

Scotland Yard in Java

Marco De Grosvorskaja

September 7, 2023

Contents

1	Introduzione	3
1.1	Motivazione	3
1.2	Obbiettivo	3
1.3	Approccio e metodo	4
1.4	Esito	4
2	Analisi	5
2.1	Problem Statement	5
2.2	Analisi dei Requisiti - Interfaccia Utente	5
2.3	Analisi dei Requisiti - Logica del Gioco	6
3	Progettazione	7
3.1	Use Case Diagrams	7
3.2	Mockups	10
3.2.1	GameMenuView Mockup	10
3.2.2	TurnMenuView Mockup	10
4	Implementazione	11
4.1	Project Directory	12
4.2	Class Diagram - Menu	13
4.2.1	Package - menu.view	15
4.2.2	Package - menu.control	16
4.2.3	Package - menu.model	17
4.3	Class Diagram - Game	18
4.3.1	Package - game.model	20
4.3.2	Package - game.control	23
4.3.3	Package - game.view	25
4.4	Class Diagram - Command	26
4.4.1	Package command.game	29
4.4.2	Package command.turn	29
4.5	Class Diagram - Utils	30
4.6	Design	31
4.6.1	MVC Design	31
4.6.2	Friendship Mechanism	34
4.6.3	Debugger	35
4.7	Command Design Pattern	36

4.7.1	Generic Command Pattern	36
4.7.2	Implemented Command Pattern	37
4.7.3	Singleton Design Pattern	40
5	Testing	41
6	Demo	42

Chapter 1

Introduzione

1.1 Motivazione

L'idea di ricreare il gioco da tavolo Scotland Yard come progetto per l'esame di Ingegneria del Software è nata un fine settimana durante una partita in famiglia.

Era da diversi giorni che ponderavo su quale sarebbe potuto essere un problema che richiedesse o per il quale fosse ragionevole l'uso di un paradigma di programmazione ad oggetti, specialmente per un linguaggio così poco permissivo come Java.

La scelta del gioco Scotland Yard appariva azzeccata, questo presentava una struttura ben definita dato che i suoi requisiti, il suo scopo e la sua logica erano noti a priori ed era credibile che ciò avrebbe compensato l'inflessibilità del linguaggio Java per ciò che concerne la prototipazione e l'elasticità nel cambiamento dei requisiti.

Scotland Yard, permetteva, inoltre, una diretta correlazione tra gli oggetti di gioco e l'astrazione fornita dal paradigma di programmazione ad oggetti che proprio per simili scenari era stata pensata.

Per ciò che concerneva gli scopi dell'esame di Ingegneria del Software, sembrava plausibile ed intravedibile la necessità dell'uso di Design Patterns ed apparivano ben formabili i documenti e gli schemi visti a lezione.

1.2 Obbiettivo

Dopo quasi un decennio di esperienza mi era chiaro che l'assoluta compartimentazione e frammentazione delle funzioni logiche e degli attributi in classi, persegue sì l'obbiettivo di incapsulamento e definizione delle responsabilità, ma porta con sé il peso di una pre-ingegnerizzazione e sovra-ingegnerizzazione che ne determinino i riferimenti, le composizioni e le interazioni.

Diventa dunque di fondamentale importanza, ridurre al minimo la formazione di *cross-references* e la compartimentalizzazione del codice in *packages* il più possibile indipendenti.

1.3 Approccio e metodo

Questa relazione esplorerà come è stato deciso di condurre la compartmentalizzazione delle responsabilità attraverso la scomposizione del macro sistema Scotland Yard in due sotto-sistemi indipendenti: *menù* e *gioco*, ciascuno dei quali adotterà un pattern *MVC* (*Model*, *View*, *Controller*) che partizionerà il sotto-sistema in tre funzioni logiche distinte che opereranno tra loro secondo uno schema definito a priori.

Si vedrà inoltre come si è fatto uso di una tecnica specifica del linguaggio di programmazione Java per la definizione di *chiavi d'accesso* (*access keys*) che ci permetterà di definire metodi all'interno di una classe che siano accessibili soltanto al detenente della chiave di accesso.

1.4 Esito

Sarà evidente continuando nella lettura della relazione che si è riusciti ad ottenere ottimi risultati per ciò riguarda la compartmentalizzazione dei comportamenti e delle associazioni tra classi.

Si è riusciti ad ottenere una netta divisione tra il componente **Game** e il componente **Menu**, essi coesistono indipendentemente l'uno dall'altro: il componente **Game** detiene una singola istanza di **Menu** e le iterazioni tra di loro vengono gestite dal *Design Pattern* **Command**, che costituisce l'insieme di *Callbacks* che invocano l'esecuzione dei metodi di gioco.

Si è riusciti ad ottenere una netta divisione tra i ruoli di **Model**, **View** e **Control** all'interno di ciascun componente.

Si è riusciti ad ottenere garanzie sui privilegi di accesso a metodi attraverso l'implementazione di **AccessKeys**.

Si è riusciti ad ottenere una classe specifica atta al **Debugging** dell'applicazione così da avere alterazioni controllate e limitate ad una singola classe.

Si è riusciti a creare facilmente una classe di **Testing** che ha dimostrato le capacità del progetto, nella sua separazione dei ruoli e dei privilegi, ad adattarsi a condizionamenti esterni.

Chapter 2

Analisi

2.1 Problem Statement

Si intende realizzare una versione digitale del famoso gioco da tavolo Scotland Yard.

Questa dovrà avere una interfaccia grafica, con un menu di gioco e con una vista principale che mostri a schermo la mappa di gioco e permetta l'iterazione con il giocatore, attraverso dei bottoni a schermo.

L'implementazione dovrà realizzare il *core* di gioco, una estensione di questo per permettere partite multi-giocatore o contro il computer, non rientra nell'ambito di questo lavoro.

2.2 Analisi dei Requisiti - Interfaccia Utente

La seguente lista di punti definisce le caratteristiche, il design e le interazioni dell'interfaccia utente che poi dovranno realizzarsi in un programma software.

- L' interfaccia utente si divide in due schermate principali:
 1. La schermata del menù di gioco.
 2. La schermata di gioco.
- La schermata del menù di gioco consiste in un titolo ("Scotland Yard") e nella seguente lista di bottoni: ["Nuova Partita", "Carica Partita", "Impostazioni", "Esci dal gioco"] che performino le relative azioni se cliccati.
- La schermata di gioco deve permettere di vedere il tabellone, di controllare la propria posizione, la posizione dei poliziotti, l'ultima posizione nota di Mr X, i propri biglietti, i biglietti dei poliziotti e infine deve permettere di uscire e di tornare al menù di gioco.
- L' azione "Impostazioni" deve permettere di modificare le impostazioni di gioco.

2.3 Analisi dei Requisiti - Logica del Gioco

La seguente lista di punti definisce le regole, le caratteristiche e più in generale, la logica del gioco che poi dovranno realizzarsi in un programma software.

Definizione degli oggetti e attori di gioco:

- Possono giocare da 3 a 6 giocatori (inclusi), un giocatore è Mr X, gli altri sono Poliziotti.
- Ci sono 5 tipi di biglietti diversi (taxi, bus, metropolitana, black, mossa doppia).

Preparazione del gioco:

- Mr X riceve 4 biglietti per il taxi, 3 per il bus, 3 per la metropolitana, 2 carte per la mossa doppia e tanti *black* tickets quanto il numero di poliziotti.

Ogni poliziotto riceve invece 10 biglietti per il taxi, 8 per l' autobus, 4 per la metropolitana.

- Ogni giocatore riceve una posizione di partenza random tra la seguente lista:
[13, 26, 29, 34, 50, 53, 91, 94, 103, 112, 117, 132, 138, 141, 155, 174, 197, 198].

Svolgimento del gioco:

- Inizia Mr X, che individua i possibili spostamenti dalla posizione corrente, paga il biglietto corrispettivo, si segna il numero della stazione nella quale si è spostato e finisce il turno.
- A turno ogni poliziotto si muove, individuando i possibili spostamenti dalla posizione corrente, pagando il biglietto corrispettivo e spostando la pedina sulla stazione sulla quale si sono spostati.

I biglietti usati dai poliziotti passano a Mr X.

- Al turno [3, 8, 13, 18] Mr X dovrà rivelare ai poliziotti il suo nascondiglio.

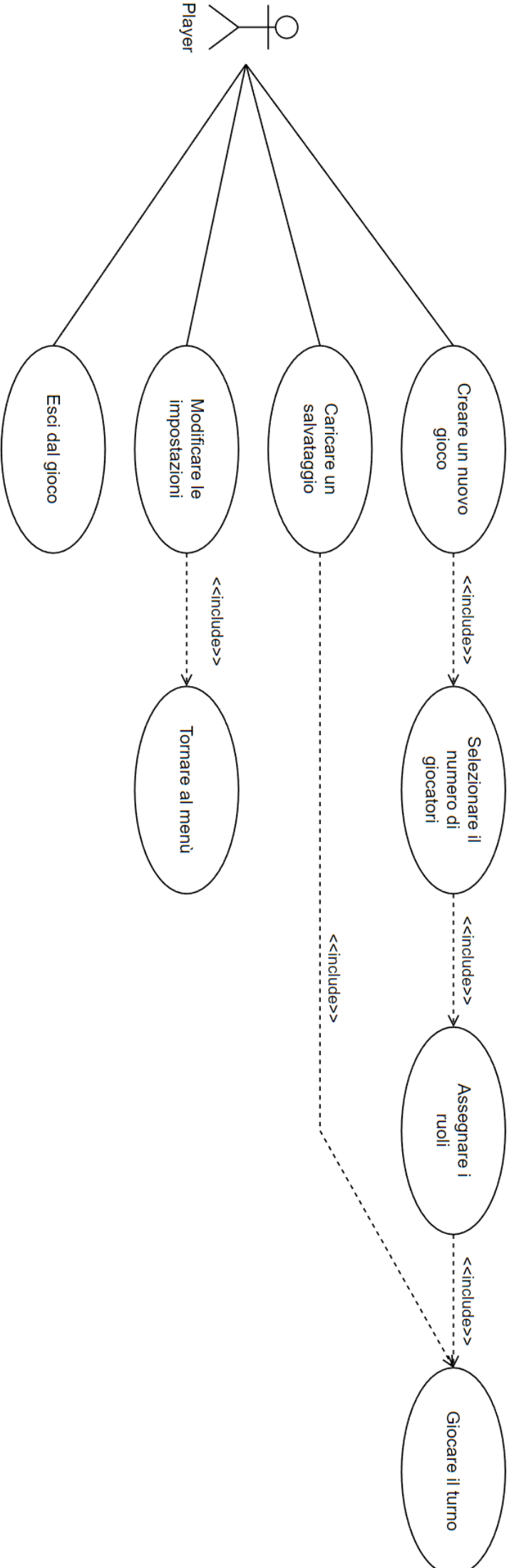
Fine della partita:

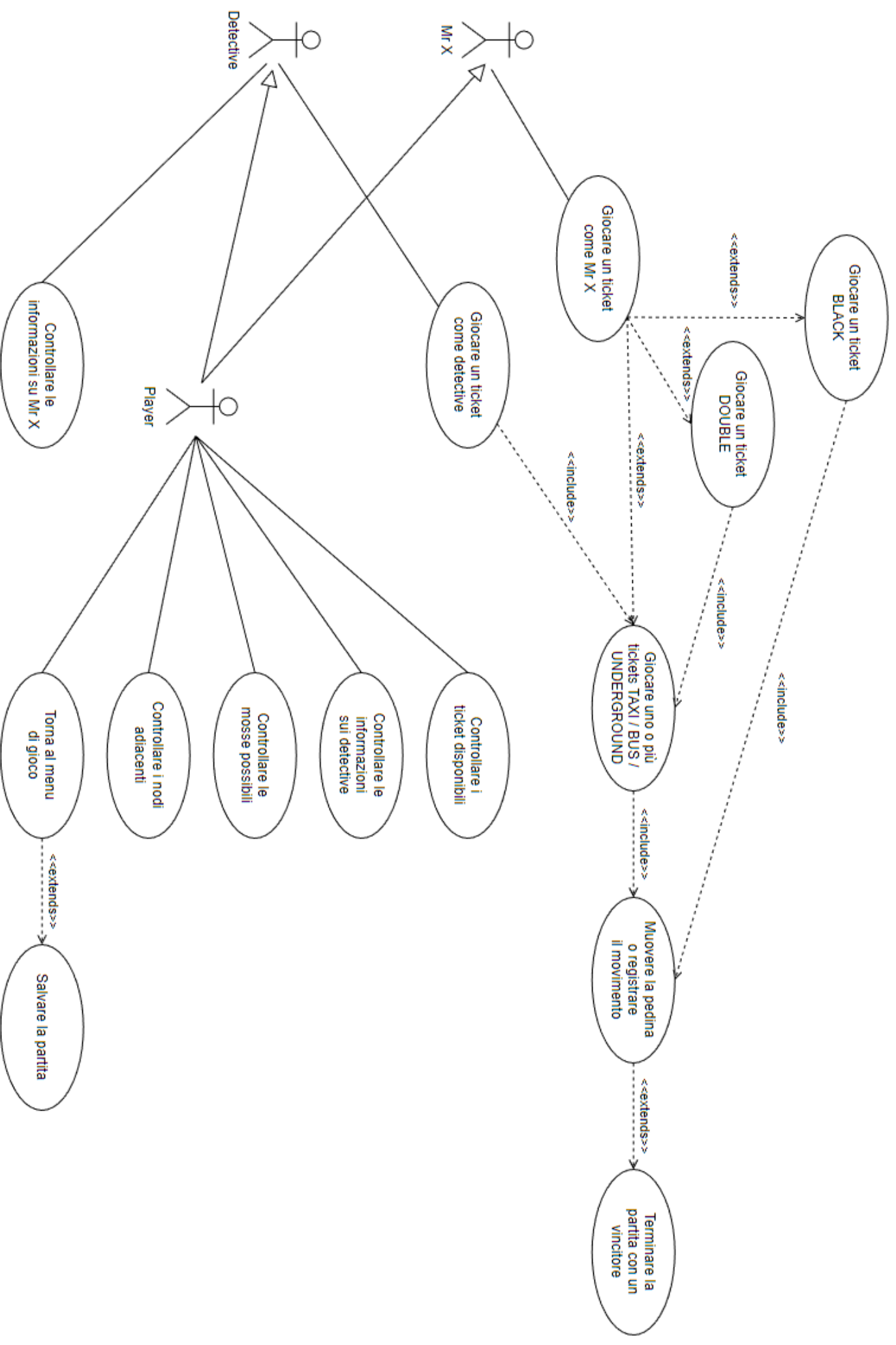
- Vince Mr X se:
 - a) I poliziotti non hanno più biglietti per continuare l'inseguimento.
 - b) Mr X conclude il 22-esimo turno.
- Vincono i poliziotti se: uno di loro riesce a catturare Mr x nella stazione in cui si era nascosto.

Chapter 3

Progettazione

3.1 Use Case Diagrams



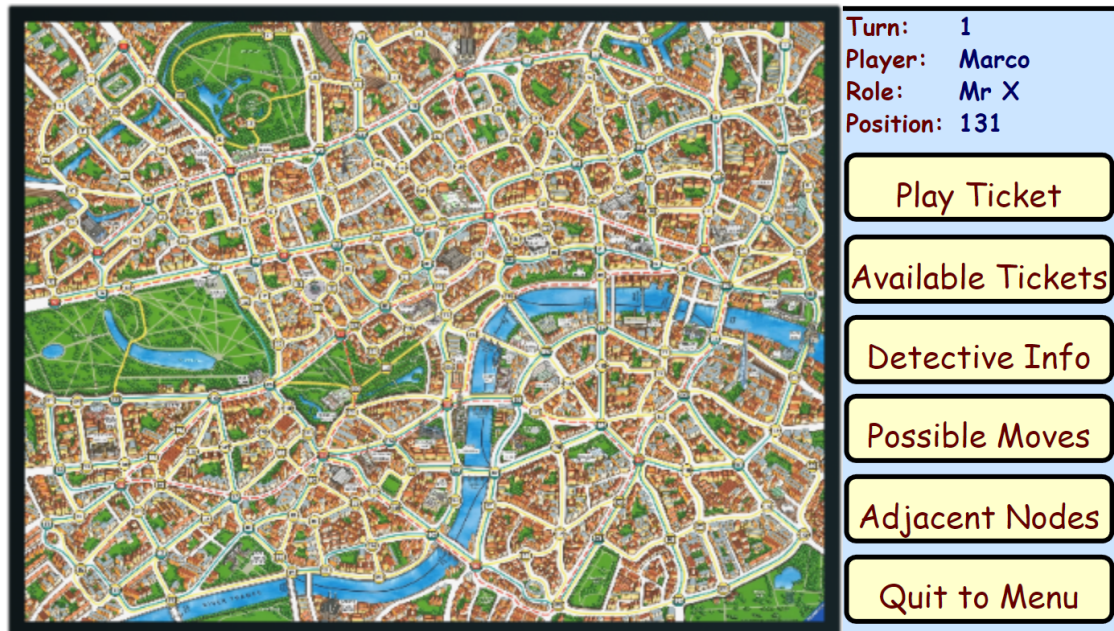


3.2 Mockups

3.2.1 GameMenuView Mockup




















































3.2.2 TurnMenuView Mockup



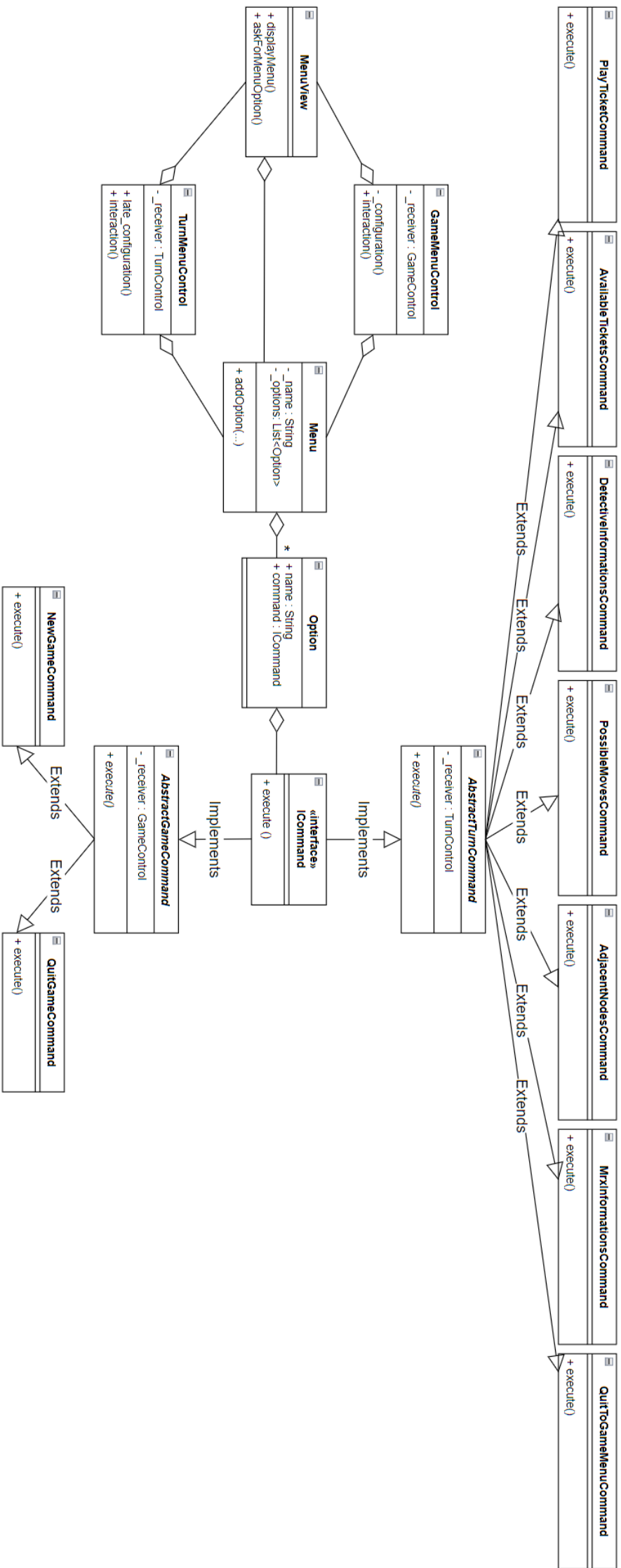
Chapter 4

Implementazione

4.1 Project Directory

- ▼  src
 - ▼  sy
 - >  Debugger.java
 - >  ScotlandYard.java
 - >  Settings.java
 - ▼  sy.command
 - >  AbstractGameCommand.java
 - >  AbstractTurnCommand.java
 - >  ICommand.java
 - ▼  sy.command.game
 - >  NewGameCommand.java
 - >  QuitGameCommand.java
 - ▼  sy.command.turn
 - >  AdjacentNodesCommand.java
 - >  AvailableTicketsCommand.java
 - >  DetectiveInformationsCommand.java
 - >  MrxInformationsCommand.java
 - >  PlayTicketCommand.java
 - >  PossibleMovesCommand.java
 - >  QuitToGameMenuCommand.java
 - ▼  sy.game.control
 - >  GameControl.java
 - >  TurnControl.java
 - ▼  sy.game.model
 - >  Board.java
 - >  Detective.java
 - >  Game.java
 - >  Mrx.java
 - >  Player.java
 - >  Role.java
 - >  StarterCards.java
 - >  State.java
 - >  Ticket.java
 - >  Turn.java
 - ▼  sy.game.view
 - >  GameView.java
 - >  TurnView.java
 - ▼  sy.menu.control
 - >  GameMenuControl.java
 - >  TurnMenuControl.java
- ▼  sy.menu.model
 - >  Menu.java
 - >  Option.java
- ▼  sy.menu.view
 - >  MenuView.java
- ▼  sy.utils
 - >  BoardLoader.java
 - >  CLI.java
 - >  Dialog.java
 - >  StringParser.java
- ▼  test
 - >  StateTest.java
- >  JUnit 5
- ▼  rsc
 - >  Board.txt

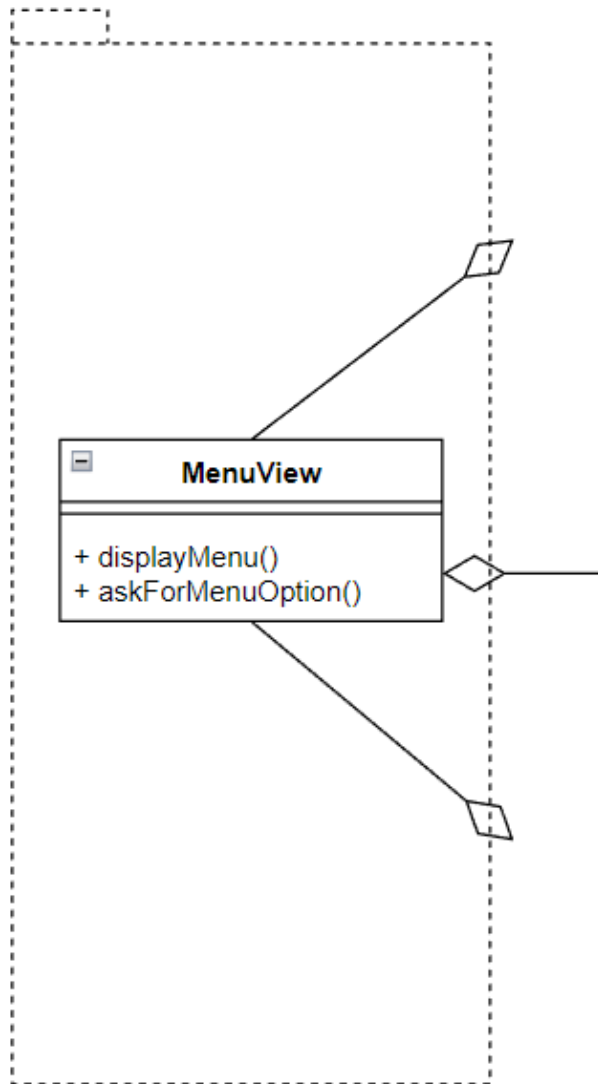
4.2 Class Diagram - Menu



4.2.1 Package - menu.view

Il package *menu.view* contiene una classe: ***MenuView***.

Questa si occupa di mostrare a schermo il menu e di chiedere all'utente di selezionarne una opzione.

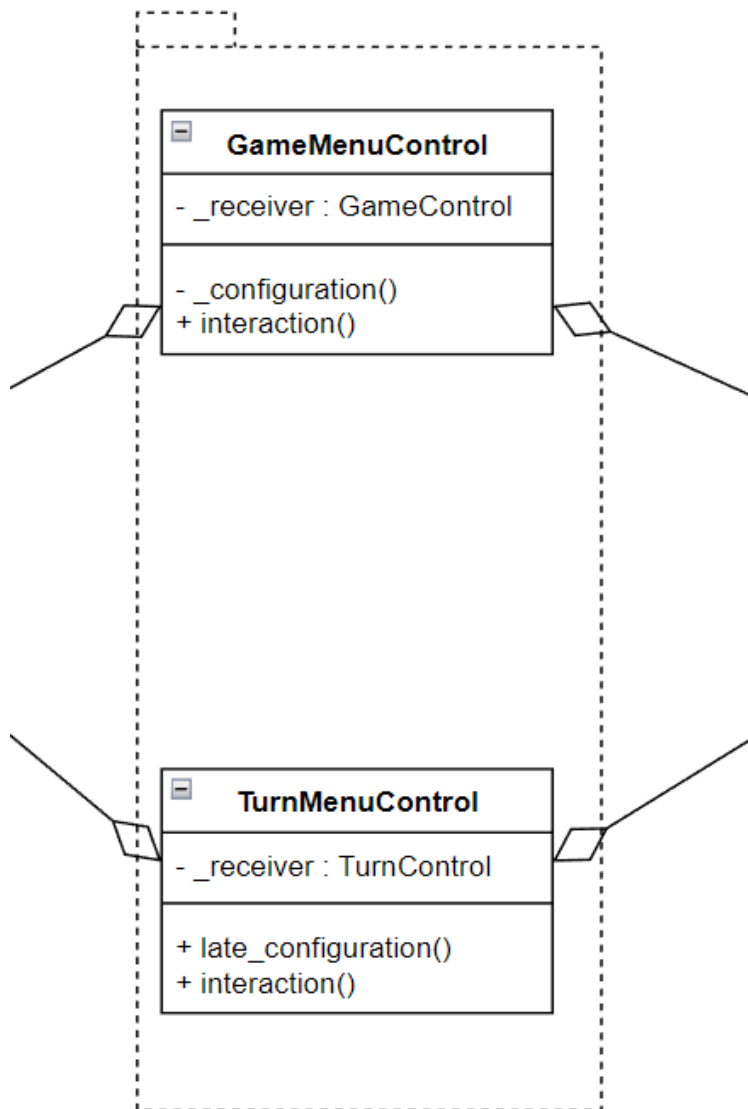


4.2.2 Package - menu.control

Il package *menu.control* contiene 2 classi: ***GameMenuControl*** e ***TurnMenuControl***.

Queste si occupano della creazione di un modello ***Menu***, della creazione di una vista ***MenuView*** e dell' associazione di ogni opzione (***Option***) del menu al suo comando (per esempio ***PlayTicketCommand***).

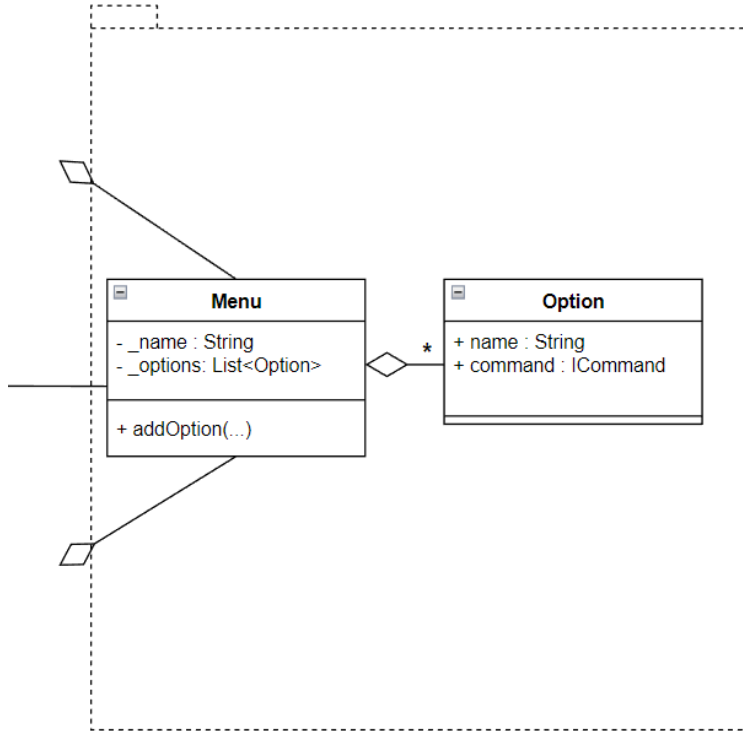
Infine espongono un metodo pubblico ***interaction()*** per chiamare la messa a schermo del menu, la richiesta di un input da parte dell' utente ed infine il comando associato all' opzione scelta dall'utente.



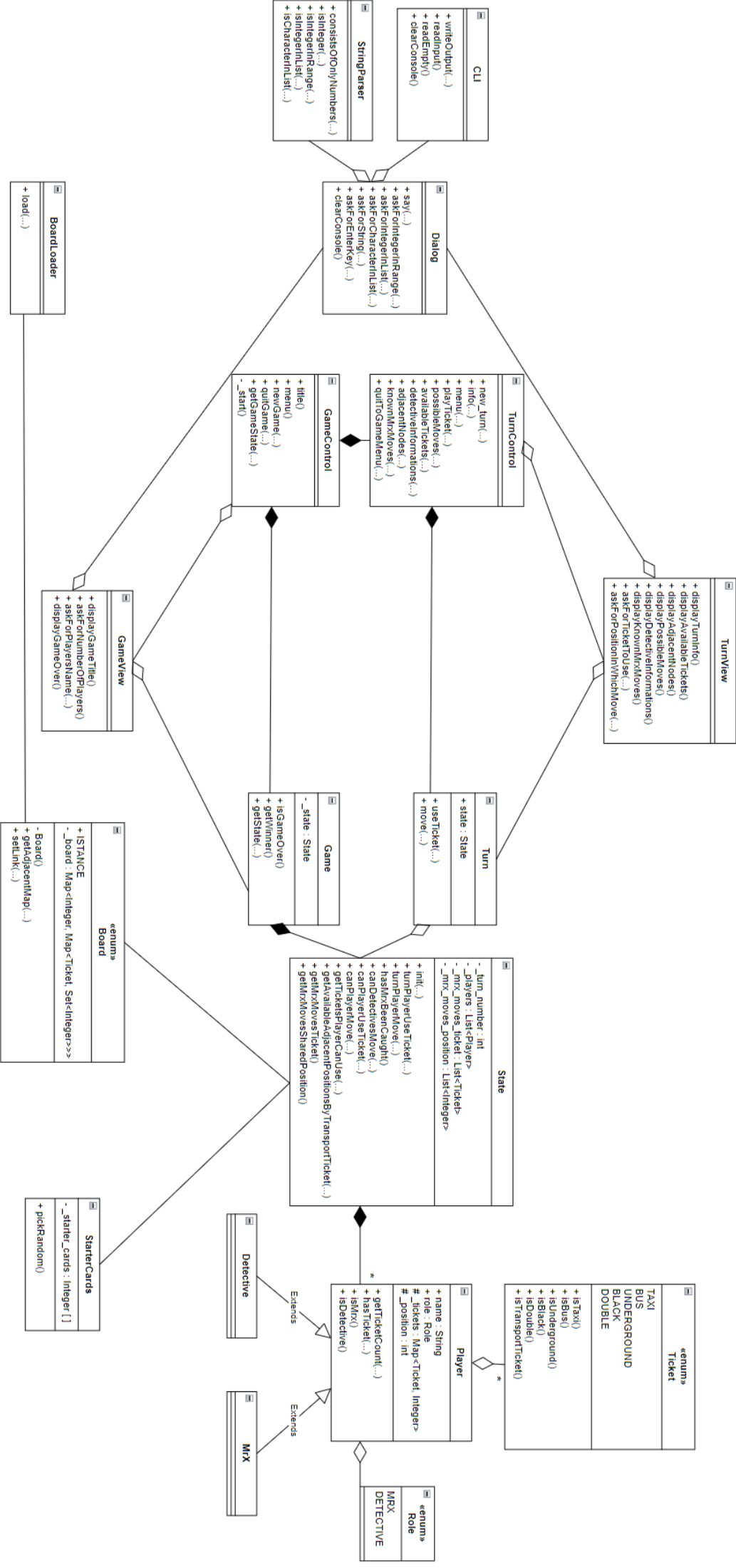
4.2.3 Package - menu.model

Il package *menu.model* contiene 2 classi: **Menu** e **Option**.

Menu specifica un nome e una lista di opzioni, mentre **Option** specifica un nome e un riferimento a un oggetto che implementi l'interfaccia **ICommand**.



4.3 Class Diagram - Game



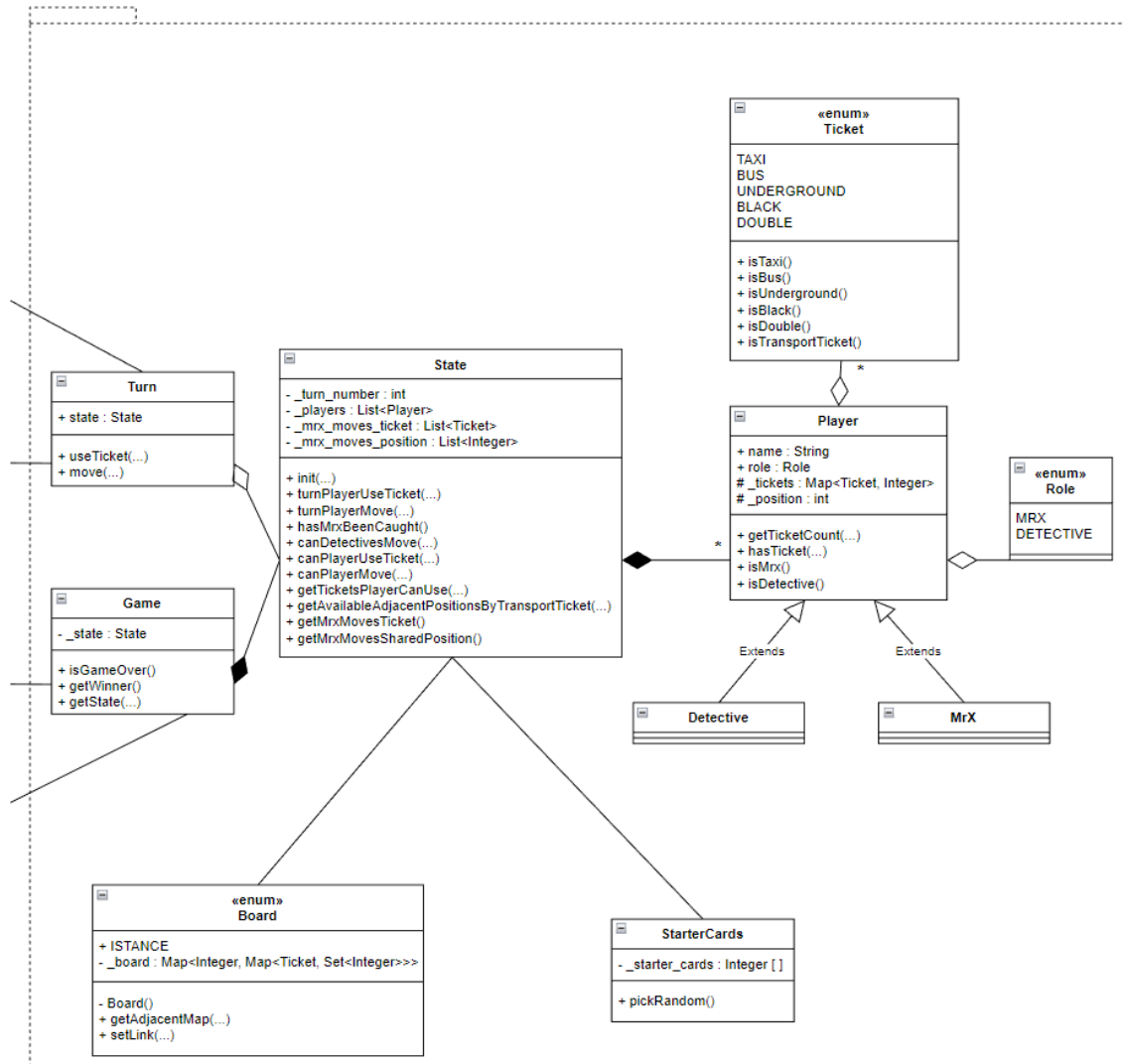
4.3.1 Package - game.model

Il package *game.model* contiene 10 classi.

Game e *Turn* sono i modelli associati ai rispettivi *Control* e *View*; entrambi possiedono un riferimento ad uno stesso oggetto di tipo *State*, ma con privilegi differenti:

- Solo un *Game* può costruire uno *State*.
- Solo un *Turn* può invocare alterazioni allo *State*.

State è la classe principale del modello di gioco, questa contiene lo stato del gioco, ovvero l'insieme di variabili che definiscono il gioco in corso e che si potrebbero modificare durante il suo svolgimento.



State:

Metodo	Privilegi ai metodi esposti da State
<i>State(...)</i>	Game
<i>init(...)</i>	Game
<i>turnPlayerUseTicket(...)</i>	Turn
<i>turnPlayerMove(...)</i>	Turn
<i>setTurnNumber(...)</i>	Turn
<i>getTurnPlayer(...)</i>	Turn
<i>getTurnNumber()</i>	: public
<i>getDetectives()</i>	: public
<i>hasMrxBeenCaught()</i>	: public
<i>canDetectivesMove()</i>	: public
<i>canPlayerMove(...)</i>	: public
<i>getTicketsPlayerCanUse(...)</i>	: public
<i>getAvailableAdjacentPositionsByTransportTicket(...)</i>	: public
<i>canPlayerUseTicket(...)</i>	: public
<i>getMrxMovesTicket()</i>	: public
<i>getMrxMovesSharedPosition()</i>	: public

Ticket e ***Role*** sono enumerazioni che identificano il tipo di biglietto e il ruolo del giocatore.

StarterCards è un classe che attraverso il metodo pubblico *pickRandom()* permette di ritornare randomicamente un intero identificante il nodo di partenza del giocatore a partire da una sequenza iniziale di possibili nodi definita in ***Settings***.

Board è un singleton che rappresenta la tabella di gioco, viene caricata attraverso la classe ***BoardLoader*** nel package *utils*.

Player è la classe che definisce il giocatore, può essere costruita solo da uno ***State***, questa si deriva in due sottoclassi ***MrX*** e ***Detective***.

Player:

Metodo	Privilegi d'accesso ai metodi esposti da Player
<i>Player(...)</i>	State
<i>setPosition(...)</i>	State
<i>setTicketCount(...)</i>	State
<i>getPosition()</i>	: public
<i>getTicketCount(...)</i>	: public
<i>hasTicket(...)</i>	: public
<i>isMrx()</i>	: public
<i>isDetective()</i>	: public

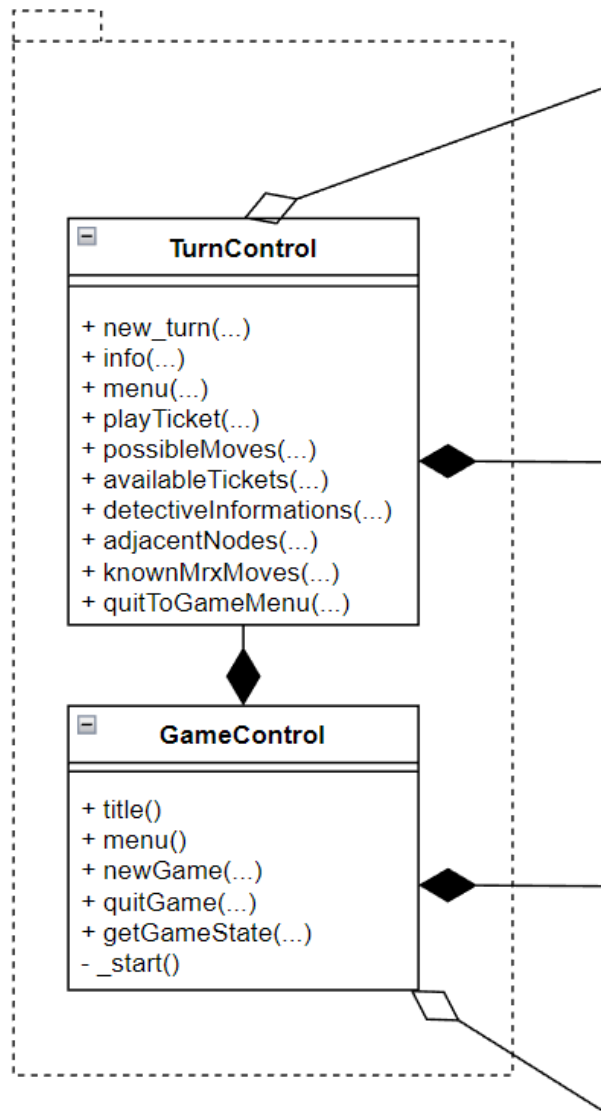
4.3.2 Package - game.control

Il package *game.control* contiene 2 classi: **GameControl** e **TurnControl**.

Queste si occupano di esporre la logica del gioco e del turno di gioco rispettivamente.

GameControl:

- è l' *entrypoint* per il gioco, questo espone un costruttore pubblico che a sua volta costruirà le seguenti classi: **Game**, **GameView**, **TurnControl**, **GameMenuControl**.
- è l'unica classe che possiede i privilegi per chiamare il costruttore di **Game** ed i suoi metodi *init(...)* e *getState(...)*.
- è l'unica classe che possiede i privilegi per chiamare il costruttore di **TurnControl** ed i suoi metodi *new_turn(...)*, *info(...)* e *menu(...)*.
- può propagare a **TurnControl** l'accesso non privilegiato a *_game.state*.



Metodo	Privilegi d'accesso ai metodi esposti da GameControl
<i>GameControl(...)</i>	: <i>public</i>
<i>title(...)</i>	: <i>public</i>
<i>menu(...)</i>	: <i>public</i>
<i>getGameState(...)</i>	TurnControl
<i>newGame(...)</i>	NewGameCommand
<i>quitGame(...)</i>	QuitGameCommand

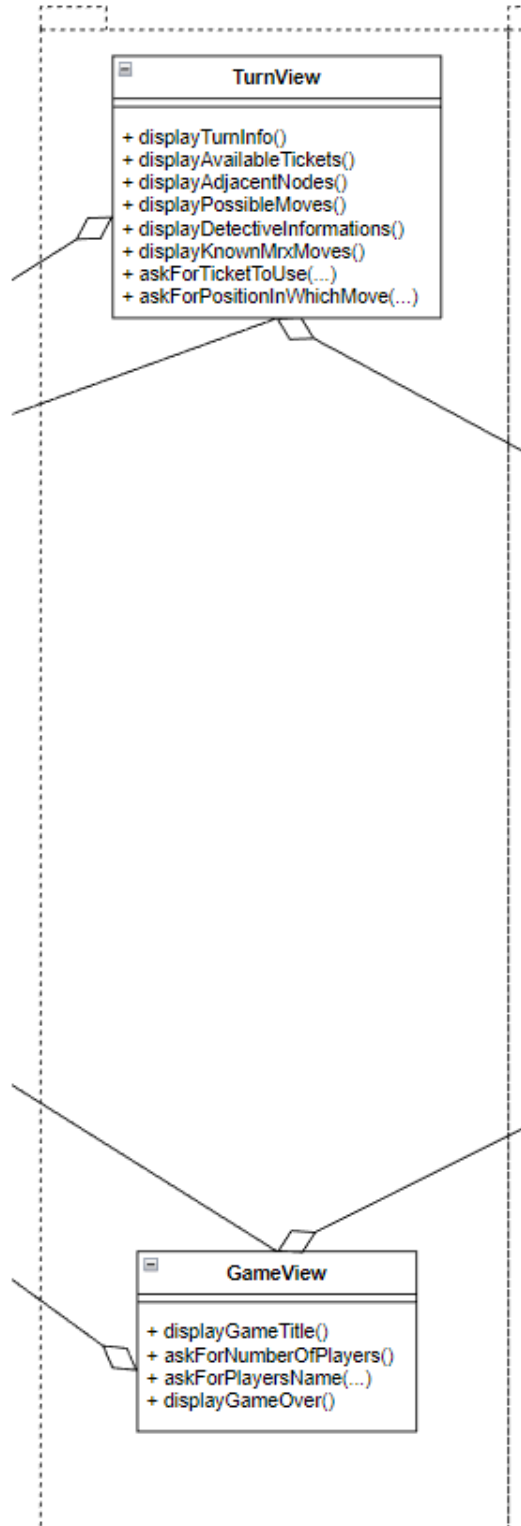
TurnControl:

Metodo	Privilegi d'accesso ai metodi esposti da TurnControl
<i>TurnControl(...)</i>	GameControl
<i>new_turn(...)</i>	GameControl
<i>info(...)</i>	GameControl
<i>menu(...)</i>	GameControl
<i>getTurn(...)</i>	TurnMenuControl
<i>playTicket(...)</i>	PlayTicketCommand
<i>possibleMoves(...)</i>	PossibleMovesCommand
<i>availableTickets(...)</i>	AvailableTicketsCommand
<i>detectiveInformations(...)</i>	DetectiveInformationsCommand
<i>adjacentNodes(...)</i>	AdjacentNodesCommand
<i>MrxInformationsCommand(...)</i>	MrxInformationsCommand
<i>quitToGameMenu(...)</i>	QuitToGameMenuCommand

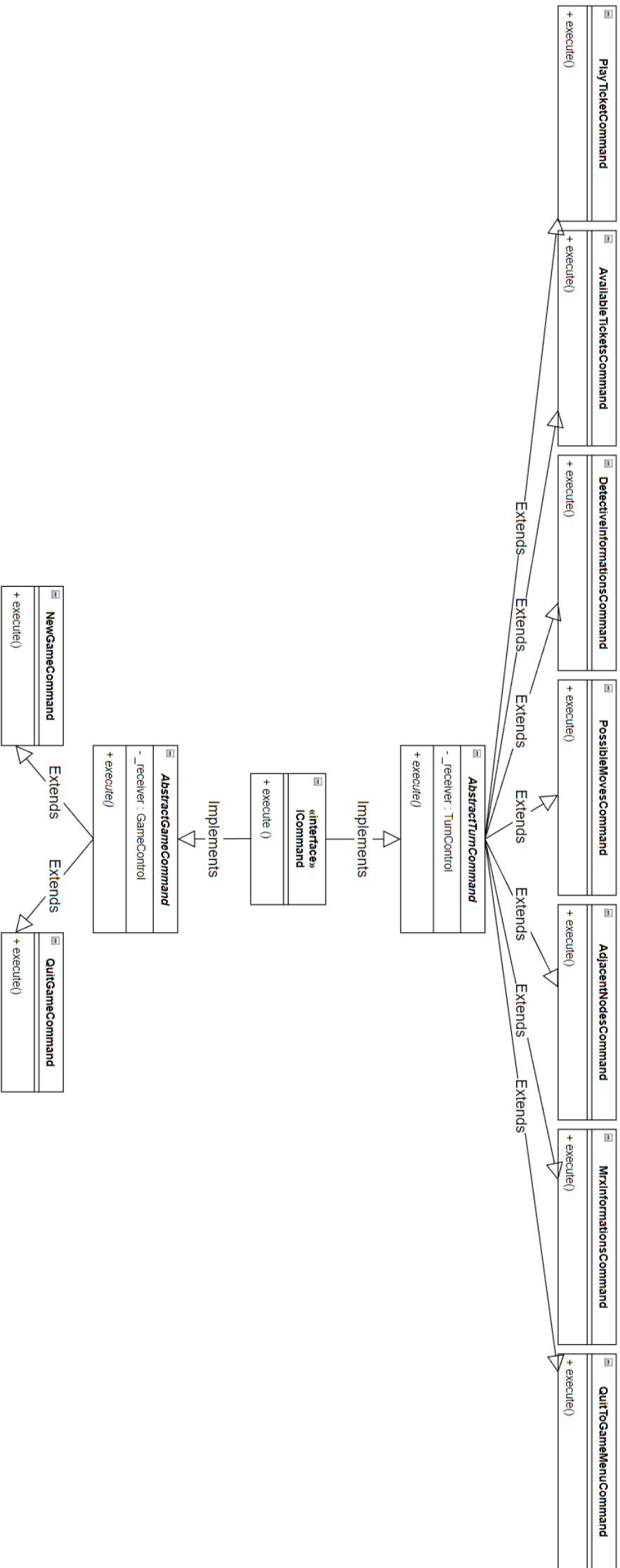
4.3.3 Package - game.view

Il package *game.view* contiene 2 classi: **GameView** e **TurnView**.

Queste si occupano di mostrare a schermo informazioni sullo stato del gioco o del turno, oltre che a richiedere un input valido all'utente.

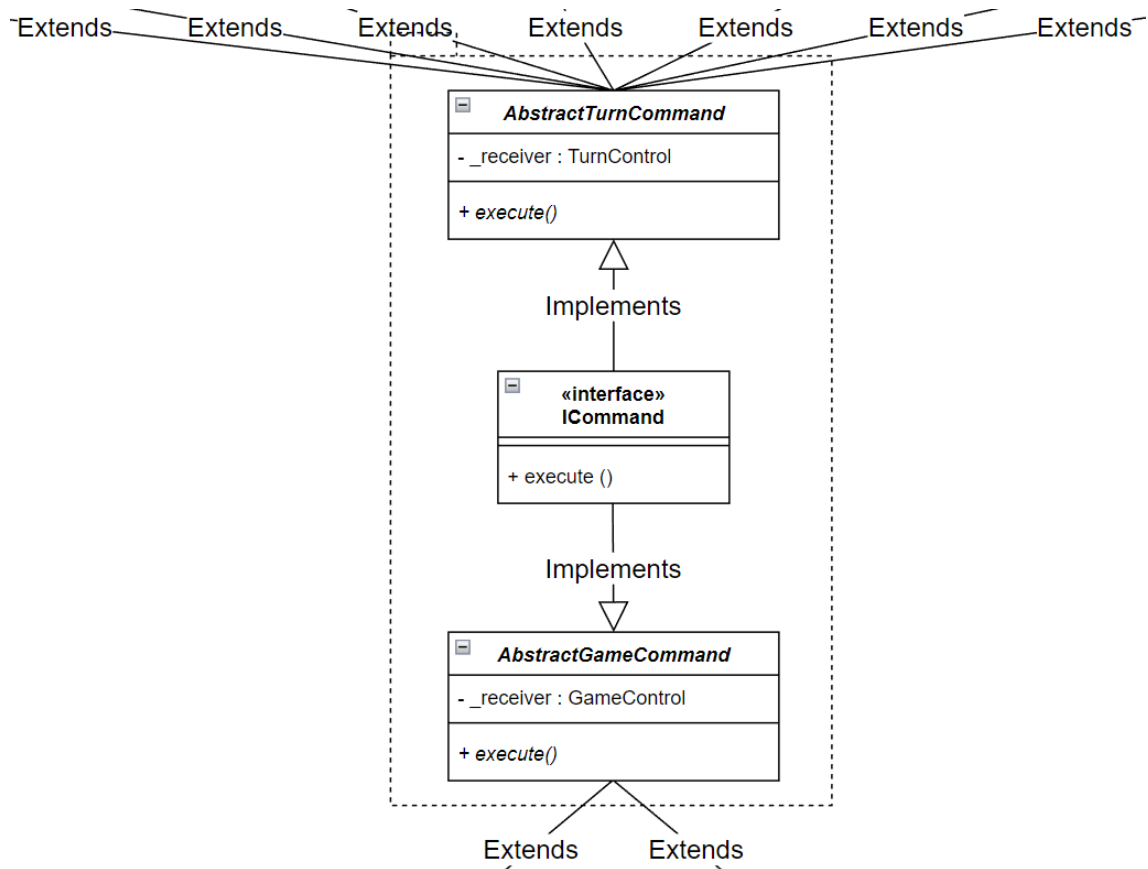


4.4 Class Diagram - Command



Il package *command* contiene 3 classi: ***ICommand*** , ***AbstractGameCommand*** e ***AbstractTurnCommand*** e 2 sub-packages: *command.game* e *command.turn* .

ICommand è l'interfaccia per tutti i comandi e dichiara il metodo *execute()* che implementerà la logica di esecuzione del comando. ***AbstractGameCommand*** e ***AbstractTurnCommand*** implementano ***ICommand*** e definiscono l' attributo protetto *_receiver* che verrà assegnato dal costruttore; ogni comando durante la sua esecuzione avrà così la possibilità di invocare metodi dell' oggetto da lui referenziato, che può essere di tipo ***GameController*** o ***TurnControl*** rispettivamente.

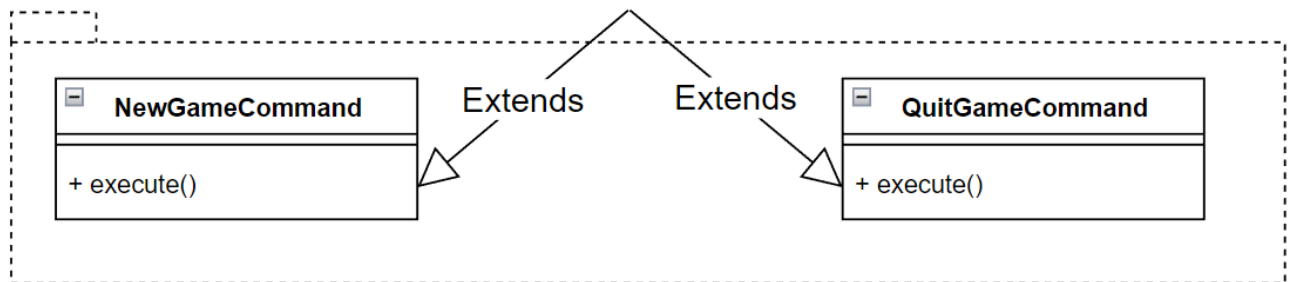


4.4.1 Package `command.game`

Il package *command.game* contiene i comandi:

- *NewGameCommand*
- *QuitGameCommand*

che performano invocazioni su un *_receiver* di tipo *GameControl*.

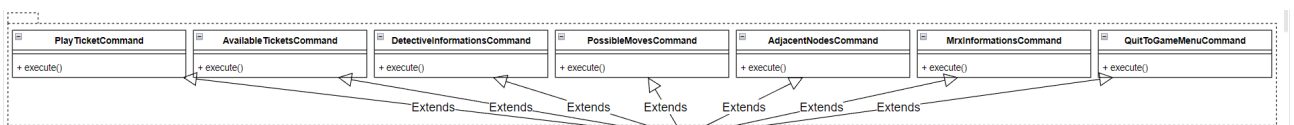


4.4.2 Package command.turn

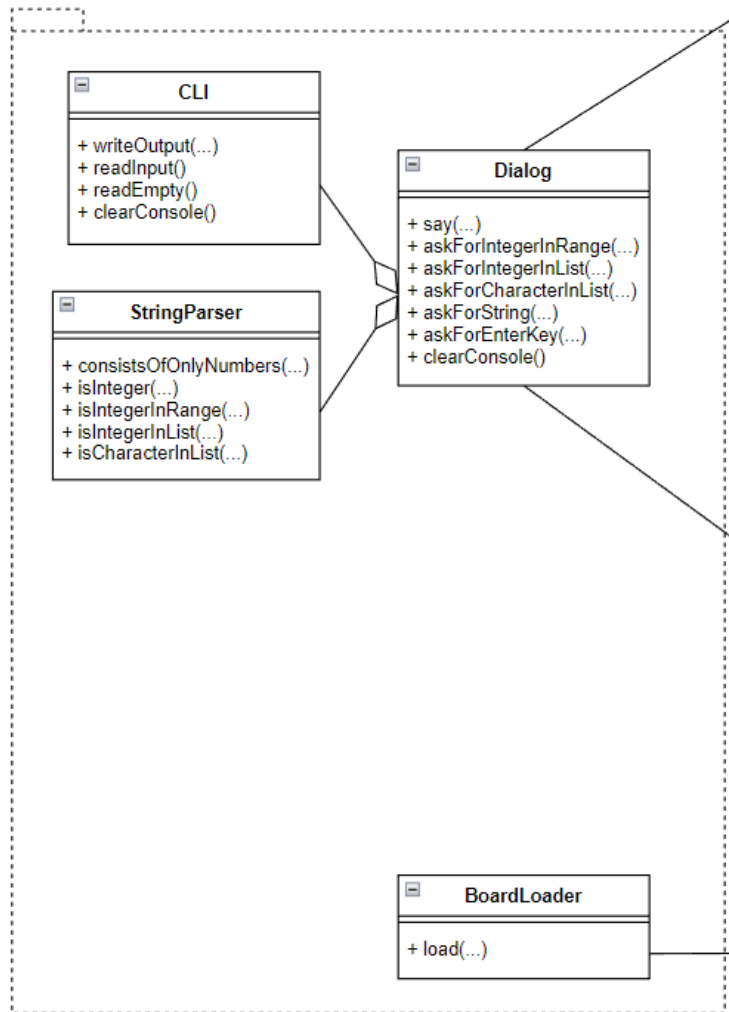
Il package *command.turn* contiene i comandi:

- *AdjacentNodesCommand*
- *AvailableTicketsCommand*
- *DetectiveInformationsCommand*
- *MrxInformationsCommand*
- *PlayTicketCommand*
- *PossibleMovesCommand*
- *QuitToGameMenuCommand*

che performano invocazioni su un *_receiver* di tipo *TurnControl*.



4.5 Class Diagram - Utils



Il package *Utils* contiene 4 classi: **CLI**, **Dialog**, **StringParser** e **BoardLoader**.

CLI si occupa di scrivere, leggere e pulire la console.

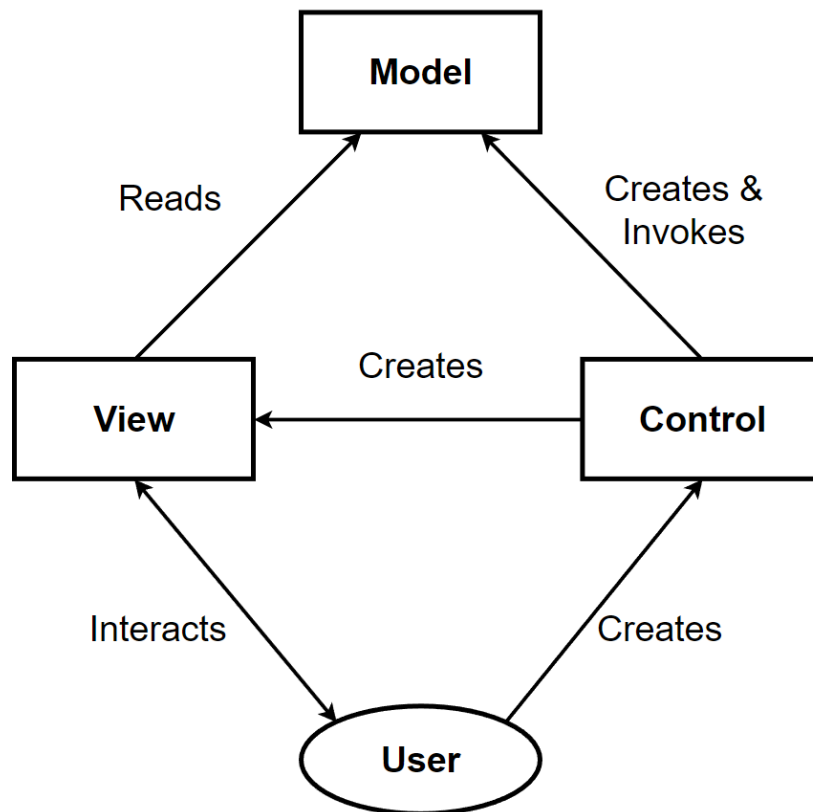
StringParser si occupa di verificare l'integrità di una stringa, per esempio verificando se questa rappresenta un numero.

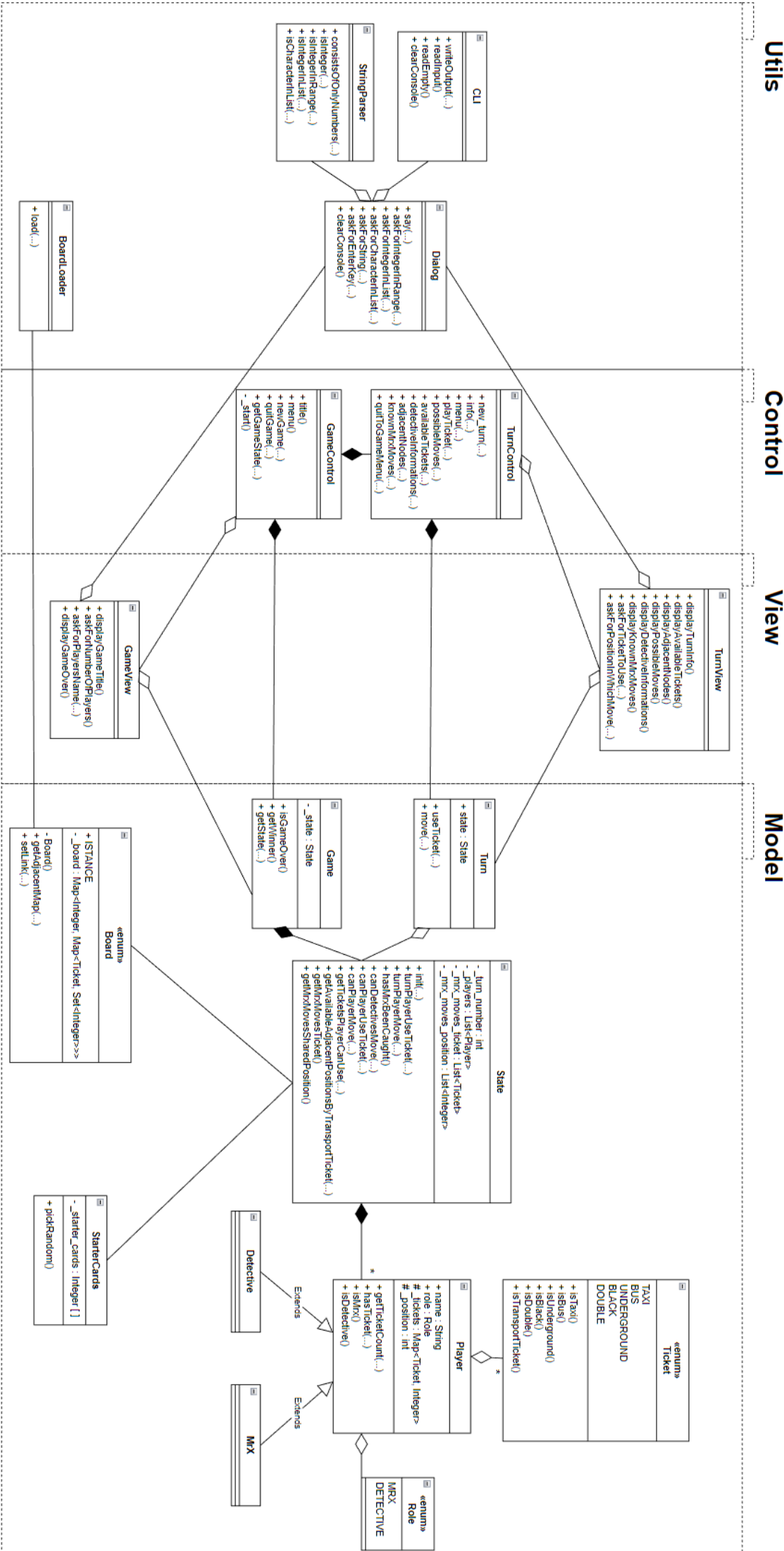
Dialog unisce le funzionalità delle due classi precedenti, in una unica classe che esponga una sequenza di funzioni per la gestione dell' IO.

BoardLoader si occupa della lettura di un file di testo contenente la mappatura della tabella di gioco e del caricamento di questo in un oggetto di tipo Board.

4.6 Design

4.6.1 MVC Design



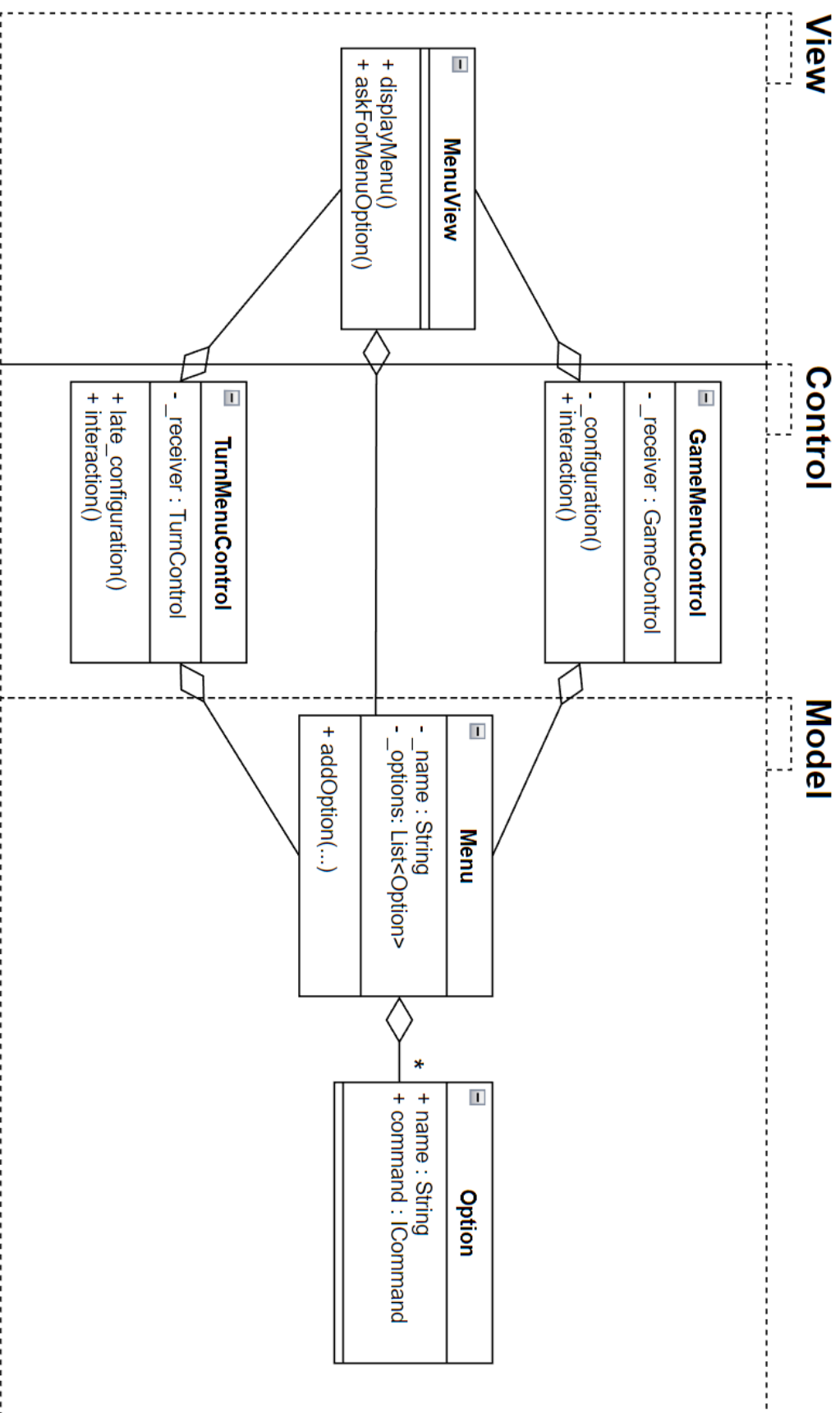


Utils

Control

View

Model



4.6.2 Friendship Mechanism

```
////////////////////////////////////  
//                          ACCESS KEY                          //  
////////////////////////////////////  
  
// Friendship mechanism: https://stackoverflow.com/a/18634125/8411453  
public static final class AccessKey  
{  
    private AccessKey() {}  
}  
private static final AccessKey _access_key = new AccessKey();
```

Il codice sopra riportato, implementa un meccanismo di "friendship", che risolve il problema di avere metodi pubblici invocabili solo da una o poche classi di oggetti. Una soluzione alternativa avrebbe richiesto l'utilizzo di interfacce, il problema è che si avrebbe inquinato il progetto di un elevato numero di interfacce, che non avrebbero avuto altra funzione se non quella di incapsulamento.

Il meccanismo funziona nel seguente modo:

1. Una classe **A** vuole esporre un metodo **a()** invocabile da **B**.
2. La classe **B** definisce la sottoclasse **AccessKey** e detiene un oggetto **_access_key** di tale classe (vedi codice soprastante).
3. La classe **A** implementa il metodo **a()** come:

```
public void a(B.AccessKey access_key)  
{  
    Objects.requireNonNull(access_key);  
}
```

Si chiede perciò un oggetto di tipo **B.AccessKey** che sia non nullo.

4. Un oggetto di classe **B** può dunque ora invocare **A.a()** passando come parametro a tale metodo la propria chiave d'accesso.

4.6.3 Debugger

La classe **Debugger** è una classe particolare: essa detiene un oggetto *master_key*, questa chiave può essere utilizzata al posto di ogni *_access_key* che è specifica per classe; difatto permettendo di ottenere i privilegi necessari ad invocare ogni metodo che richieda una chiave di accesso.

```
/*
 * The Debugger class has privileged access to all the game components
 */
public class Debugger
{
    public static class MasterKey
    {
        private MasterKey() {}
    }
    public static final MasterKey master_key = new MasterKey();
}
```

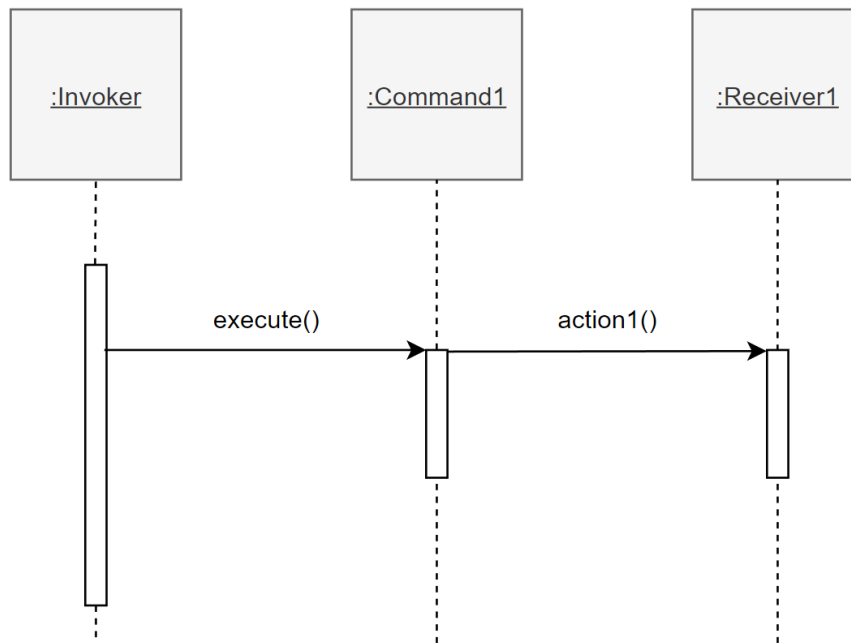
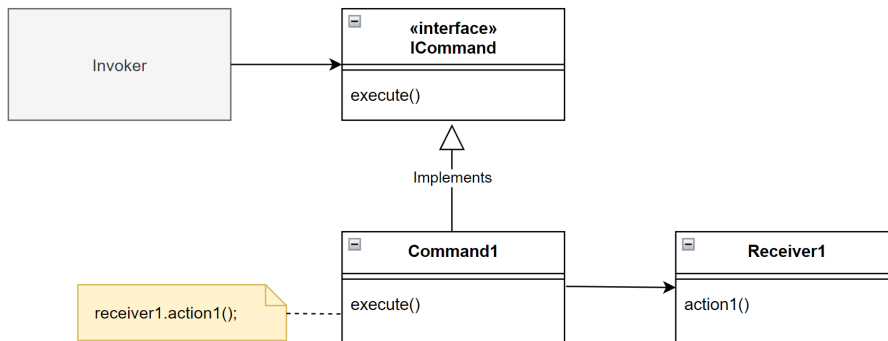
Ogni classe che vuole permettere l'accesso attraverso la *master_key*, modifica la sua *AccessKey* affinché accetti di essere costruita attraverso la chiave maestra:

```
public static final class AccessKey
{
    private AccessKey() {}
    public AccessKey(Debugger.MasterKey master_key) { Objects.requireNonNull(master_key); }
}
private static final AccessKey _access_key = new AccessKey();
```

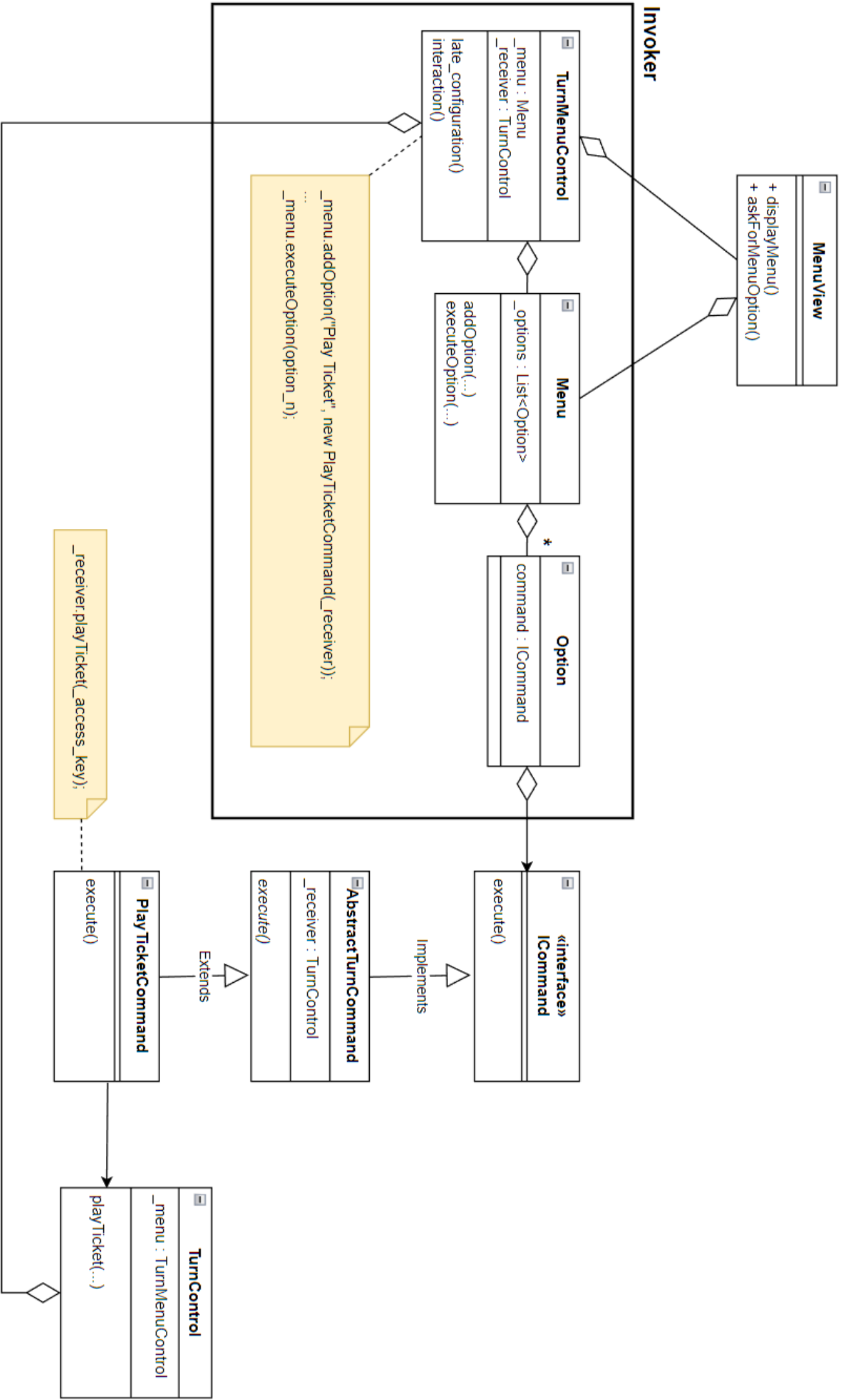
4.7 Command Design Pattern

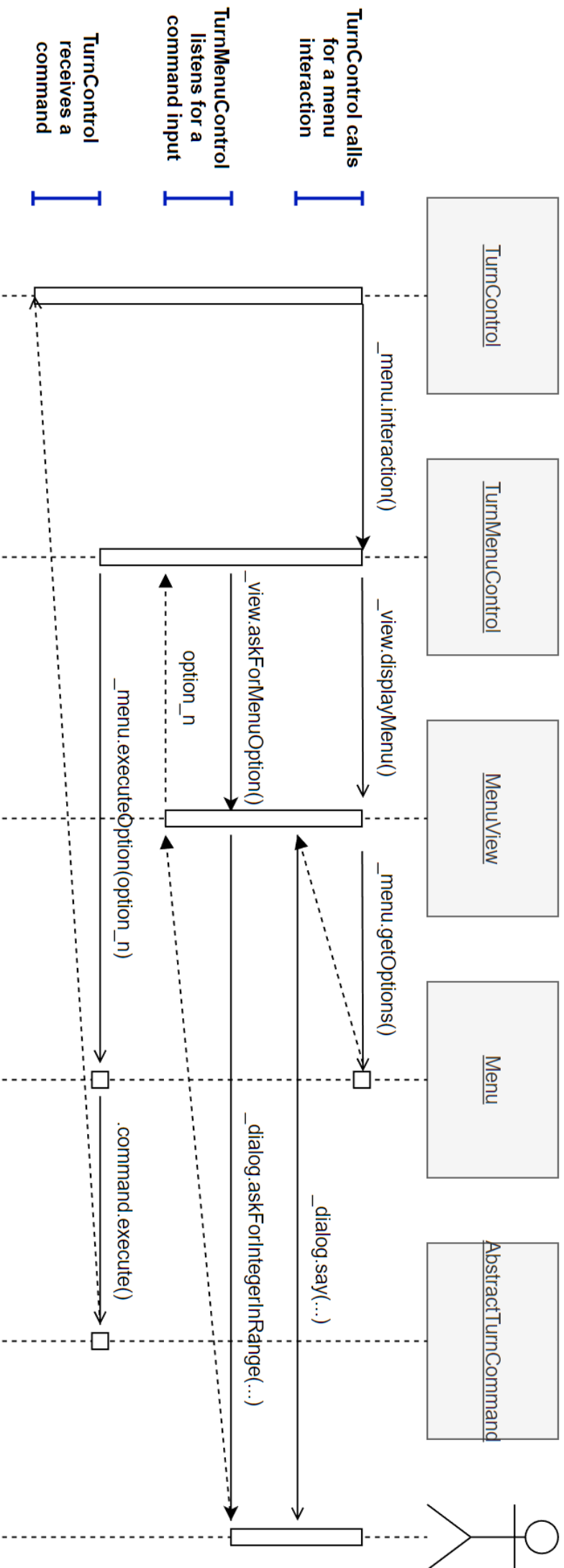
Command è un *Design Pattern* comportamentale, che incapsula la logica di invocazione di un comando. Esso è una interfaccia nell' esecuzione delle operazione tra un generico **Invoker** ed un generico **Receiver**, dove il primo si limita a chiamare il metodo *execute()* definito in una interfaccia **ICommand** implementato da un **ConcreteCommand** che si occupera di gestire l'invocazione dei metodi definiti da Receiver. Si è difatto separato le responsabilità di ogni classe.

4.7.1 Generic Command Pattern



4.7.2 Implemented Command Pattern





TurnControl calls
for a menu
interaction

TurnMenuControl
listens for a
command input

TurnControl
receives a
command

4.7.3 Singleton Design Pattern

Viene usato il *Design Pattern Singleton* per la classe **Board**.

Perchè ?

- La classe **Board** non espone pubblicamente nessun attributo.
- La classe **Board** espone pubblicamente un unico metodo: *getAdjacentMap(Integer node)*.
- Il metodo esposto ritorna una *Collections.unmodifiableMap*.

Difatto **Board** è una classe di sola lettura.

Inoltre:

- La classe **Board** durante la sua costruzione legge il file *Board.txt* e ne carica il contenuto in una mappa.

Tale operazione è costosa.

Pertanto ha senso che durante l'intera esecuzione del programma sia garantita la presenza di una sola istanza di classe **Board**.

Perchè un Enum ?

è l'implementazione adottata da Josh Bloch in una sua presentazione:

http://www.youtube.com/watch?v=pi_I7oD_uGI#t=28m50s

Chapter 5

Testing

Viene Implementata una unità di testing con **JUnit 5**

Il package *Test* contiene la classe ***StateTest*** che si occupa di effettuare test sui metodi di tale classe.

StateTest

- **Test01**

Si posiziona MrX in un nodo random della mappa, si posizionano Detectives in tutti i nodi adiacenti a MrX, si vuole verificare che MrX non ha la possibilità di giocare alcun ticket.

Chapter 6

Demo

Game Title

```
*****
* Title          Scotland Yard
* Author         Marco De Groskovskaja
*****
```

Game Menu

```
*****
*   ### Game Menu   ###
*
* 1: New Game
* 2: Load Game
* 3: Settings
* 4: Quit
*
* Enter the menu option:
```

Game Menu - New Game

```
*****
* Enter the number of players (from 3 to 6): 3
*****
* Enter player name: Marco
* Enter player name: Luca
* Enter player name: Maria
```

Turn Menu

```
*****
*   ### Turn Menu   ###
*
*   1: Play Ticket
*   2: Available Tickets
*   3: Detective Informations
*   4: Mrx Informations
*   5: Possible Moves
*   6: Adjacent Nodes
*   7: Quit to Game Menu
*
*   Enter the menu option:
```

Turn Menu - Play Ticket

```
*****
*   Enter the ticket to use (1: TAXI, 2: BLACK, 3: DOUBLE): 1
*   Enter the node to which move [34, 35, 62, 63]: 62
```

Turn Menu - Available Tickets

```
*****
* Tickets:
*   - TAXI:          9
*   - BUS:           8
*   - UNDERGROUND:  4
*
* Press ENTER to return the menu
```

Turn Menu - Mr X Informations

```
*****
* Mrx last known positions:
*   Move: 3          Position: 100
*
* Mrx moves:
*   Move: 1          Ticket: TAXI
*   Move: 2          Ticket: TAXI
*   Move: 3          Ticket: BLACK
*
* Press ENTER to return the menu
```

Turn Menu - Detective Informations

```
*****
* Detectives:
*
*   -----
*   Luca
*   -----
*   position: 133
*
*   tickets :
*           TAXI          : 9
*           BUS           : 8
*           UNDERGROUND   : 4
*
*   -----
*   Maria
*   -----
*   position: 173
*
*   tickets :
*           TAXI          : 9
*           BUS           : 8
*           UNDERGROUND   : 4
*
* Press ENTER to return the menu
```