresentano un'interfaccia e permettono alla *View* di comunicare con il *Model*. Nello specifico, a seguito di un'azione eseguita dall'utente e ricevuta dalla *View*, quest'ultima si affida al *Controller* adatto per eseguire l'azione richiesta.

- ***Command***
  Il pattern *Command* è stato applicato per la classe *Move*. *Move*, infatti, rappresenta una mossa effettuata dal giocatore nel gioco del Klotski. È stata creata una classe specifica per rappresentare una mossa poiché in questo modo è possibile supportare l'operazione di Undo e inoltre è possibile serializzare le mosse per salvarle in un file di persistenza. Nel nostro caso la classe responsabile della gestione dei *Command* è *StateHandler* che può quindi essere considerato l'*Invoke*r. La classe che invece conosce come eseguire una *Move* è *Grid,* che può essere quindi considerata il *Receiver*.

- ***Observer***
  Il pattern *Observer* è stato applicato per le classi *GameHandler* e *GameView*. Difatti, il *Controller* (*GameHandler*), dopo aver comunicato con il *Model*, comunica alla *View* di aggiornarsi tramite le funzioni dell'interfaccia *Observable* (notify o update). Più *View* (*Observer*) possono registrarsi allo stesso *Observable*. Tra le "sfumature" sopra citate di MVC, il pattern *Observer* può essere utilizzato tra *Controller* e *View* (come in questa occasione) oppure direttamente tra *Model* e *View* (comodo quando una singola classe del *Model* ha tutti i dati necessari alla *View* corrente, non è questo il caso poiché i dati sono spezzati tra più classi del *Model* e unificati tramite il *Controller*).

5 – System Sequence Diagram



6 – Internal System Sequence Diagram