Documentazione del progetto

“ESCAPE FROM THE ALIENS IN OUTER SPACE”

Prova finale di Ingegneria del Software

Andrea Lekkas, Michele Madaschi

Indice

1 Struttura generale

1.1 Perchè scegliere un doppio Model-View-Controller

Le varie responsabilità del Server vengono suddivise, costruendo una struttura del tipo Model-View-Controller, in questo modo:

- Model : Comprende lo stato del gioco e le sue entità
- Controller : La logica applicativa, che invia comandi al Model a seconda di quanto ricevuto dagli utenti
- View : Riceve i comandi / le richieste dai Client

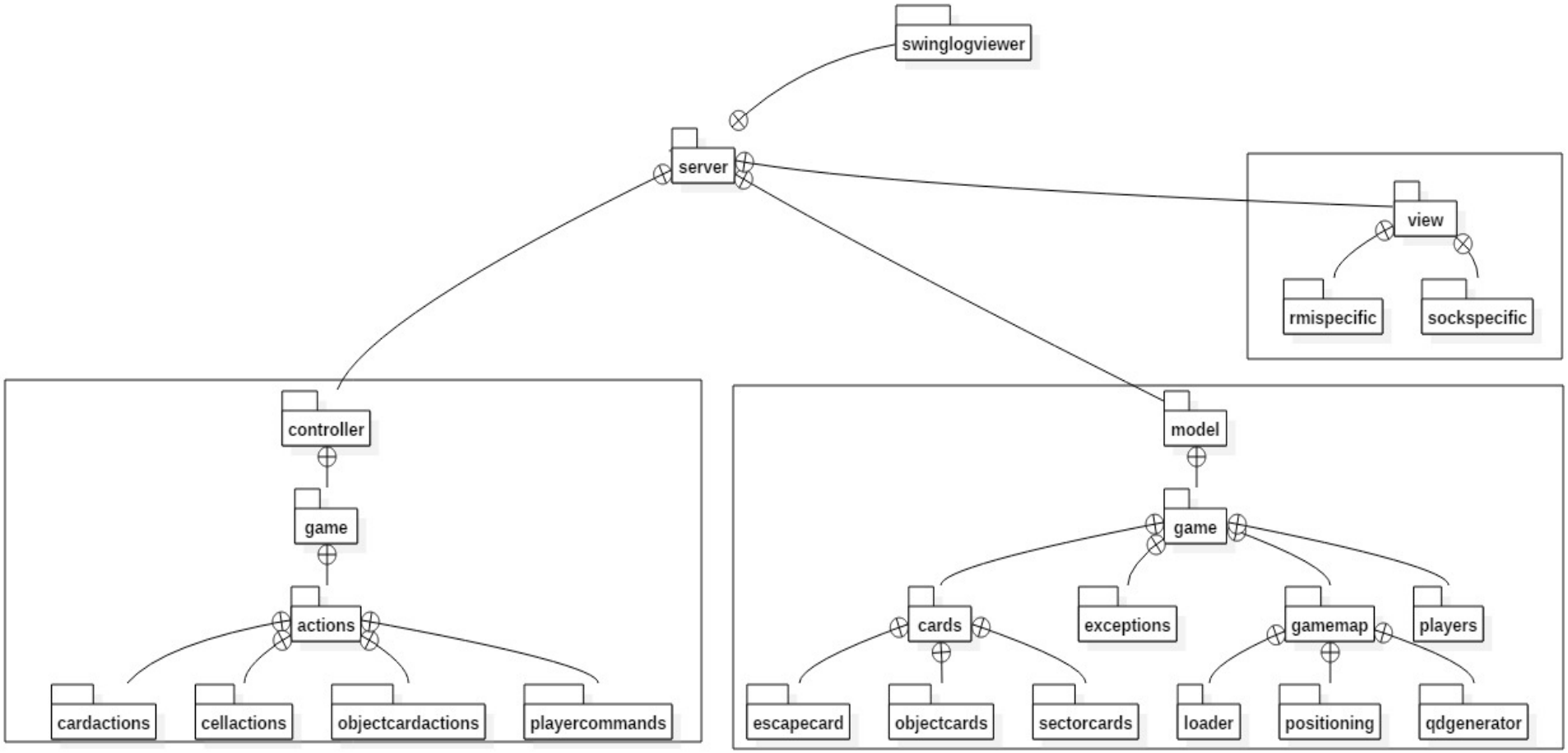

Illustration 1: Struttura dei package del Server

Anche il Client impiega Model-View-Controller, infatti:

- View : Organizza l'interfaccia utente (sia CLI sia la GUI con Swing), riceve i comandi dell'utente e li passa al Controller.

- Controller : A seconda dei messaggi ricevuti dal Server, può modificare il Model, e modificare la View (ad esempio, inviare una richiesta all'utente).
  Inoltre, a seconda dell'input dell'utente ricevuto dalla View,
  nel caso della comunicazione via Socket dà l'ordine alla connessione di inviare la stringa al Server, mentre nel caso della comunicazione via RMI chiama il metodo appropriato dell'interfaccia remota del Server.

- Connection : Gestisce la comunicazione con il Server, può usare sia Socket sia RMI

- Model : Contiene alcune informazioni sullo stato del gioco che sono utili al Client che fa uso della GUI; ad esempio : lo status corrente degli altri giocatori, il numero del turno, etc.

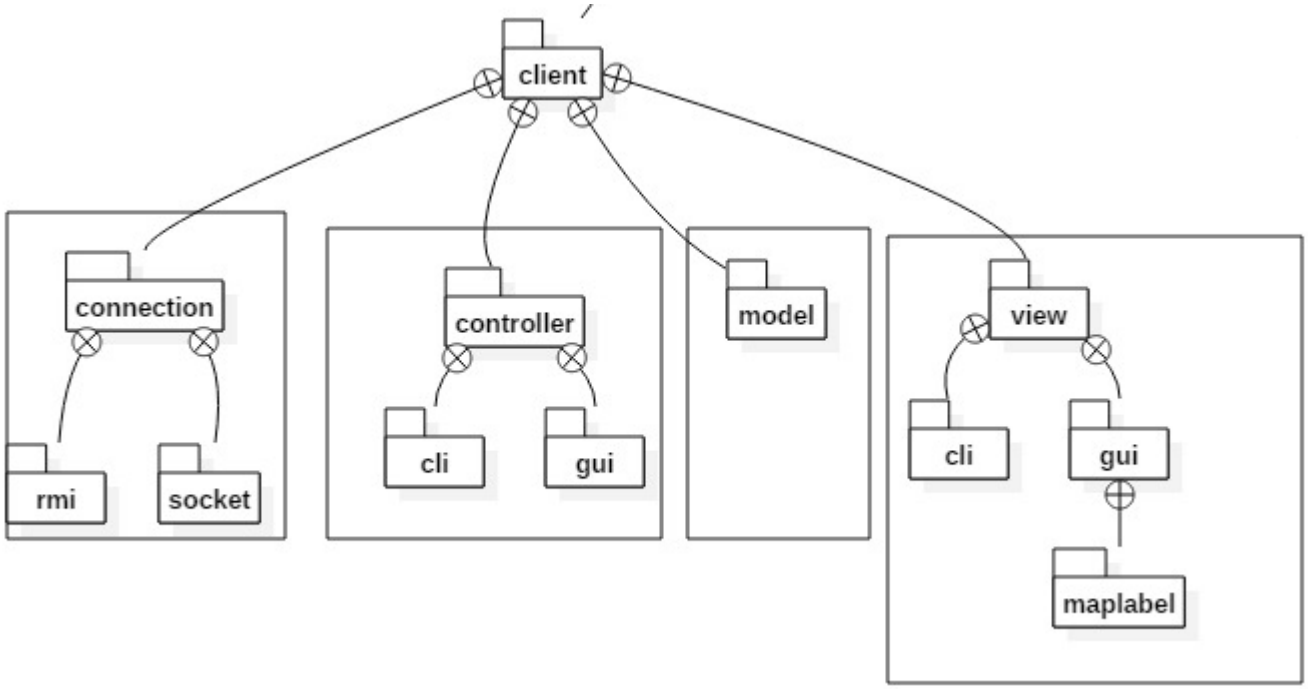

Illustration 2: Struttura dei package del Client

2 Il Server

2.1 Controller: il turno

Struttura del turno e azioni possibili	
Umano	Alieno
• Inizia turno • Può giocare una Carta Oggetto (Teletrasporto/Adrenalina/Sedativi/Luci) • Muove: ◦ richiedi input all'utente ◦ la GameMap controlla la legittimità del movimento proposto e aggiorna la posizione • Se decide di attaccare (cosa che può fare solo giocando la Carta Oggetto Attacco), non deve pescare carte settore • Conseguenza del movimento: ◦ Se richiesto, pesca una carta Settore/Scialuppa ◦ Se una Carta è stata pescata, esegui azione corrispondente ◦ Se la Carta Settore lo richiede, 2ª azione: pesca una carta Oggetto • Può giocare una Carta Oggetto (Teletrasporto/Luci) • Fine turno (n: resettare eventuali stati, come Adrenalina)	• Inizia turno • Muovi: ◦ richiedi input all'utente ◦ la GameMap controlla la legittimità del movimento proposto e aggiorna la posizione • Può Attaccare nella cella in cui si è spostato. Se decide di attaccare non deve pescare carte settore • Se non ha attaccato, Conseguenza del movimento: ◦ Se richiesto, pesca una carta Settore ◦ Se la Carta è stata pescata, esegui azione corrispondente (Noise/Silence) ◦ Se la Carta Settore lo richiede, 2ª azione: pesca una carta Oggetto • Fine turno

Definiamo le responsabilità per alcune classi:

- **TurnHandler** (nel Controller) contiene la logica applicativa del turno, ed esegue le varie fasi.
- **UserMessagesReporter** (nel Controller) contiene funzioni per richiedere input all'utente.
- **Player** (nel Model) contiene informazioni sullo stato del personaggio.
- **GameMap** (nel Model) contiene la posizione di ciascun giocatore.
- **Announcer** annuncia in broadcast gli eventi del gioco.

Nel TurnHandler, il metodo executeTurnSequence() , in ordine, invoca i metodi:

```
hailPlayer();
initialize();  // step 0
turnBeforeMove();  // step 1
turnMove();  // step 2
turnLand();  // also step 2
turnAfterMove();  // step 3
deInitialize();  // cleanup
farewellPlayer();
```

TurnHandlerHuman e TurnHandlerAlien sono sottoclassi di TurnHandler, e implementano alcuni di questi metodi in maniera differente.

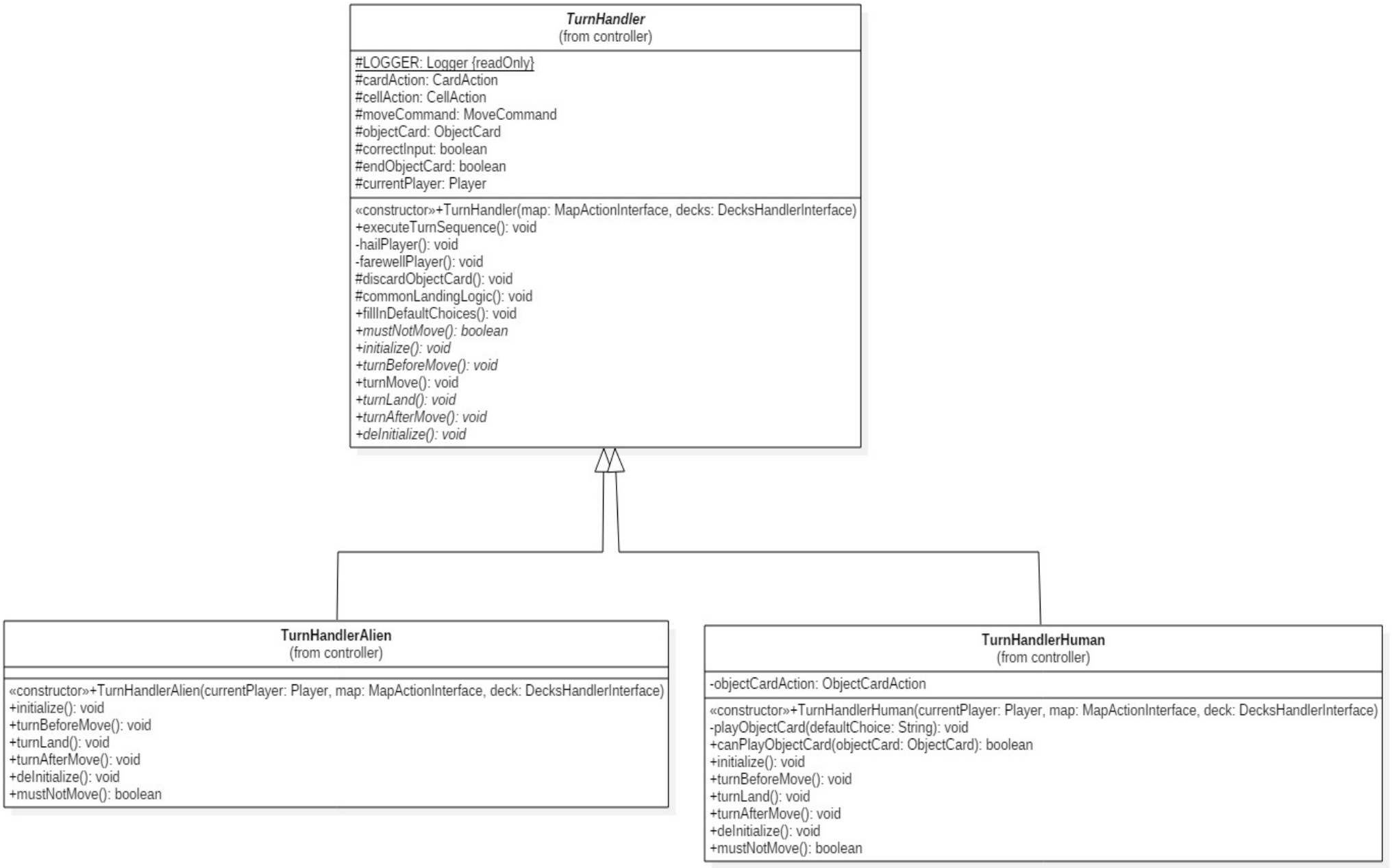

Illustration 3: TurnHandlers

Per quanto riguarda le azioni che il giocatore deve eseguire, si utilizza il Pattern Command: Le azioni sono incapsulate in oggetti. Questi oggetti contengono metodi come:
public void execute (Player currentPlayer, ...).

Ruoli nel pattern Command:

- Invoker - La classe che invoca i metodi dei comandi:
  - TurnHandler, TurnHandlerAlien, TurnHandlerHuman

- Command – Le classi che specificano le interfacce per i diversi tipi di comandi :
  - CellAction
  - CardAction
  - playerCommand
  - ObjectCardAction

- ConcreteCommand – Le classi che implementano i comandi, rispettivamente:
  - GetEscapeCardAction, GetSectorCardAction, NoCellAction
  - NoiseAnywhere, NoiseAnywhereWithObject, NoiseHere, NoiseHereWithObject, Silence, SilenceWithObject, Escape, CanNotEscape, DrawObjectCard
  - Move, Attack
  - Adrenaline, Sedatives, Lights, Teleport, AttackOrder, Defense

- Receiver – La classe che riceve gli effetti delle azioni invocate :
  - Il Player, con le sue sottoclassi Human e Alien; si trovano all'interno del Model, dato che contengono informazioni sullo stato corrente del personaggio
  - La GameMap, che contiene le posizioni dei Player

- Client – La classe che crea i comandi:
  - UserMessagesReporter, che crea il comando Movecommand
  - TurnHandler, che crea il comando DrawObjectCard
  - Le SectorCard, che ritornano CardActions
  - Le ObjectCard, che ritornano ObjectCardActions

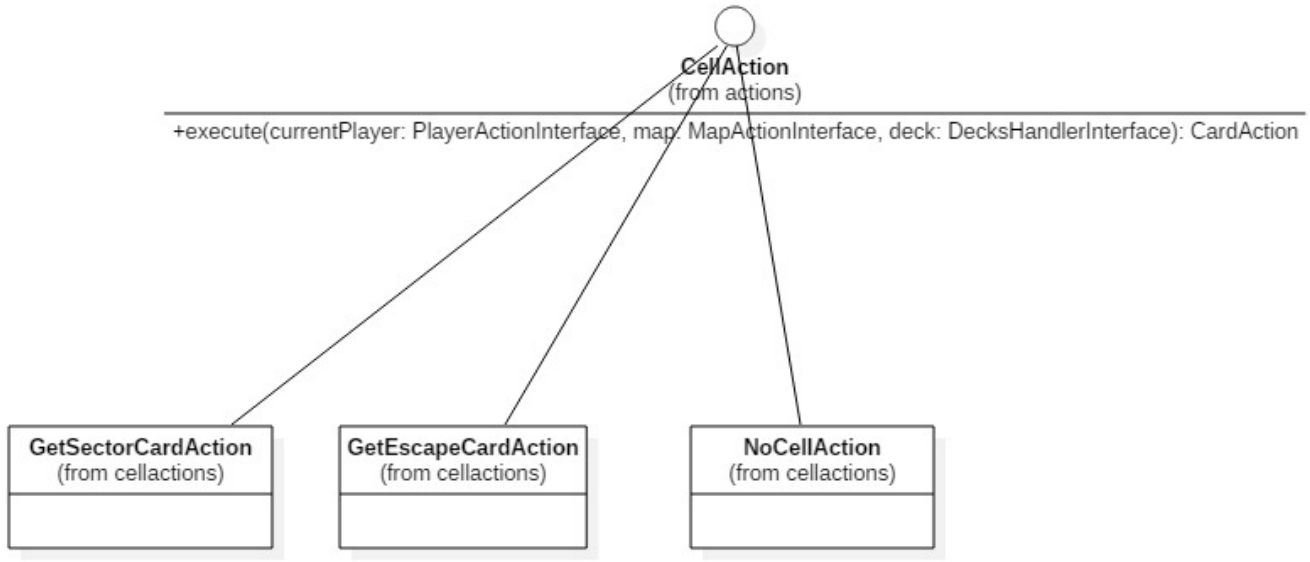

Illustration 4: CellActions

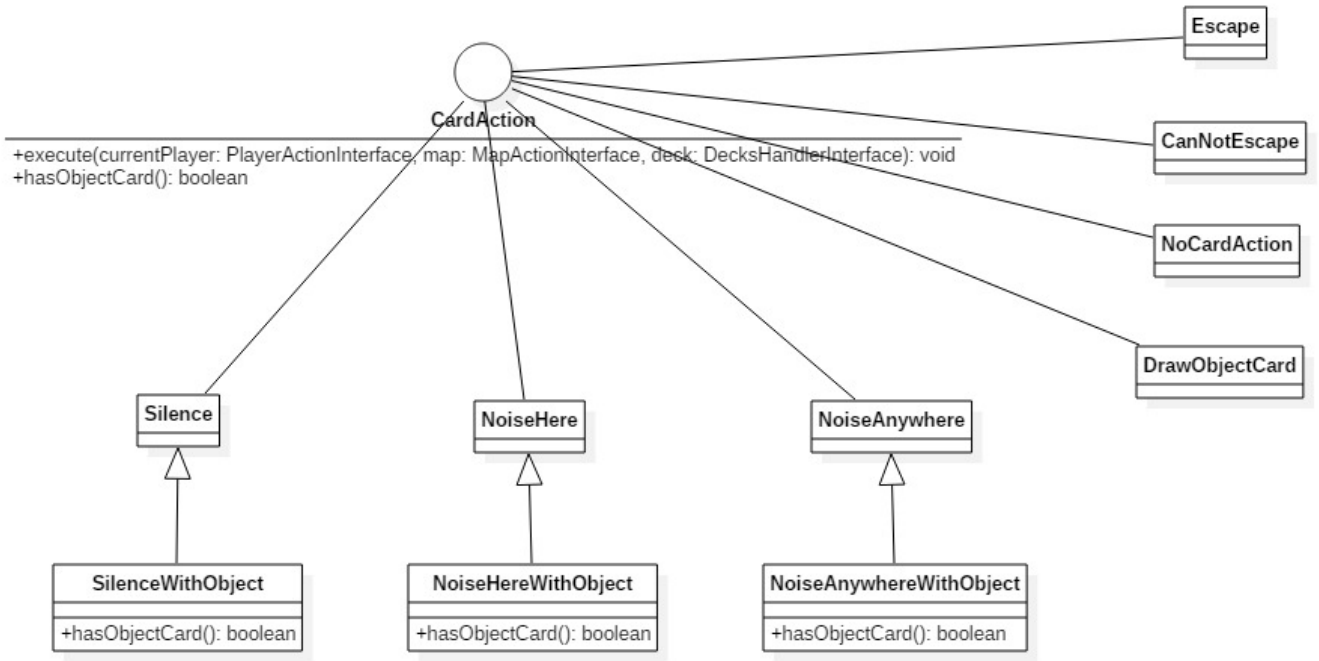

Illustration 5: CardActions

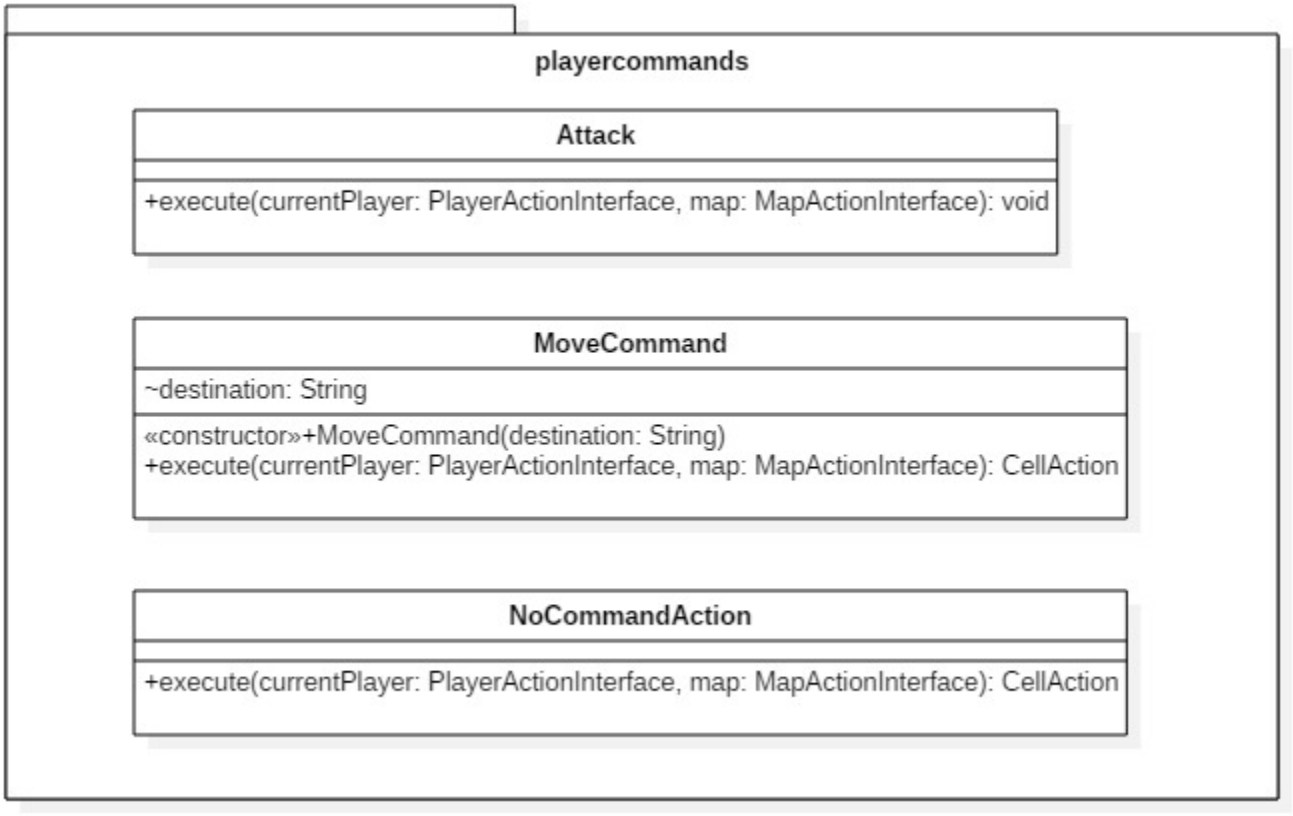

Illustration 6: Player commands

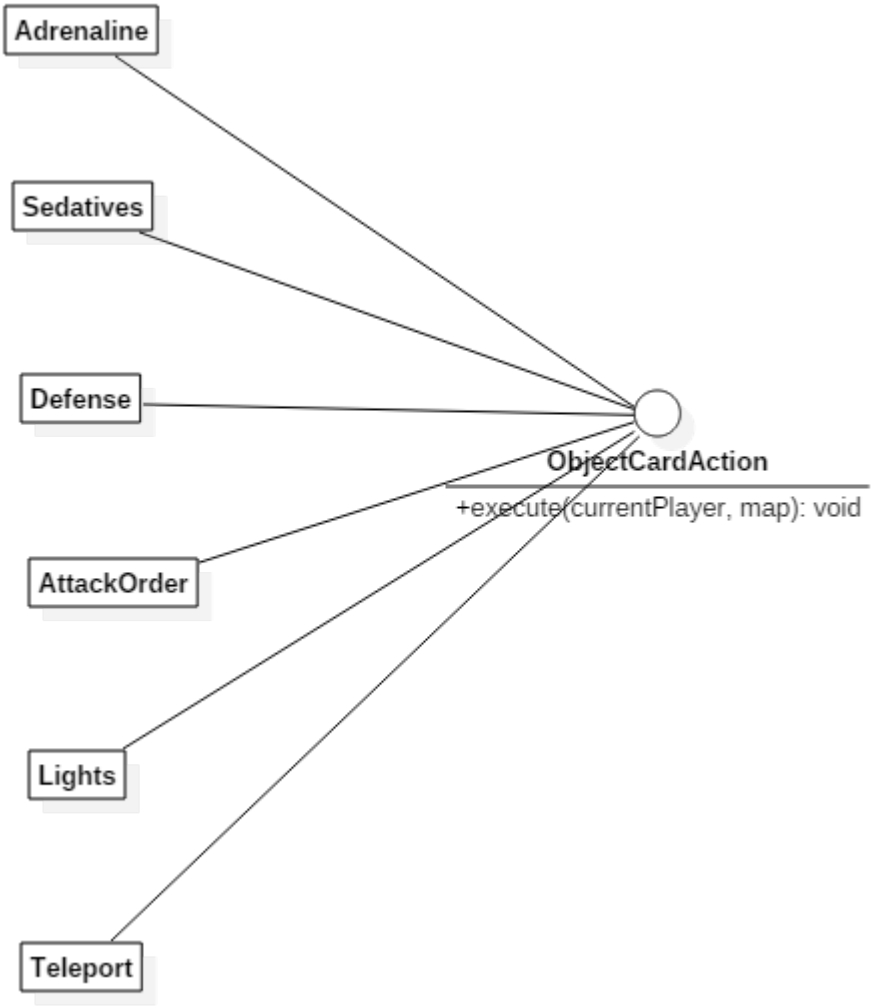

Illustration 7: ObjectCardActions

Seguono i diagrammi di sequenza del turno umano:

- Inizio del turno e fase pre-mossa
- Movimento
- Post-mossa, CellAction
- Post-mossa, estrarre carta settore
- Giocare carta oggetto
- Cella Scialuppa

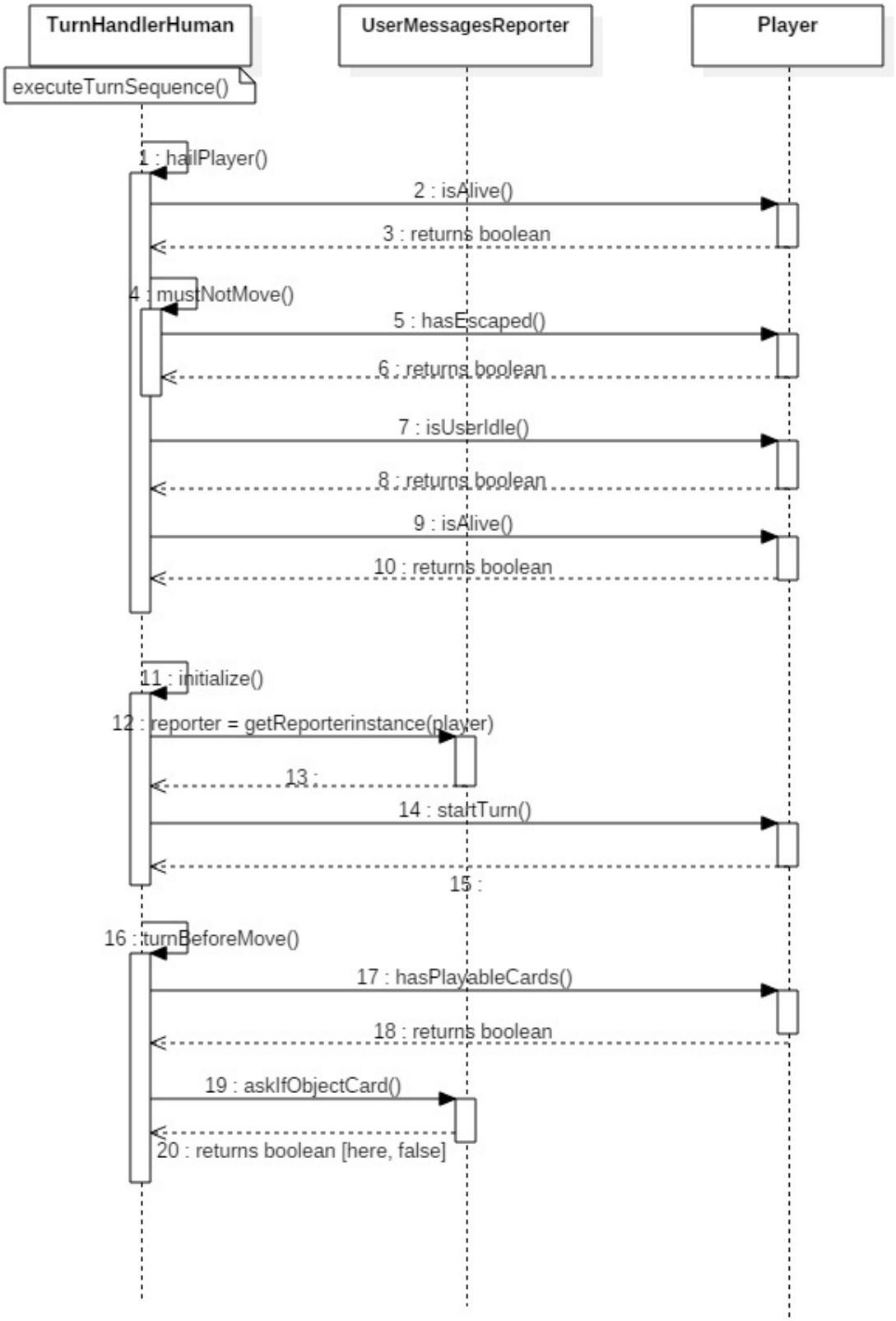

Illustration 8: Turno I : Inizio del Turno e fase pre-mossa

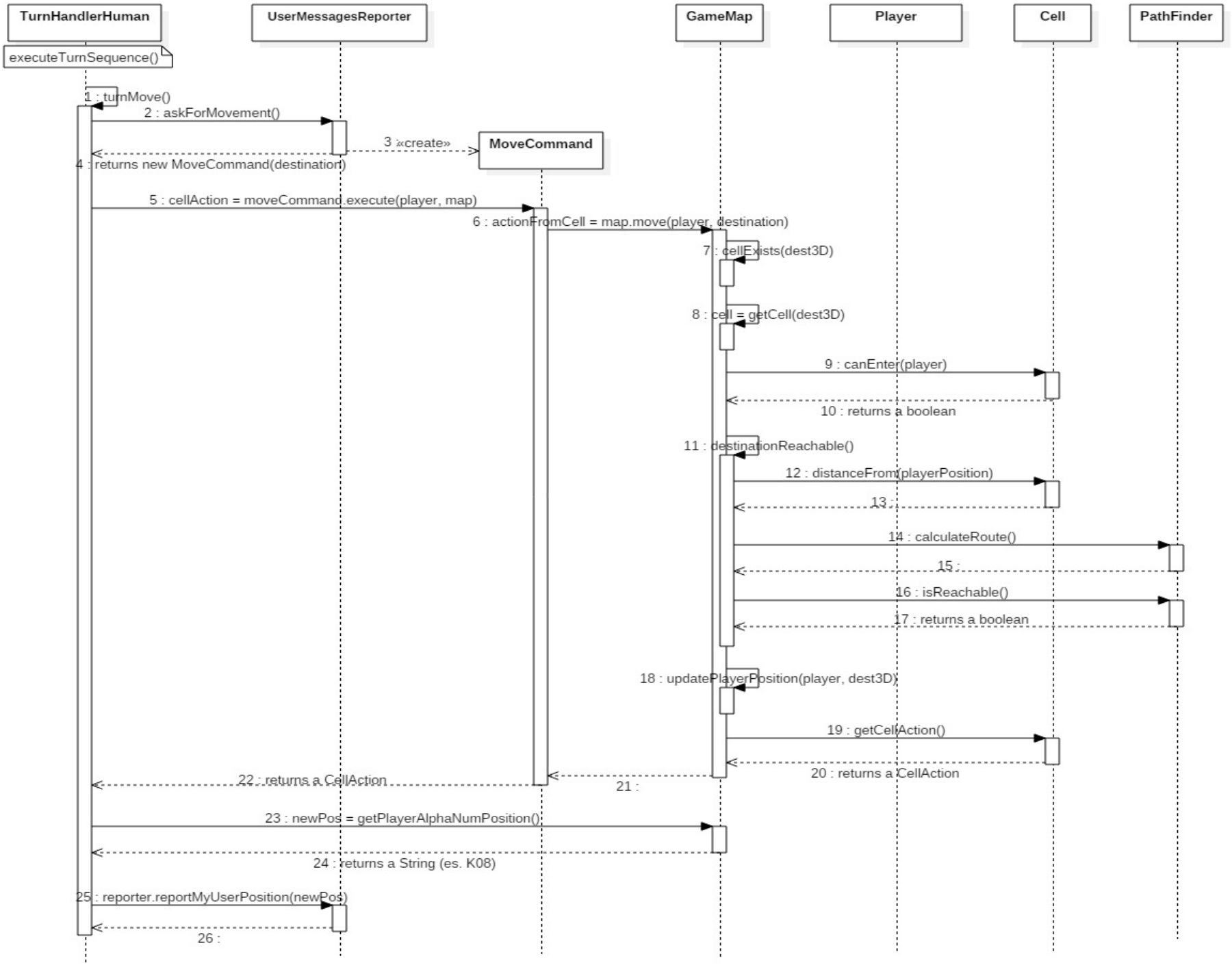

Illustration 9: Turno II : Movimento

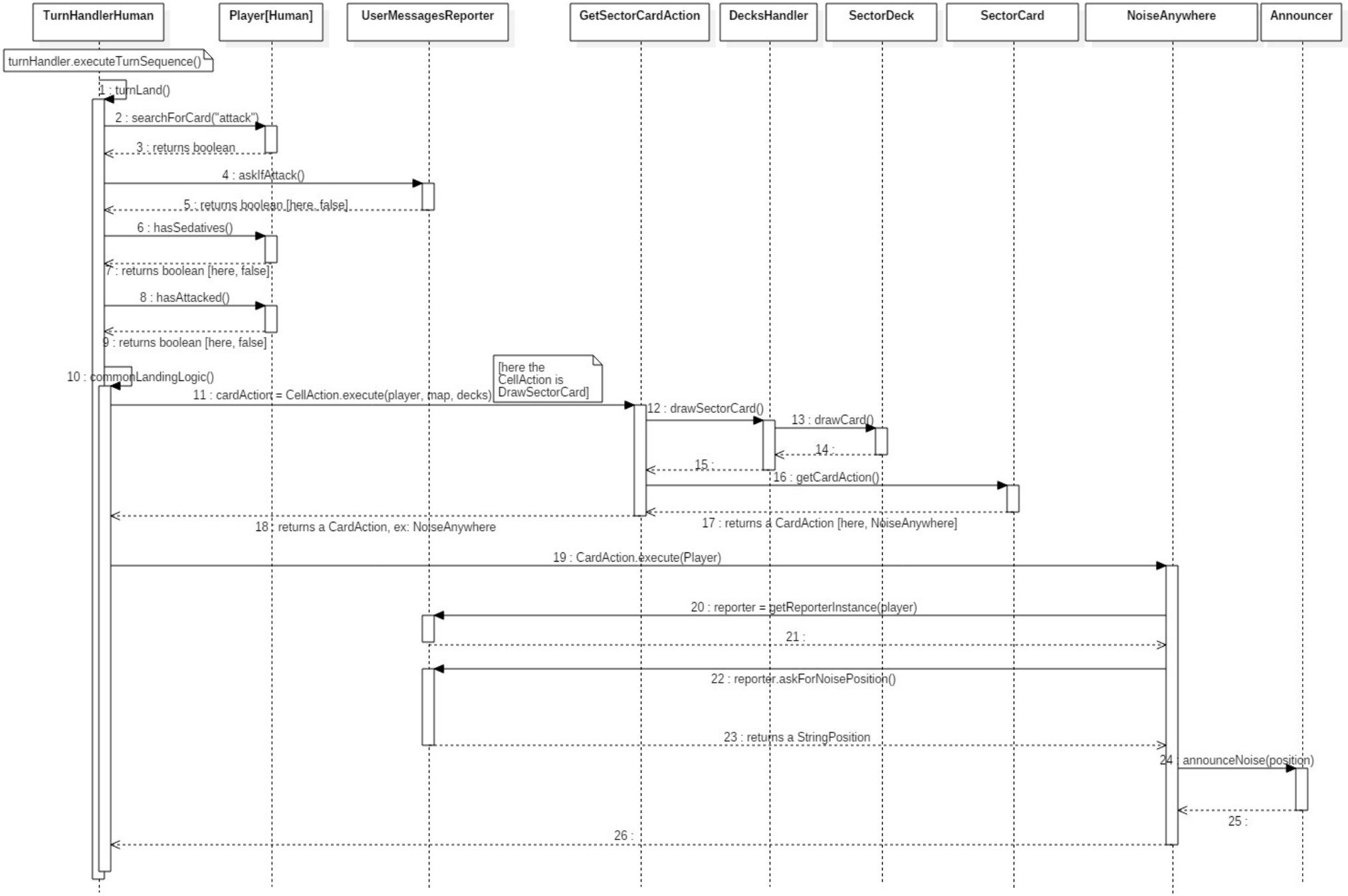

Illustration 10: Turno III: Post-mossa, CellAction

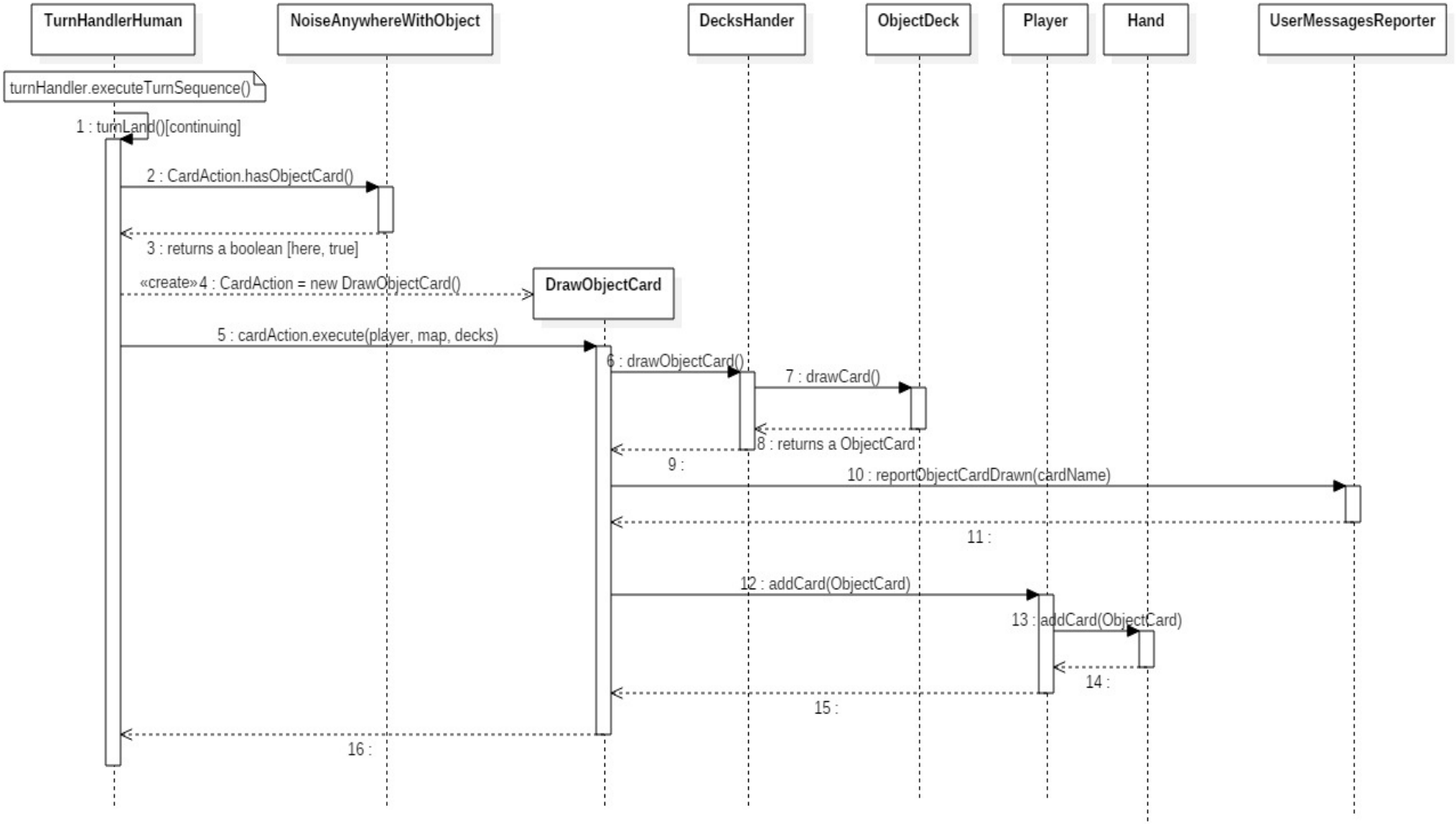

Illustration 11: Turno IV: Post-Mossa, Estrarre carta settore

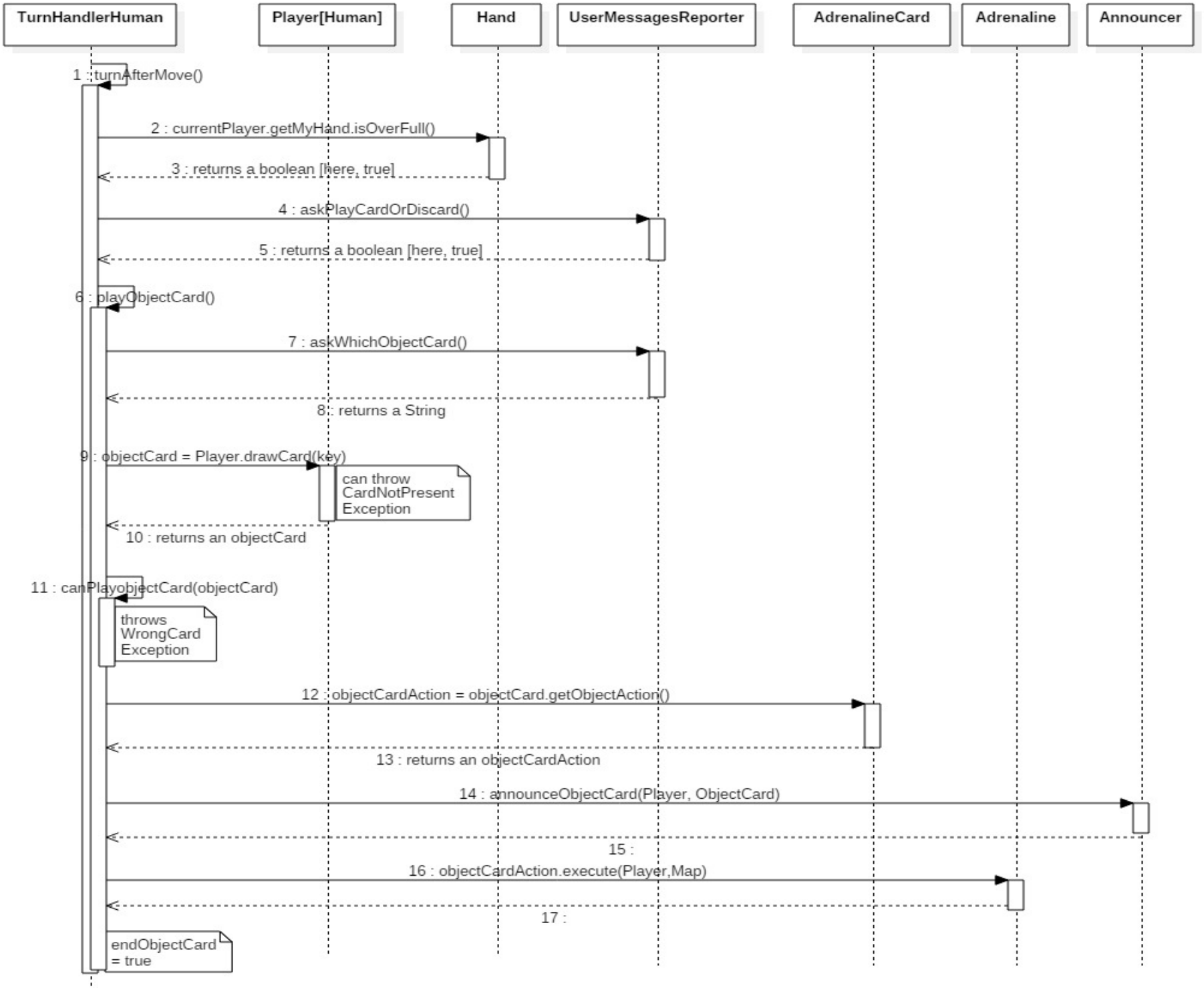

Illustration 12: Turno V : Giocare Carta Oggetto

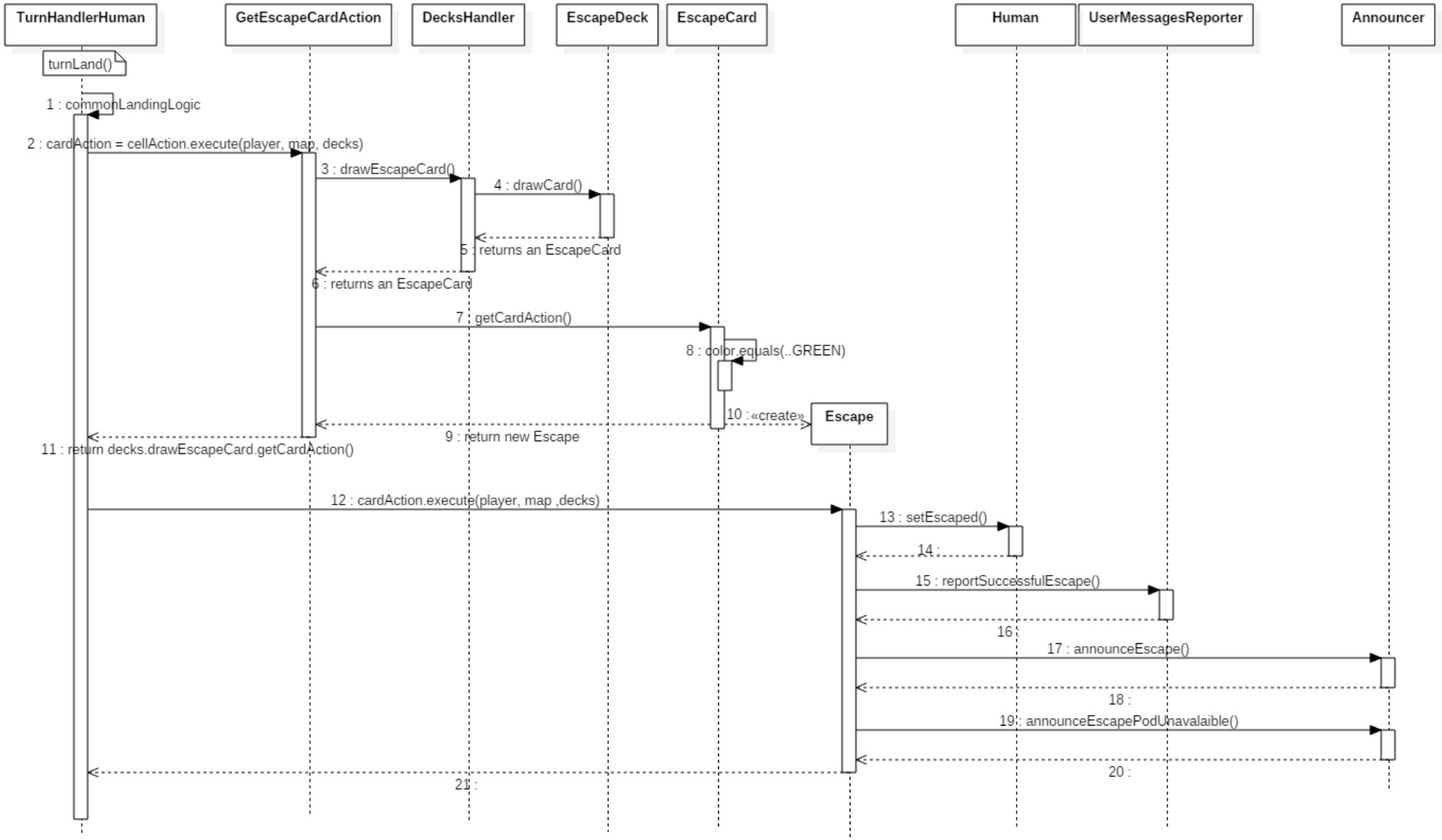

Illustration 13: Turno VI : Cella Scialuppa

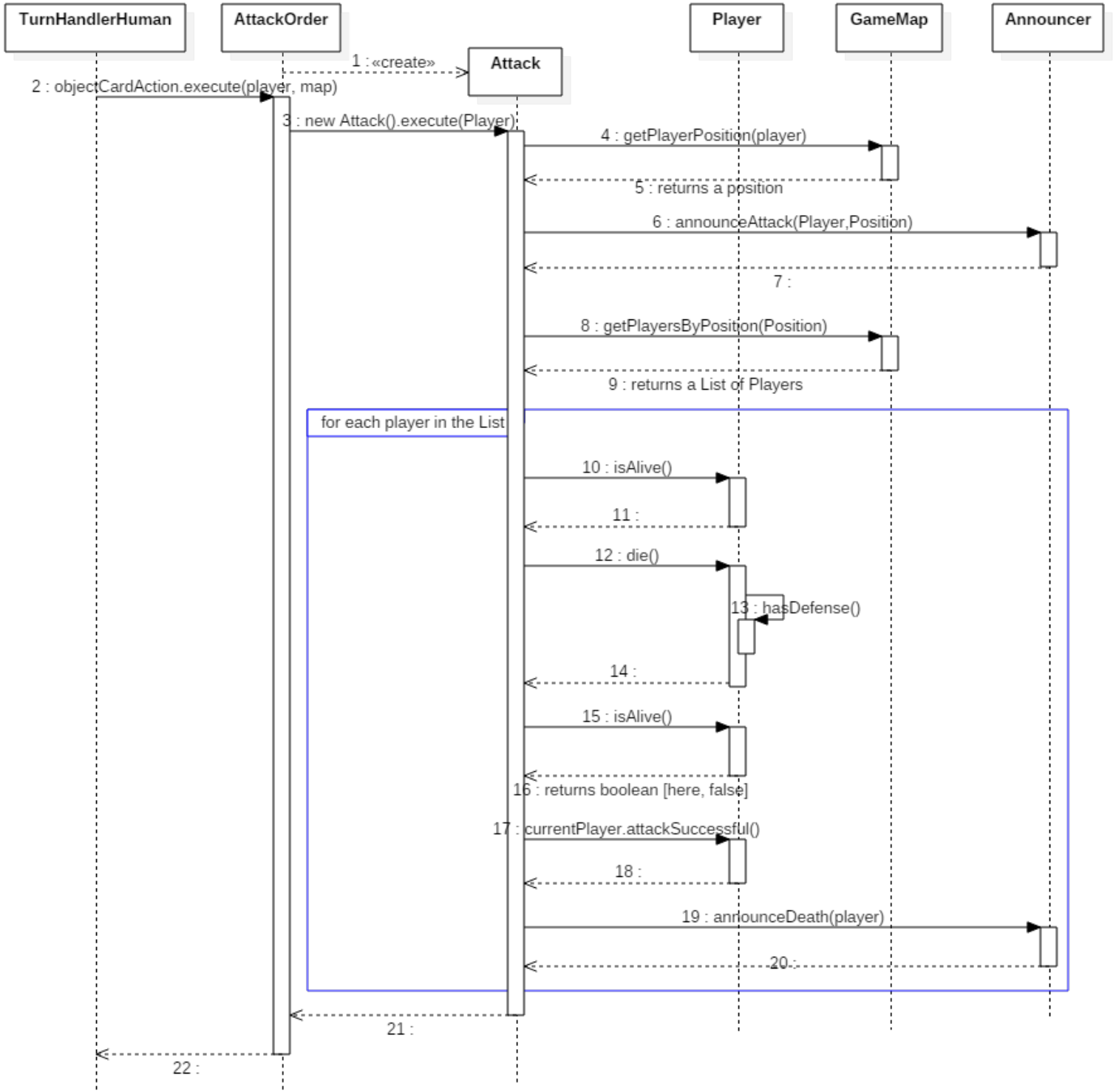

Illustration 14: ObjectCardAction : Attack

2.2 Controller: Completamento automatico del turno: timeout e override

L'utente può non rispondere, e fare scadere il tempo del suo turno.

In tal caso, il Server completerà il turno dell'utente automaticamente e passerà il turno al prossimo giocatore. L'applicazione infatti è in grado di eseguire un turno di gioco in maniera autonoma.

L'esecuzione di un turno è gestita da 2 thread, che sono attivi contemporaneamente:

- **Il thread dell'ExecutiveController**, che esegue le funzioni che definiscono le fasi del turno di un utente.

  Quando l'utente non invia l'input richiesto, questo thread è in attesa dell'arrivo di un input, cioè **si trova in una wait()** in MessagingChannelSocket o in MessagingChannelRMI.

- **Il thread del TimeController**, che gestisce il timeout di un turno, e indica al thread dell'ExecutiveController quando occorre incominciare il turno di un nuovo giocatore.

  In questo caso, **finisce di eseguire wait(TIMEOUT)**, e vede che il turno non è ancora completato.

  Mette in moto quindi il meccanismo dell'Override.

Meccanismo dell'Override, prima parte:

- Il thread n.2, quello del TimeController, va a invocare il metodo dello UserMessagesReporter fillInDefaultOnce(),
- il quale invoca il metodo di MessagingChannel overrideDefault(). Questo metodo consegna al thread n.1 la risposta di Default impostata per il tipo di richiesta fatta, e risveglia il thread n.1 con una notify().
- Il thread n.1 consegna quindi al controller del Server la risposta ricevuta, come se l'avesse inserita l'utente.

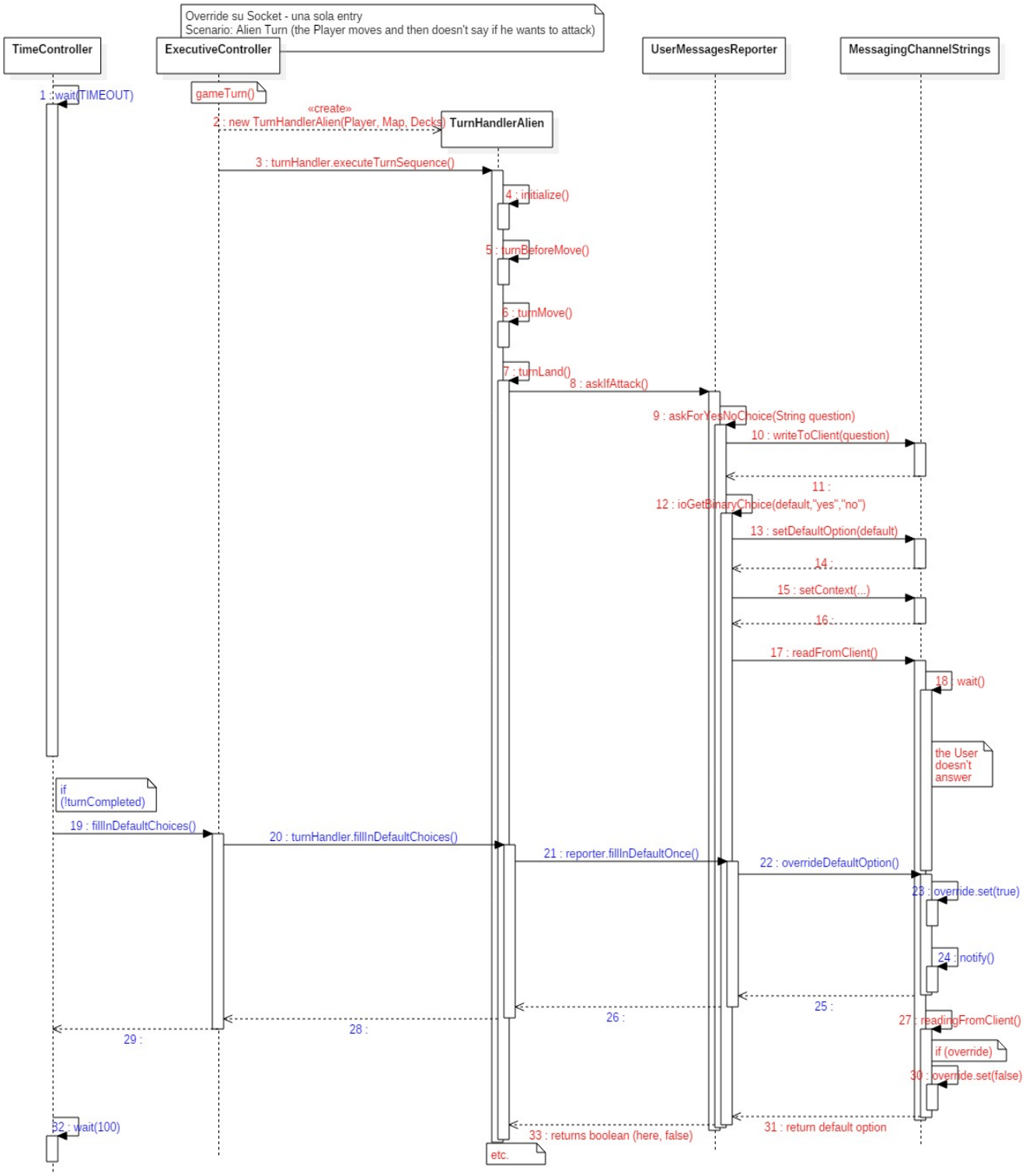

Illustration 15: Override di una sola entry (usando Socket)

Meccanismo dell'Override, seconda parte:

- Il thread n.2, dopo aver invocato nello UserMessagesReporter fillInDefaultOnce(), invoca fillInDefaultAlways(). Questo pone a true la variabile booleana automaticOverride.
- Se rimangono ancora altri input che l'utente avrebbe dovuto inserire, lo UserMessagesReporter vede che automaticOverride è a true e non richiede input all'utente tramite MessagingChannel; invece, ritorna subito un valore di default al TurnHandler .
    - Regole per determinare il valore di default:
        - Il giocatore non esegue alcuna azione volontaria, come giocare una carta oggetto o attaccare.
        - Se deve scegliere una posizione in cui fare un rumore, o giocare la carta Oggetto Luci, la scelta di default è la posizione in cui si trova.
        - Il giocatore non esegue alcun movimento, rimane nella cella in cui si trova.

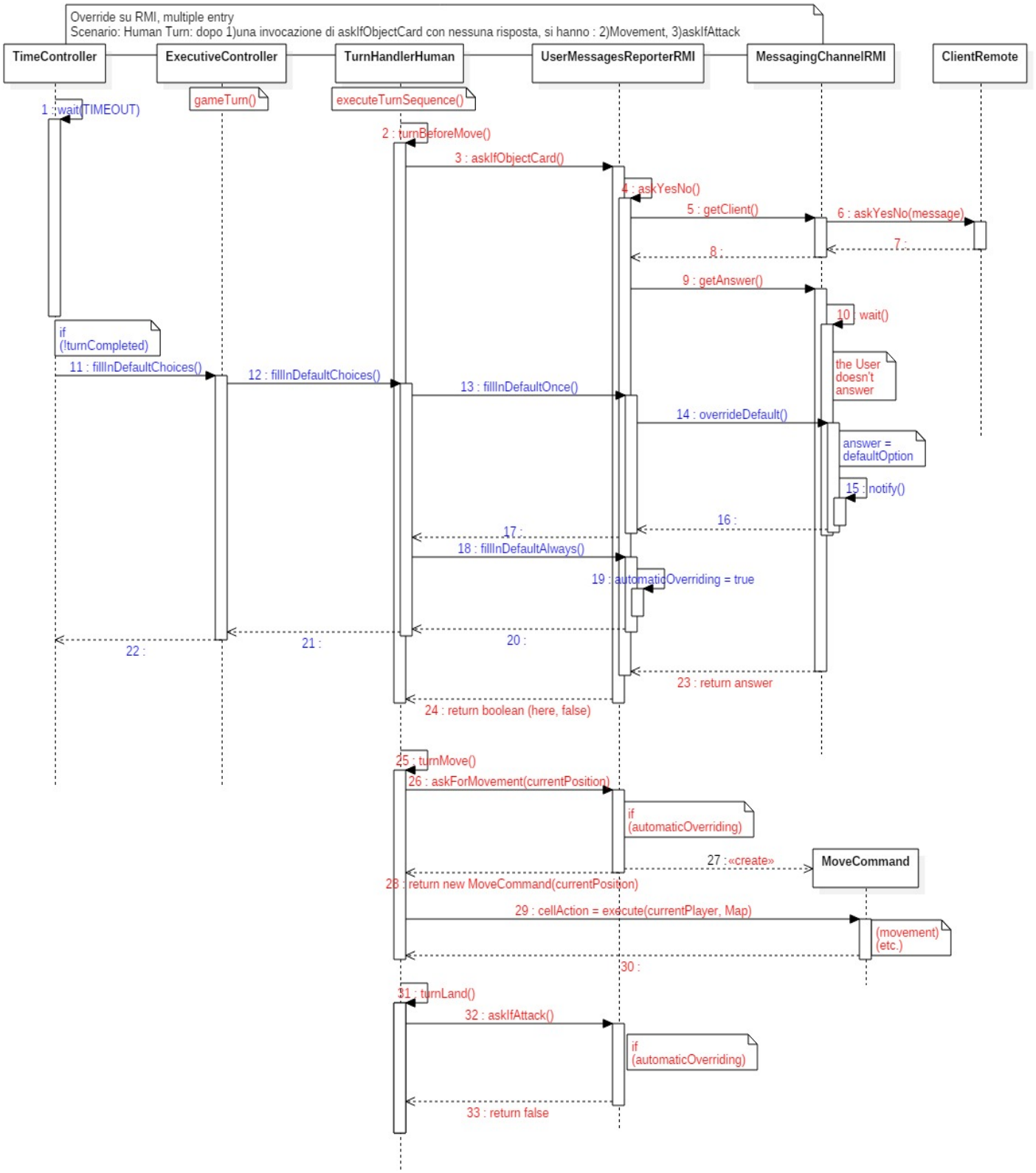

Illustration 16: Override di multiple entry (usando RMI)

2.3 Controller: Gestione di multiple partite (Master e GameMaster)

Le responsabilità della classe **Master** sono:

1. Tenere una lista dei GameMaster (che gestiscono ciascuno una partita), e crearne di nuovi o eliminarne se occorre.
2. Ricevere la notifica della connessione di un giocatore; in tal caso, controlla se esiste un GameMaster che ha posti liberi e non sta gestendo una partita già avviata.
   Una volta trovato (o creato) un GameMaster in grado di accogliere il giocatore, Master invoca le funzioni necessarie all'interno del GameMaster così che il giocatore sia aggiunto in quella partita.
3. Ricevere la notifica della disconnessione di un giocatore; in tal caso, passa la notifica al GameMaster a cui il giocatore apparteneva, che gestirà la disconnessione dell'utente dalla partita.
4. A seconda del tipo di connessione(Socket-RMI) usata dai giocatori che si connettono, associare un tipo di Announcer a un GameMaster.
   Announcer è la classe che serve a mandare messaggi in broadcast a tutti i giocatori di una partita che usano un certo tipo di connessione.
   (Per i Socket esiste AnnouncerStrings, per RMI esiste AnnouncerBroadcastRMI. Un GameMaster può avere entrambi gli Announcer associati se si connettono utenti che usano diversi tipi di connessione.)

nota: i metodi della classe Master da noi usati sono tutti metodi statici.

La classe **GameMaster** gestisce una singola partita. Le sue responsabilità sono:

- Gestire l'entrata di un nuovo utente, cioè:
  - Associare la sua interfaccia di comunicazione (MessagingChannelStrings o MesssagingChannelRMI) a un Player, che potrà essere Umano o Alieno in modo da avere le 2 squadre con un numero bilanciato di giocatori.
  - Ordinare alla mappa di piazzare il nuovo Player nella casella di partenza appropriata.
  - Ordinare all'Announcer di annunciare ai giocatori già presenti che un nuovo giocatore si è connesso.
- Gestire la disconnessione di un utente, cioè:
  - Eliminare il Player dalla lista di quelli nella partita.
  - Controllare la fase della partita in cui ci si trova: Prima dell'inizio, in corso, finita.
  - Nel caso la partita fosse in corso e rimane solo 1 giocatore, oppure un intero team si è disconnesso, terminare immediatamente la partita.

- Attendere il timeout definito (default: 60 secondi), prima di controllare se esistono le condizioni per fare iniziare una partita.
- Far partire i thread di ExecutiveController e TimeController, che gestiranno l'esecuzione dei turni di gioco e la conclusione del gioco.
- Mantenere un riferimento alla classe VictoryChecker, la quale quando il gioco è finito accederà alle informazioni di stato dei Players per determinare vincitori e perdenti.

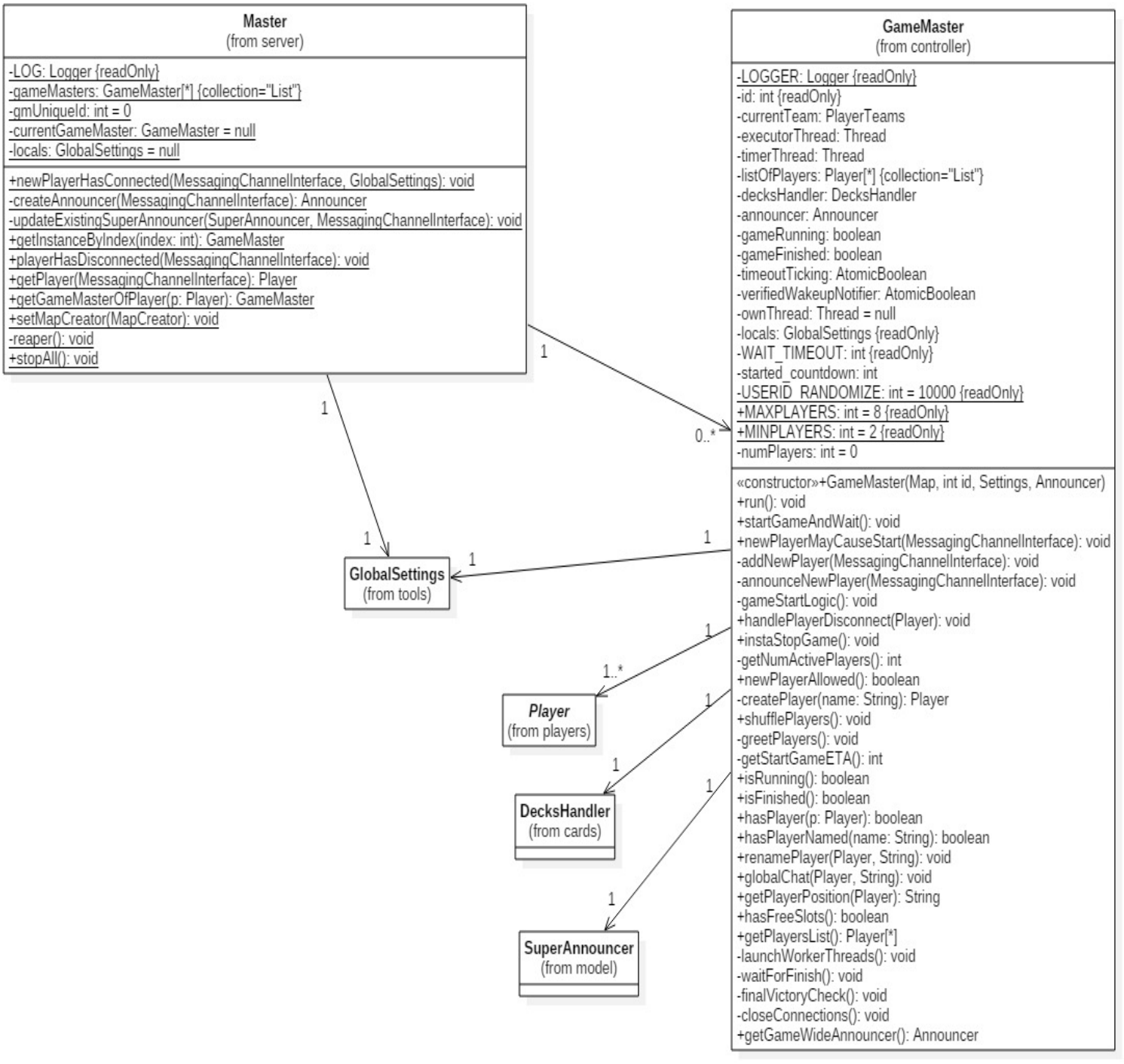

Illustration 17: Master&GameMaster : diagramma delle classi

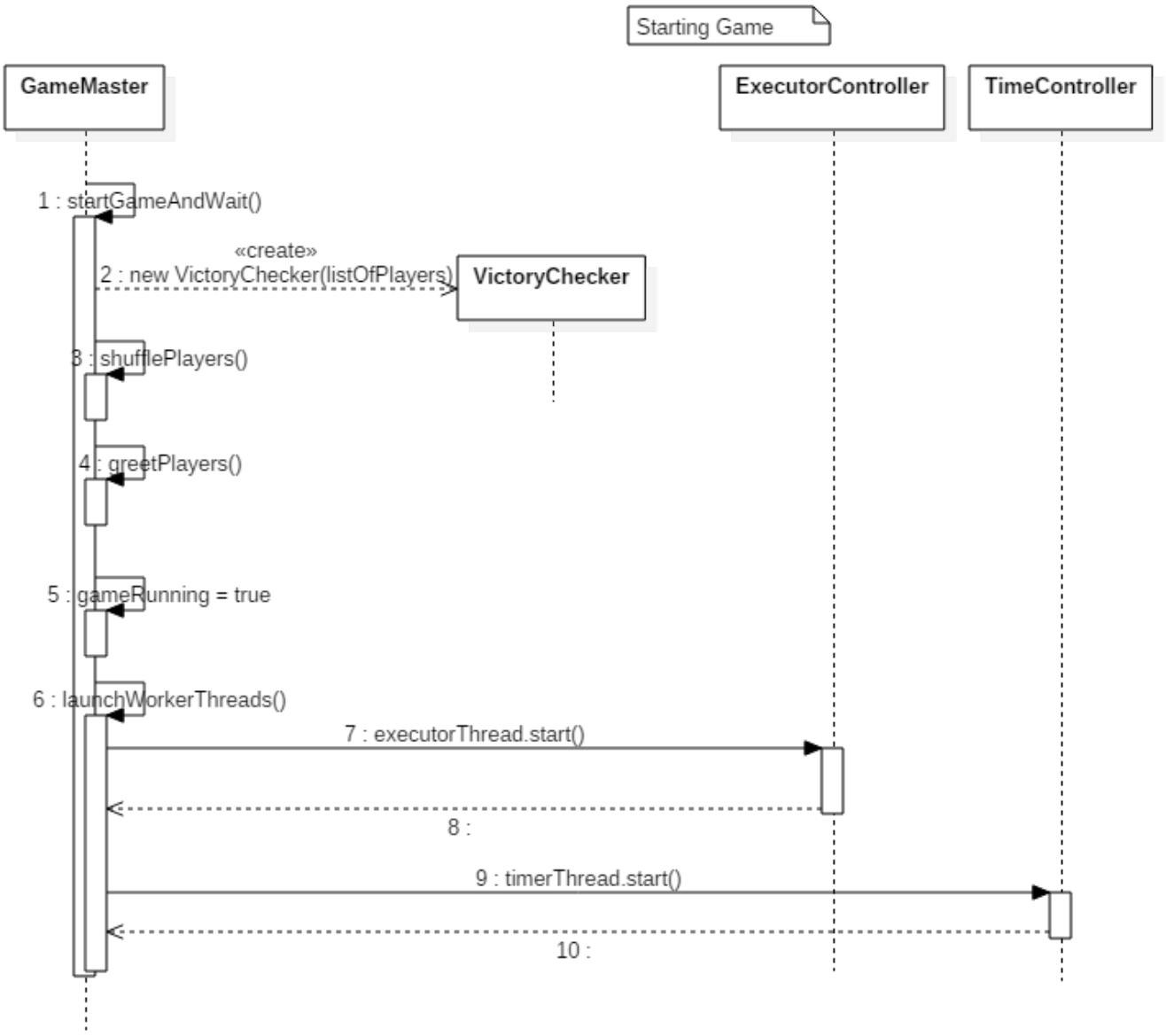

Illustration 18: GameMaster: Avvio di una partita

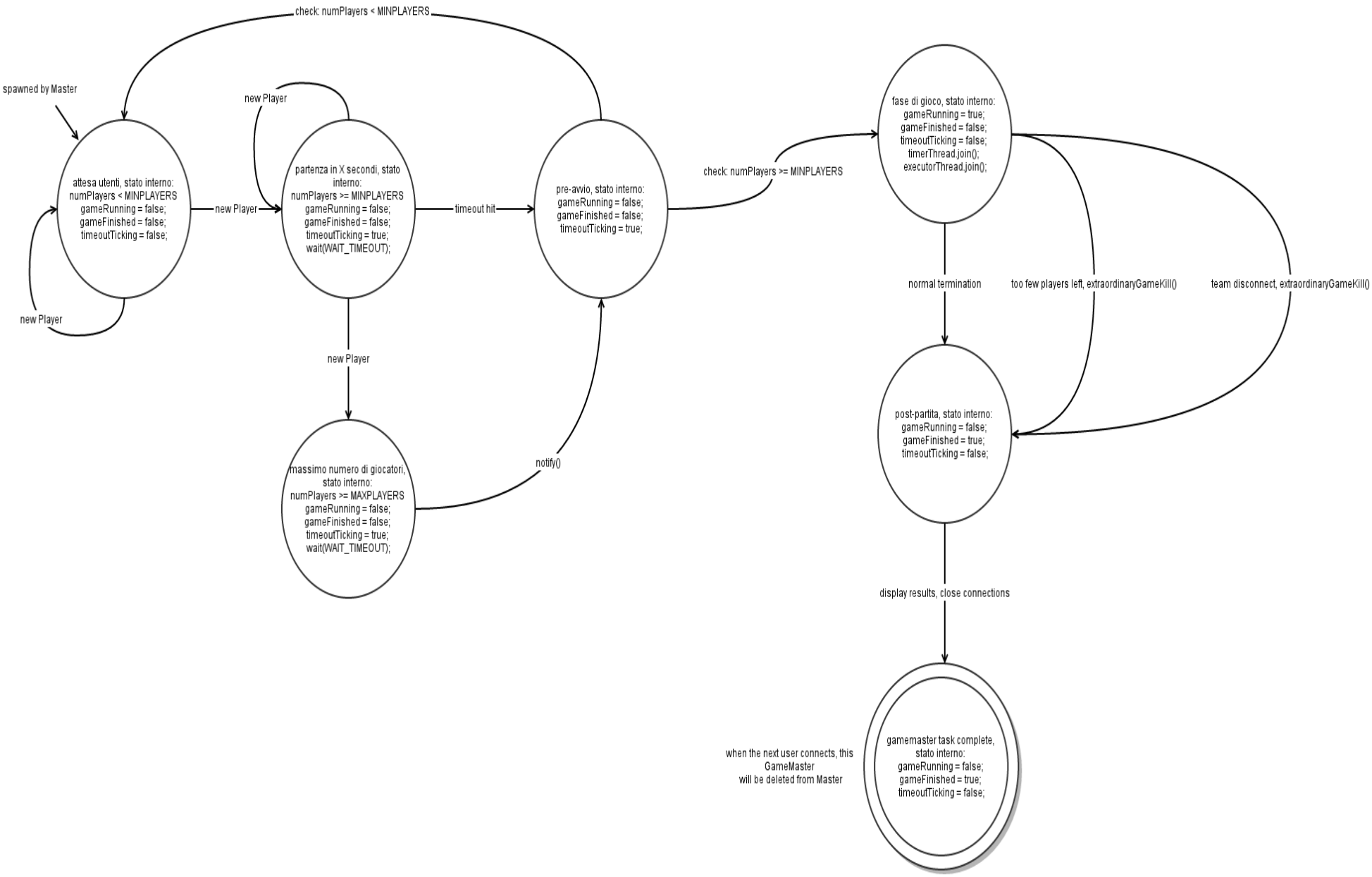

Illustration 19: GameMaster, diagramma di stato

2.4 Model: Caselle per la mappa

Ogni Casella nella mappa è identificata dalla sua posizione.
Le celle sono esagonali; per gestire le coordinate di una cella, e l'individuazione delle sue vicine, sfruttiamo la metodologia definita in : http://www.redblobgames.com/grids/hexagons

Per descrivere la posizione di una cella, usiamo 3 differenti sistemi di coordinate:

- **coordinate cubiche, es: (1,3,4).** Usando 3 assi di riferimento sul piano, è possibile descrivere ogni cella senza dover attuare degli offset. Ogni cella conosce la sua posizione in termini di coordinate cubiche.
  La classe PositionCubic contiene una posizione a 3 dimensioni, e alcuni metodi utili, come equals() o distanceFrom(PositionCubic other).

- **coordinate a 2 dimensioni, es: (1,2)**. Utilizza 2 assi di riferimento sul piano; sono utili principalmente perchè permettono la conversione in coordinate alfanumeriche.
  La classe Position2D contiene una posizione a 2 dimensioni.

- **coordinate alfanumeriche, es: (B,5).** Il modo 'classico' di riferirsi alle caselle, e anche quello che viene visualizzato graficamente sulla mappa.
  Sono rappresentate mediante stringhe, sulle quali viene effettuato un check di formato.

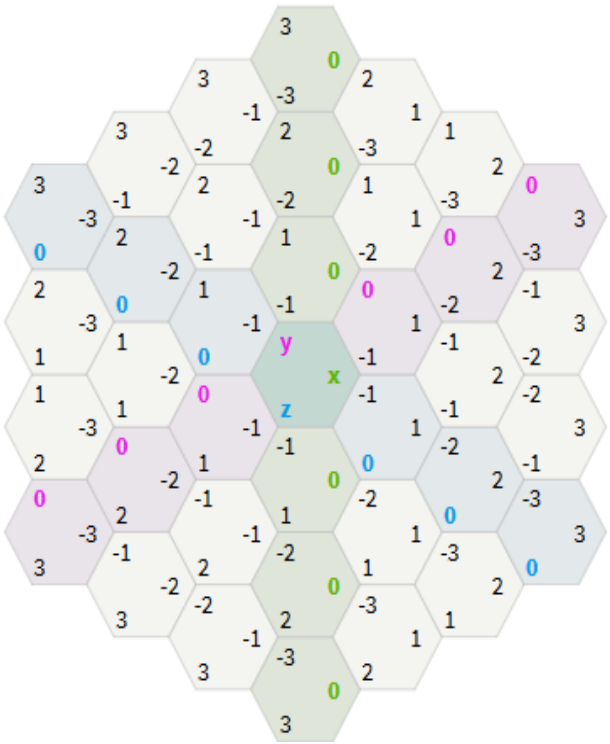

Illustration 20: Coordinate cubiche

Esempio di coordinate bidimensionali:

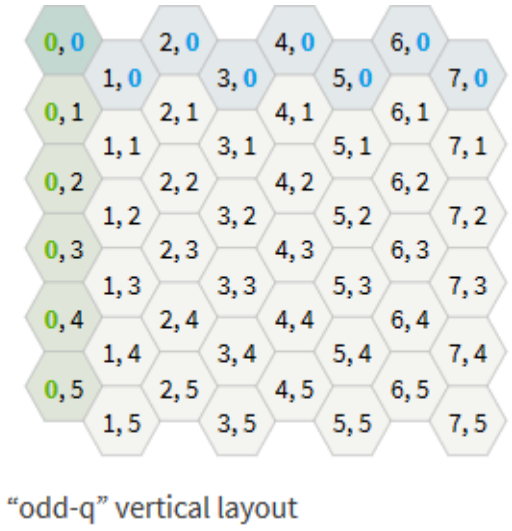

Illustration 21: Coordinate bidimensionali

Occorre una classe che possa convertire da un tipo di coordinate all'altro.
La classe **CoordinatesConverter** contiene una serie di metodi statici utili a questo scopo.

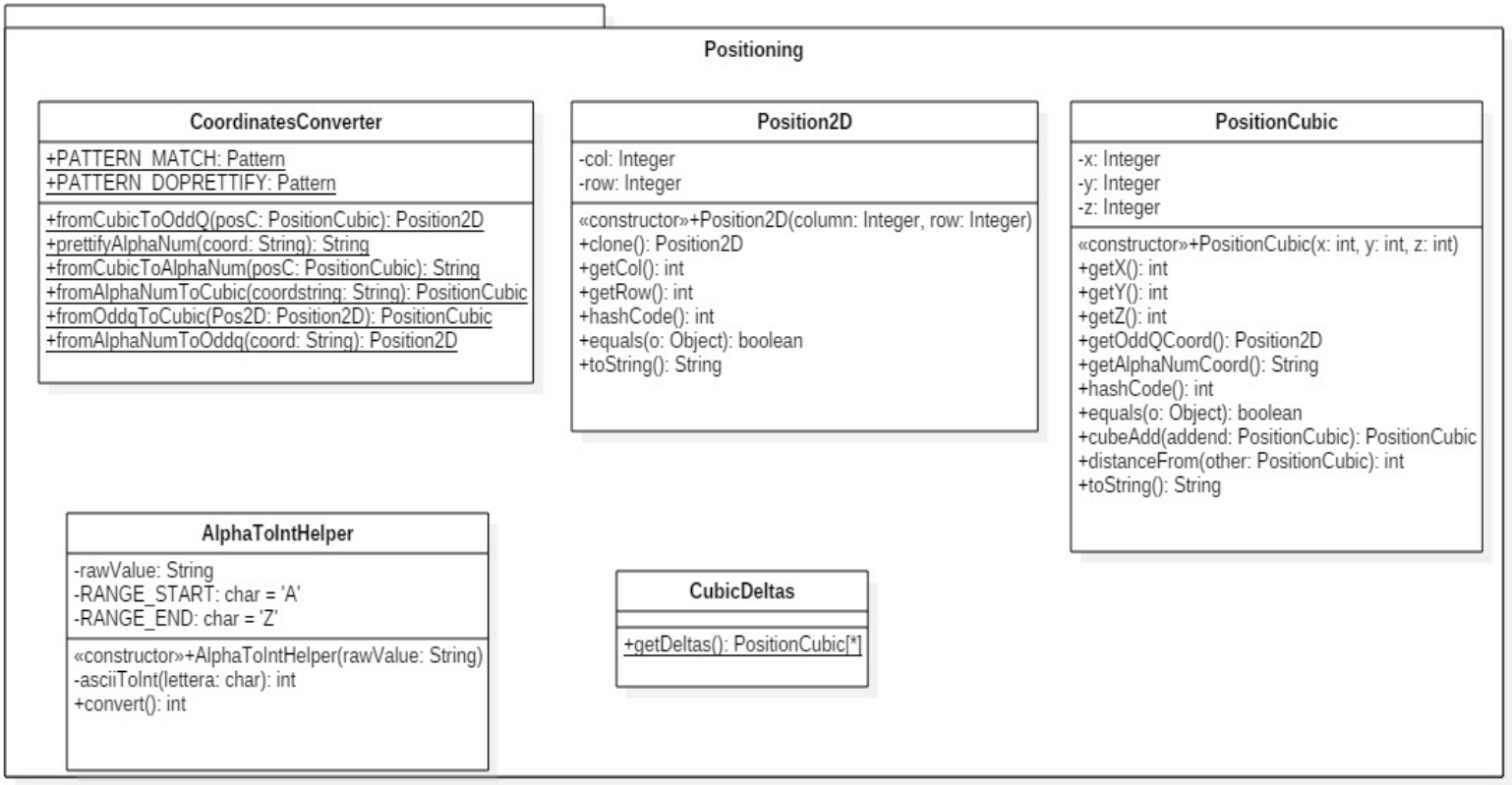

Illustration 22: Positioning, diagramma delle classi

Grazie al sistema di coordinate cubiche, una cella non deve contenere 6 puntatori ai propri vicini. Avendo una cella con posizione (x,y,z):
vicini = celle con posizioni tali che 2 variabili tra x, y e z differiscono una di +1 e un'altra di -1

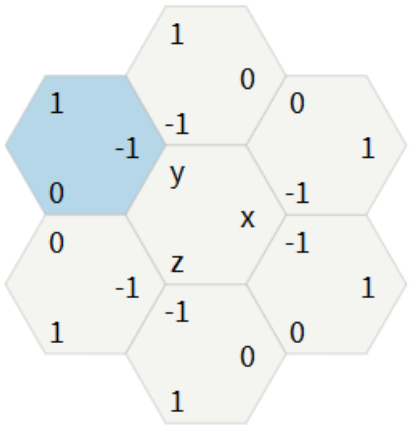

Illustration 23: Una casella e le sue vicine in coordinate cubiche

Una cella deve contenere l'informazione riguardante il tipo di settore: sicuro, pericoloso, partenza umani, partenza alieni, scialuppa.
Le sottoclassi di Cell sono:
SafeCell, DangerousCell, EscapeCell, StartingCell.

Una cella può essere la destinazione di un movimento o meno, anche a seconda del team del giocatore; la EscapeCell / cella scialuppa può essere destinazione di un movimento per un Umano, ma non per un Alieno. Si definisce perciò il metodo

public boolean canEnter(Player curPlayer)

che può ritornare **true** o **false,** a seconda del tipo di cella e del tipo di giocatore.

A seconda del tipo di cella, il giocatore dovrà eseguire diverse azioni:

- pescare una carta settore
- pescare una carta scialuppa
- niente

Seguendo il pattern Command, ogni azione è incapsulata in un oggetto, rispettivamente:

- **public class** GetSectorCardAction
- **public class** GetEscapeCardAction
- **public class** NoCellAction

Ogni cella ha quindi un metodo **public** CellAction getCellAction(), che ritorna uno degli oggetti citati.

2.5 Model: La mappa di gioco

2.5.a GameMap

Il file della mappa di gioco è in formato JSON, per diversi motivi:

- Human readability
- Diffusione del formato
- Semplicità delle API (org.json , sviuppata da Apache)

Il file JSON conterrà 2 oggetti:

- Una collezione di **attributi globali** della mappa (nome, dimensioni)
- Un'**array** di tutte le celle contenute, dove ogni cella è descritta da:
  - la propria posizione (in coordinate cubiche)
  - il proprio tipo

Gli elementi più importanti nella mappa di gioco:

- La hashmap **private** Map<String, Cell> **cells:**

  contiene le celle esistenti nella mappa, associando coordinate alfanumeriche a coordinate cubiche; questo permette di velocizzare la ricerca di una cella durante le fasi di gioco.

  es .<”A03”,Cell(2,3,5)>

- La hashmap **private** Map<Player, Cell> **playersPositions**;

  contiene la posizione attuale di ciascun giocatore

- Il metodo
  public CellAction move(PlayerActionInterface curPlayer, String destination)
  il quale:
  - invoca altri metodi per verificare se il movimento proposto è effettivamente eseguibile;
  - se no, lancia una delle possibili eccezioni (BadCoordinatesException, DestinationUnreachableException, CellNotExistsException, PlayerCanNotEnterException)
  - se sì, aggiorna la posizione del Player sulla mappa.

  Per il funzionamento dettagliato del metodo move, si rimanda ai diagrammi di sequenza della parte Controller: turno

- Il metodo
  public List<PlayerActionInterface> getPlayersByPosition(PositionCubic pos)
  che ritorna una lista con tutti i Player che si trovano in una data posizione (utile per il comando Attack o la carta oggetto Lights)

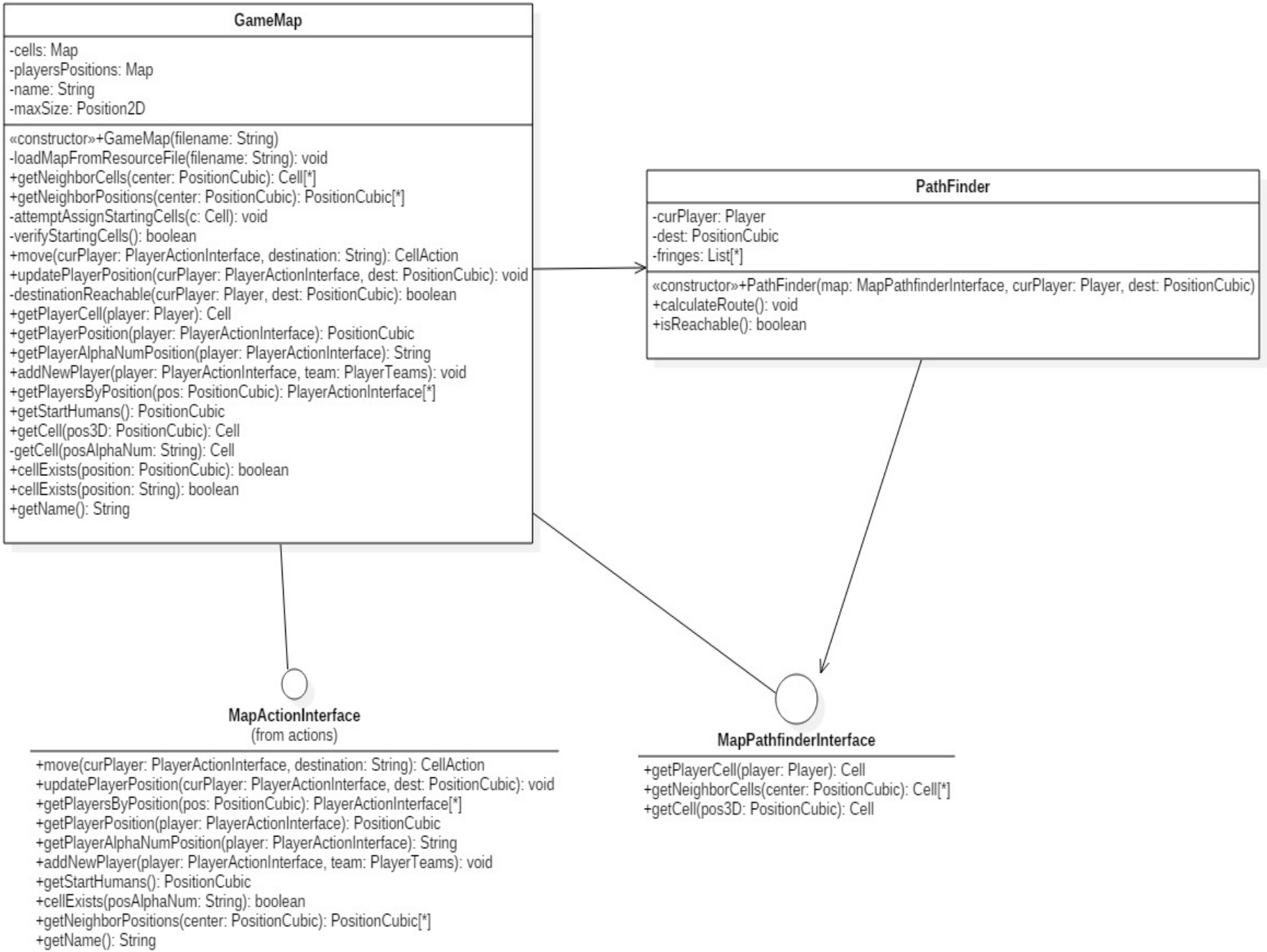

Illustration 24: GameMap: diagramma delle classi

2.5.b MapLoader

La classe MapLoader permette di caricare una mappa da file. (La mappa è memorizzata in un file JSON)

MapLoader svolge le seguenti funzioni:

- Offre un'iteratore sulle celle caricate da file

  ```
  public MapIterator iterator()
  ```

- Offre getters per i dati globali della mappa
  - ampiezza massima

    ```
    public Position2D getMapSize()
    ```
  - nome

    ```
    public String getMapName()
    ```

Il costruttore della classe è `public MapLoader(InputStream ingresso)` , dove *ingresso* sarà, nel nostro caso, un FileInputStream.

Tale InputStream è letto riga per riga e convertito in stringa. Questa stringa è immagazzinata da MapLoader, ed è usata dalla procedura decode() , che popola le variabili interne (dati globali della mappa e lista celle).

A MapLoader (che implementa Iterable) è affiancata la classe accessoria MapIterator (che implementa Iterator), con il solo scopo di iterare sulle celle precedentemente caricate.

Inoltre, per alleggerire la funzione decode() , è stata creata una classe supplementare denominata **CellGenerator**, che crea gli oggetti *Cell* veri e propri.

In CellGenerator avviene la mappatura:

“tipo cella”:String → “cella specializzata”:(? extends Cell).

Essa è implementata come una serie di if (equals).

2.5.c MapGenerator

Il file JSON prende l'input da terminale, con la tecnica del “rapid fire typing”.

A questo scopo è creato il package qdgenerator (Quick and Dirty Generator).

Il suo funzionamento è molto semplice: l'acquisizione delle celle avviene in un loop, governato da un pattern-state; è implementato uno stato per ogni tipo di cella.

```
while (current != null) {
        current = current.storeCell(ret, promptUppercase(current.getPromptText()));
}
```

Il ciclo comincia leggendo le celle *Safe*, promptUppercase(String) presenta un messaggio, e chiede le coordinate (automaticamente convertite in UpperCase); una volta lette le coordinate, storeCell() appende la cella letta alla List *ret* come side-effect, e ritorna lo stato successivo (ancor *SafeCellAcquirer*).

Se l'utente ha inserito un punto ".", nessuna cella viene aggiunta, ed il metodo ritorna lo stato *DangerousCellAcquirer*; al termine della catena di celle da acquisire, il metodo ritorna null ed il ciclo termina.

2.6 Model: I mazzi e le carte

La classe **Deck**, rappresenta un mazzo di carte generico e il suo comportamento.

La struttura dati usata è una ArrayList;

Vi è un counter, che tiene traccia dell'estrazione delle carte (la performance è migliore rispetto a eliminare gli oggetti, e ricrearli quando si deve rimescolare il mazzo).

Le sottoclassi

- **EscapeDeck,**
- **SectorDeck,**
- **ObjectDeck**

ereditano il comportamento della sovraclasse, e in più vi aggiungono un costruttore che crea i mazzi con i tipi di carte specificati.

(nota: quando i mazzi vengono creati, si esegue anche la funzione **shuffleDeck()** per mescolarli)

Tutti i diversi tipi di carte implementano l'interfaccia Card, per potere essere messe in un Deck.

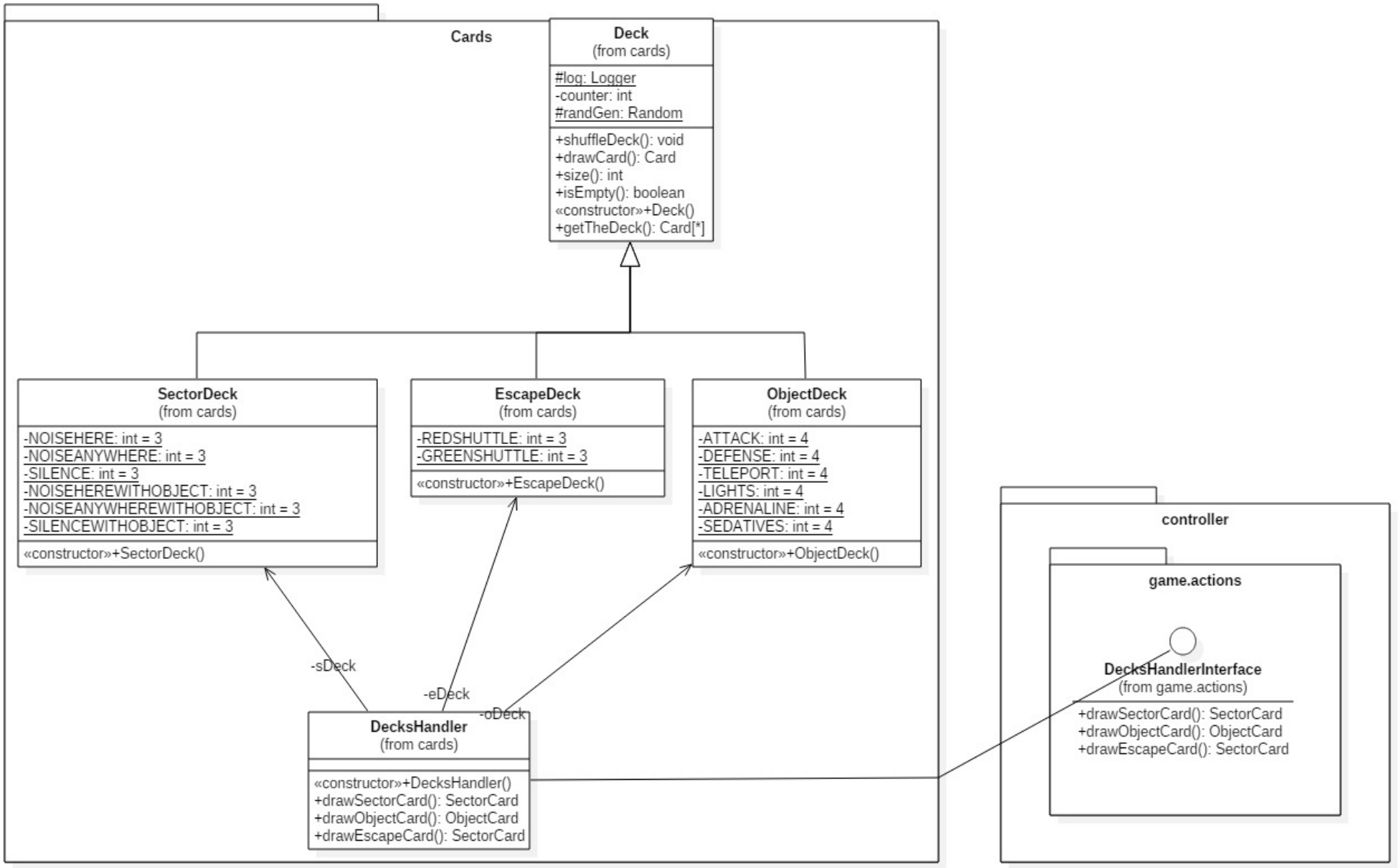

Illustration 25: The Decks

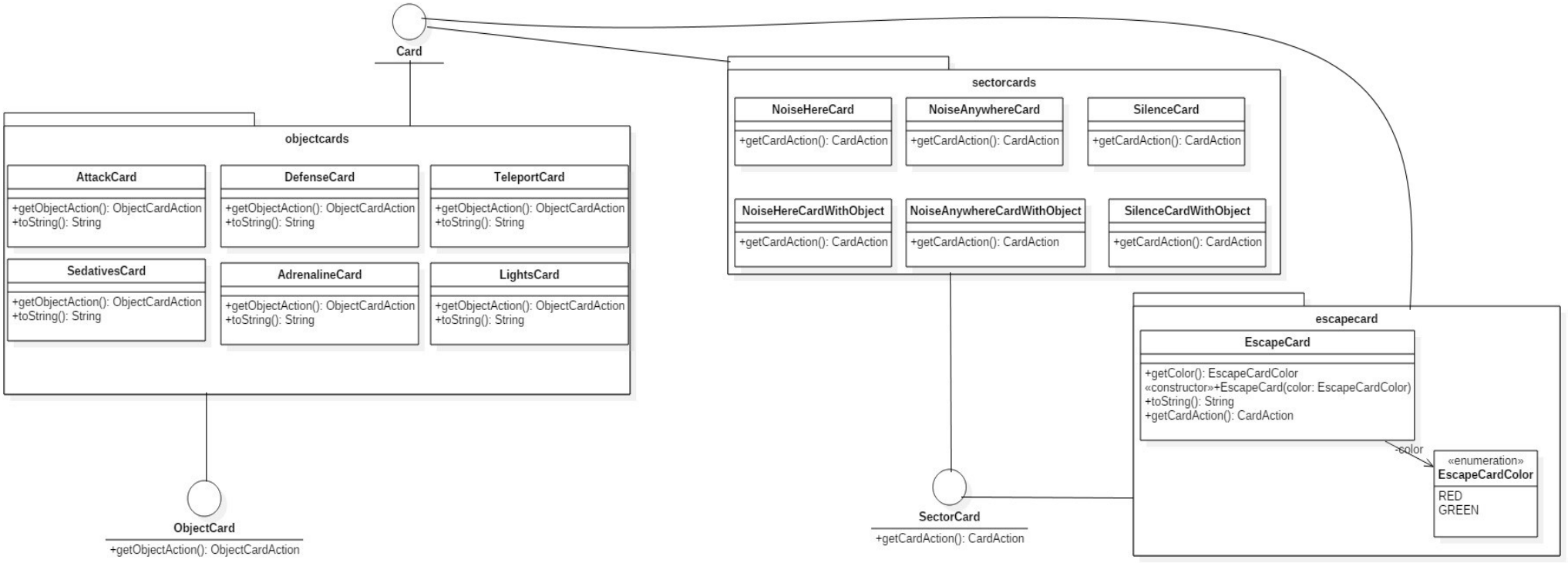

Illustration 26: The Cards

2.7 Model: I personaggi

La classe **Player** serve a incapsulare informazioni di stato sul personaggio, che vengono memorizzate sia per un Umano sia per un Alieno, quali:

- se il personaggio è ancora vivo o è stato eliminato
- se l'utente di questo personaggio si è disconnesso dalla partita (il turno di questo personaggio viene saltato)
- se il personaggio ha attaccato nel suo turno corrente (in tal caso, non deve pescare carte settore)
- se personaggio ha già effettuato il movimento nel suo turno corrente

La classe Player contiene inoltre un riferimento alla classe **Hand**, che gestisce la mano di carte possedute dal giocatore (aggiungere / rimuovere carte, controllare se si eccede il numero massimo di carte che si possono avere).

La classe **Human** contiene informazioni di stato e metodi di modifica dello stato specifici per l'Umano, in particolare:

- il range di movimento corrente, modificabile tramite setAdrenaline()
- se l'Umano è sotto l'effetto di Sedativi (e quindi non deve pescare carte settore)
- se l'Umano è già scappato (e quindi non deve fare niente durante il suo turno, e alla fine sarà incluso tra i winners)

La classe **Alien** contiene informazioni di stato e metodi di modifica dello stato specifici per l'Alieno, in particolare:

- il range di movimento corrente, che viene aumentato quando l'Alieno ha mangiato qualcuno mediante il metodo attackEndedSuccessfully()

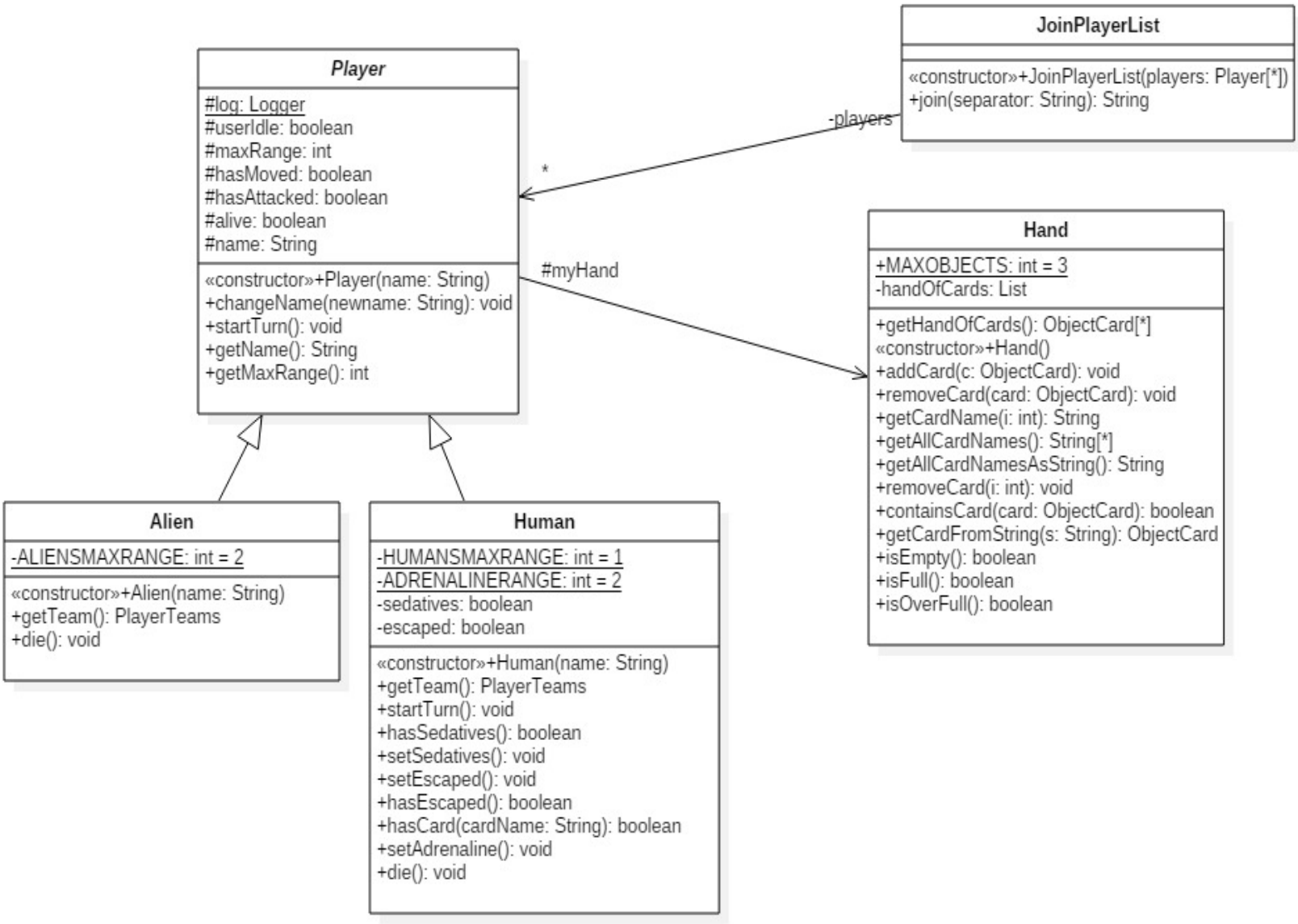

Illustration 27: The Players

3 Il Client

3.1 Client con Command Line Interface

Rispetto a un qualsiasi client non specifico con linea di comando, come “nc” in Linux, il nostro Client-CLI ha 2 feature in più:

- Mostrare il prompt che indica quale tipo di input il client può / deve inserire, come ad esempio:
  - `(Someone else's turn, free-action)>`
  - `(My turn, coordinates)>`
  - `(My turn, yes/no)>`
- Effettuare un controllo di formato sulle stringhe che l'utente invia, a seconda della fase del turno in cui ci si trova.

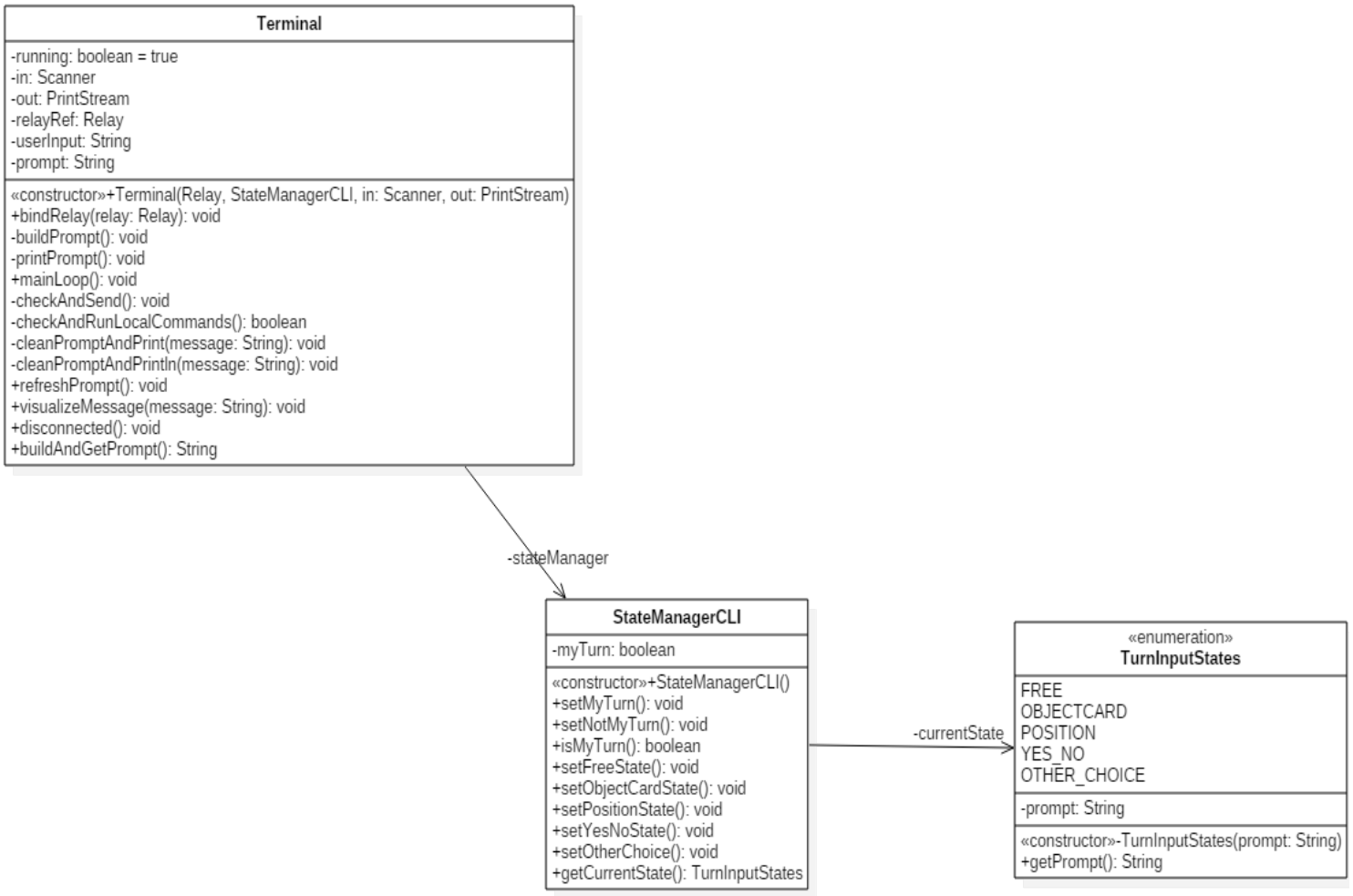

Illustration 28: Client-View-CLI

3.2 Client con Graphical User Interface (Swing)

Si riporta qui la struttura del Client che utilizza la GUI.

Viene impiegato il Pattern Model-View-Controller per gestire il fatto che il Client Swing è più complesso del Client CLI (non basta stampare delle stringhe di testo per comunicare), e ha bisogno di un Model che rappresenti almeno una parte dello stato interno del gioco.

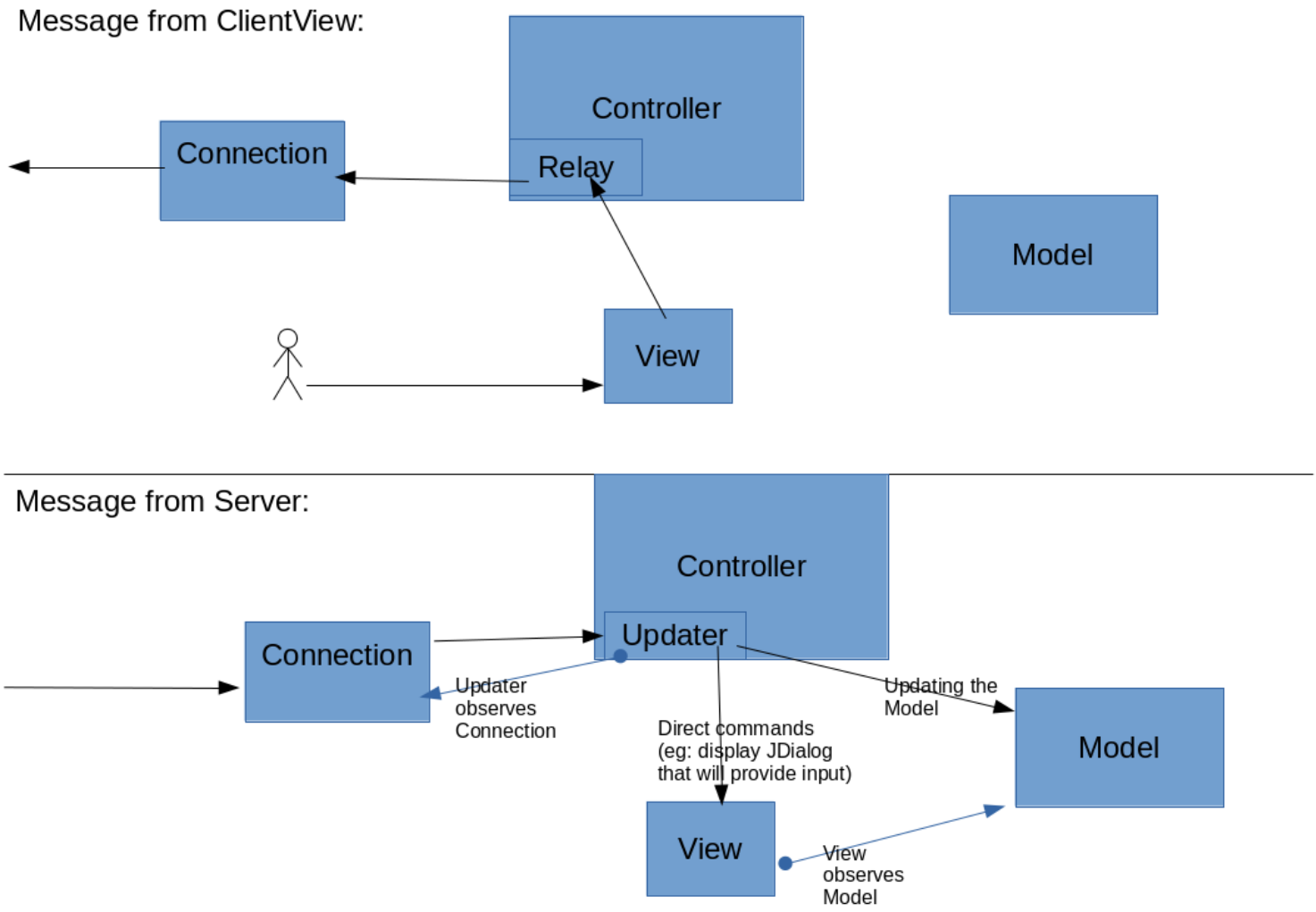

Illustration 29: Struttura MVC del Client

Utilizzando la comunicazione via Socket, le stringhe di testo inviare dal Server vengono processate nel Client da UpdaterSwing, che controlla se corrispondono a uno dei Pattern definiti. Se la stringa ricevuta corrisponde a un dato Pattern, viene chiamato il metodo corrispondente.

Ad esempio, ricevere la stringa `“please move your character on the map”` causa l'invocazione del metodo `askForMovement()`, che richiede all'utente di cliccare sulla mappa per specificare la destinazione del movimento.

Utilizzando la comunicazione via RMI, il metodo in UpdaterSwing fa parte dell'interfaccia ClientRemoteInterface, e viene invocato direttamente da UserMessagesReporterRMI o da AnnouncerBroadcastRMI, che forniscono gli opportuni parametri.

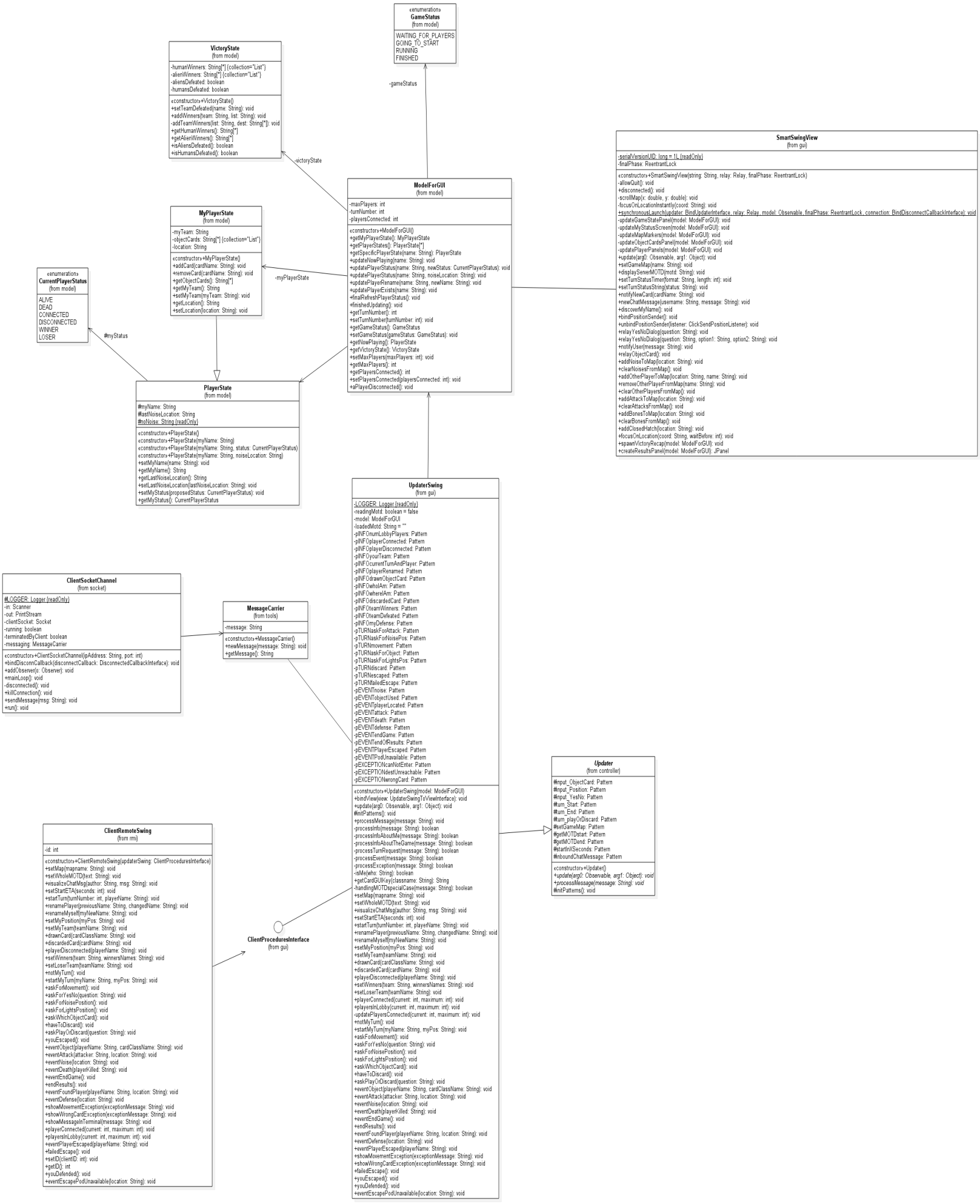

Illustration 30: Swing Client diagramma delle classi principali

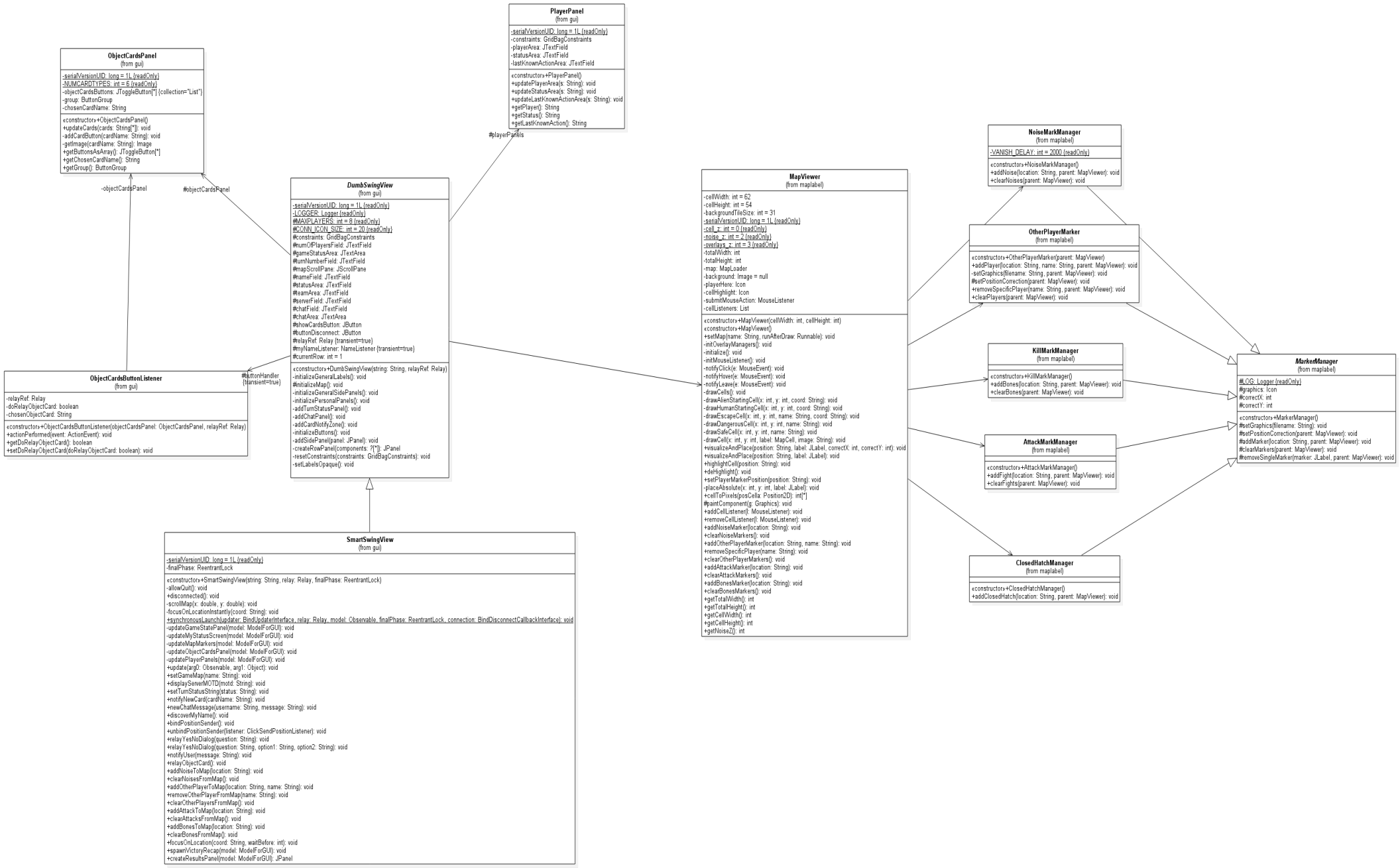

Illustration 31: Client Swing View, diagramma delle classi

4 Comunicazione

4.1 Socket

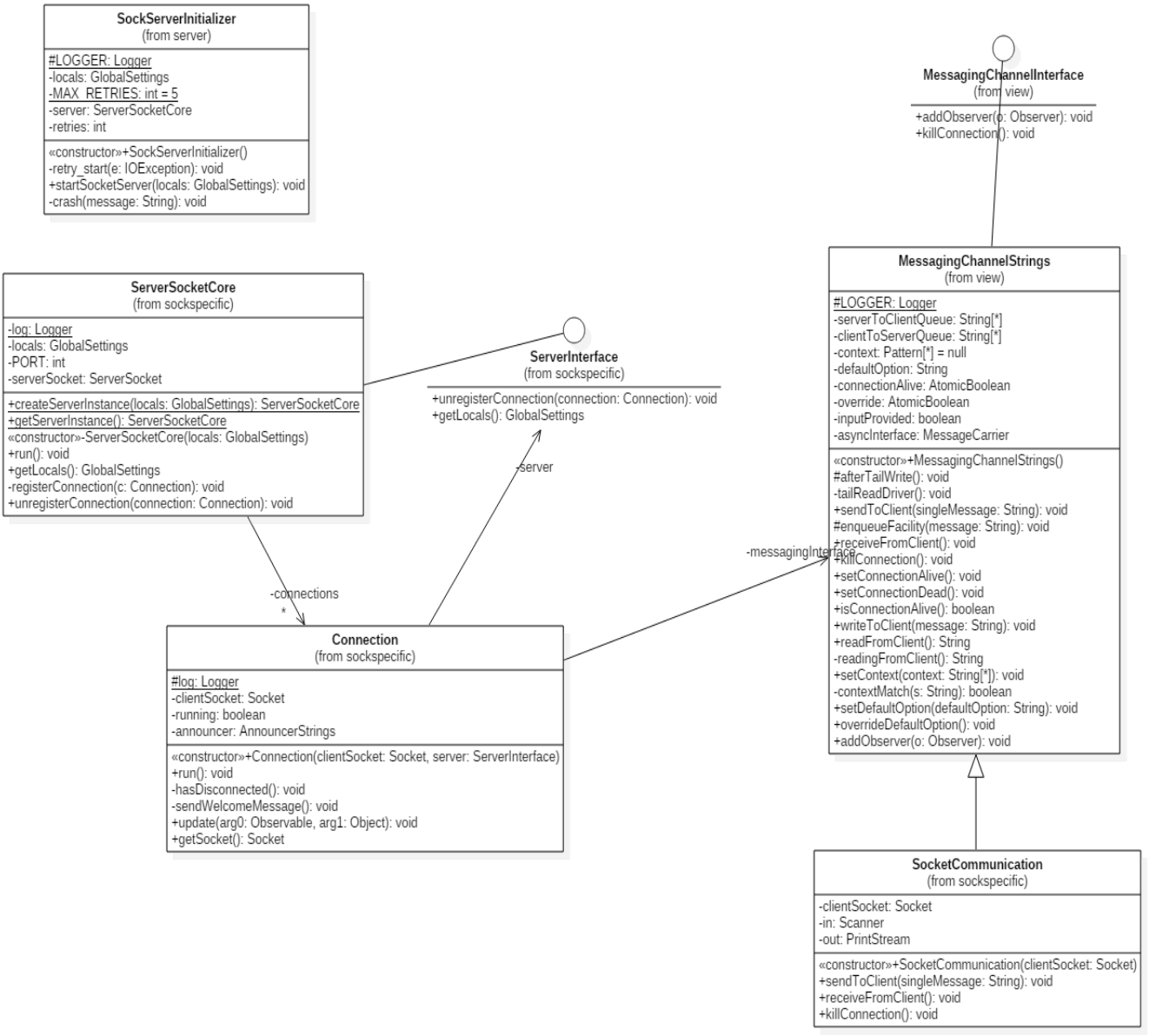

Illustration 32: Socket – classi del Server

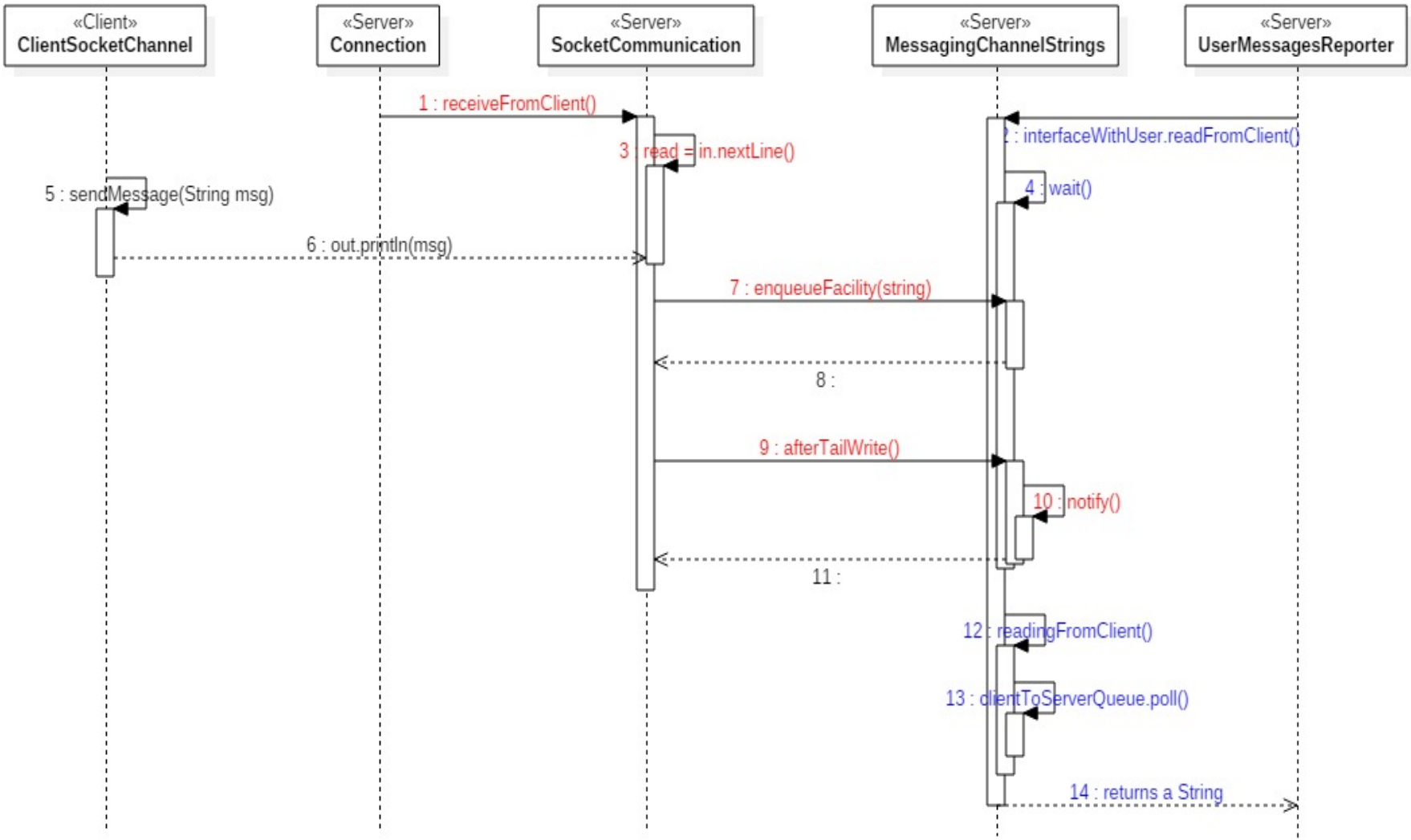

Illustration 33: Socket, comunicazione da Client a Server

4.2 RMI

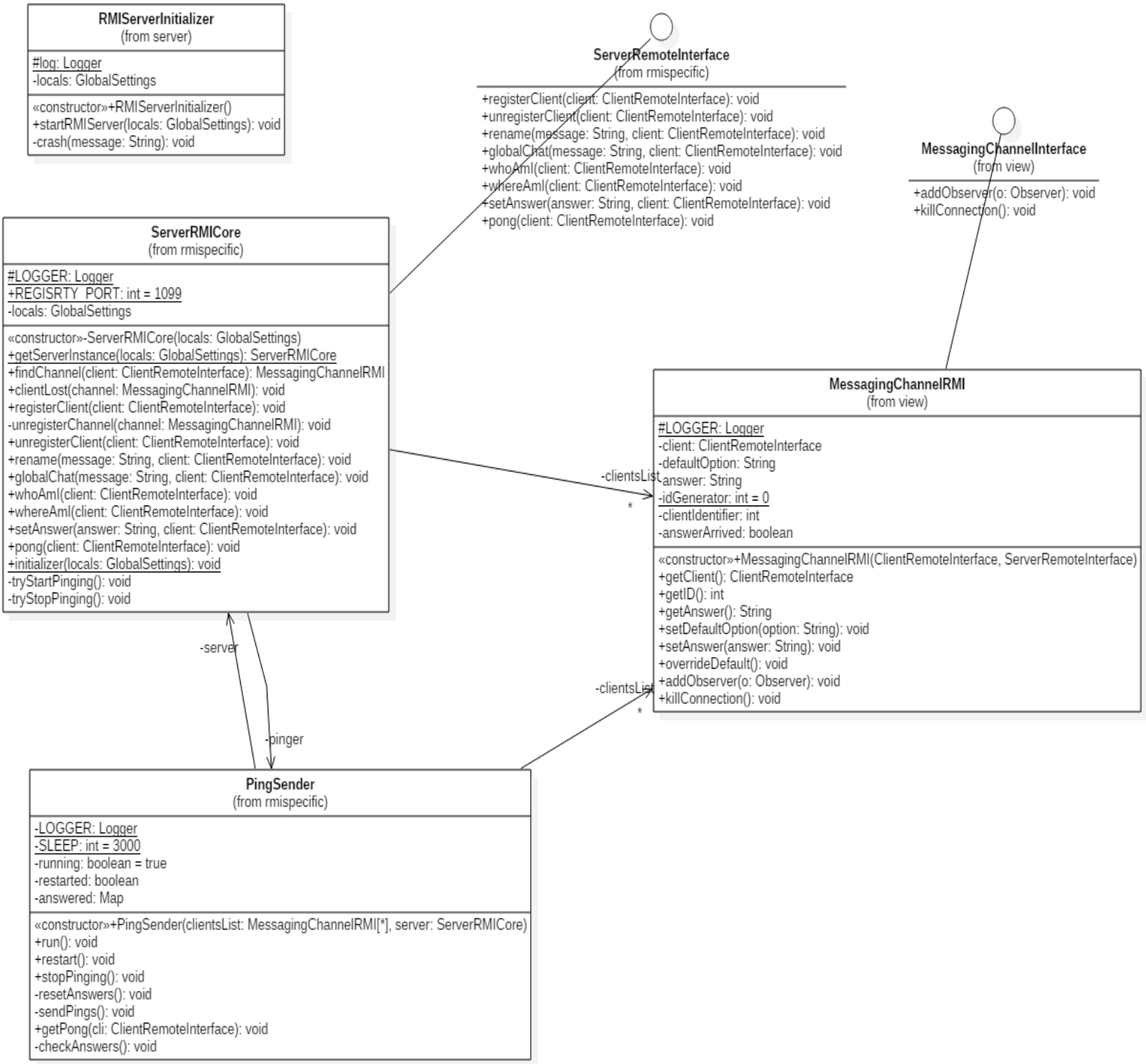

Illustration 34: RMI - classi del Server

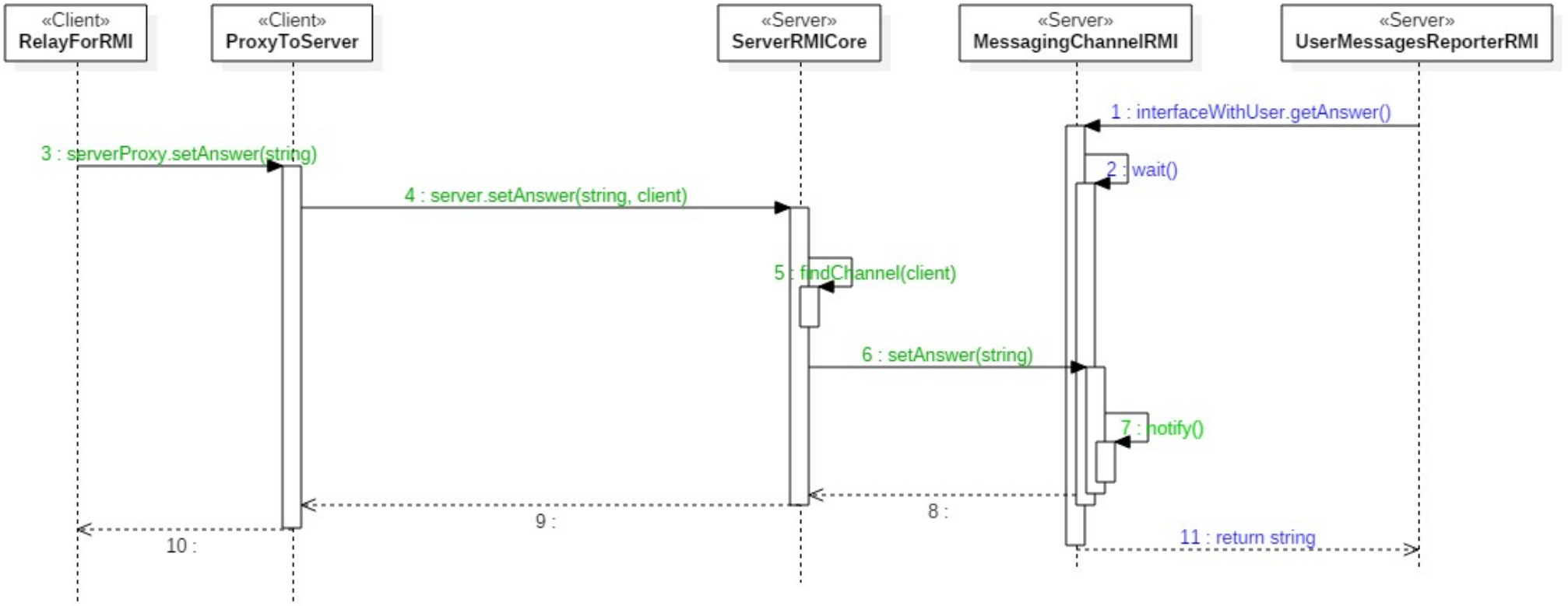

Illustration 35: RMI - comunicazione da Client a Server

5 Design patterns utilizzati

5.1 Model-View-Controller

Model: Contiene informazioni sullo stato dell'applicazione, e le entità che ne fanno parte.

Controller: Contiene la logica dell'applicazione, in base all'input ricevuto effettua modifiche sullo stato (sul Model).

View: Gestisce l'acquisizione dell'input (che sia il Server a riceverlo dall'utente, o il Client dal Server)

Il pattern viene impiegato sia nella struttura del Server che in quella del client.

(v. capitolo 1.1.Perchè scegliere un doppio Model-View-Controller)

5.2 Command

Incapsula un'azione da eseguire, ponendola all'interno di un oggetto dedicato.

Partecipanti:

- **Command :** dichiara un'interfaccia per l'esecuzione di una operazione
- **ConcreteCommand :** implementa l'interfaccia Command per eseguire una operazione; contiene il riferimento al Receiver
- **Receiver :** contiene le funzioni per eseguire le operazioni ordinate dal ConcreteCommand
- **Client:** Crea un ConcreteCommand e gli fornisce un riferimento al receiver
- **Invoker :** Invoca i metodi del ConcreteCommand, ordinandogli di eseguire l'azione richiesta sul Receiver

Il pattern costituisce il core della logica applicativa del turno di gioco; i ruoli delle classi nel pattern sono i seguenti:

- **Command**: le interfacce:
  - CellAction
  - CardAction
  - playerCommand

- **ConcreteCommand:**
  - implementano CellAction :
    GetEscapeCardAction, GetSectorCardAction, NoCellAction
  - implementano CardAction :

NoiseAnywhere, NoiseAnywhereWithObject, NoiseHere, NoiseHereWithObject, Silence, SilenceWithObject, Escape, CanNotEscape, DrawObjectCard

- implementano ObjectCardAction:
  Adrenaline, Sedatives, Lights, Teleport, AttackOrder, Defense

- **Invoker:**
  - TurnHandler, TurnHandlerAlien, TurnHandlerHuman

- **Receiver:**
  - Il Player, e le sue sottoclassi Human e Alien, che contengono i metodi setter per modificare il loro stato
  - La GameMap, che contiene le posizioni dei Player

- **Client**
  - UserMessagesReporter, che crea il comando Movecommand e lo ritorna all'Invoker (TurnHandler)
  - TurnHandler, che crea il comando DrawObjectCard
  - Le SectorCard, che creano CardActions e le ritornano
  - Le ObjectCard, che creano ObjectCardActions e le ritornano al TurnHandler

5.3 Observer – Observable

Definisce un legame tra l'oggetto Publisher e uno o più oggetti Subscriber. Quando il Publisher subisce una modifica del suo stato, tutti i Subscriber ricevono la notifica e possono compiere operazioni di aggiornamento.

Partecipanti:

- **Subject:** Provvede un'interfaccia per aggiungere o escludere degli oggetti Observer
- **Observer :** definisce un'interfaccia per effettuare l'aggiornamento quando lo stato del Subject cambia.
- **ConcreteSubject :** implementa l'interfaccia Subject. Contiene le informazioni di interesse per i ConcreteObserver, e manda una notifica ai ConcreteObserver quando lo stato cambia.
- **ConcreteObserver:** mantiene un riferimento ai ConcreteSubject; implementa l'interfaccia Observer, per aggiornarsi quando lo stato dei ConcreteSubject cambia.

I casi in cui viene impiegato il pattern Observer-Observable sono:

Caso 1:

Nel Server, il meccanismo Observer-Observable viene usato per gestire lo scambio di messaggi che

sono asincroni rispetto alle richieste fatte dal Server al Client in un dato momento del gioco.
In particolare, i messaggi asincroni gestiti in questo modo sono:

- cambiare il proprio nome
- mandare un messaggio di chat
- richiedere la propria posizione attuale
- richiedere il proprio nome attuale.

Nel Server, la classe AsyncUserListener richiede di osservare la classe MessagingChannelStrings, la quale aggiunge AsyncUserListener come Observer non a se stessa, ma a un oggetto della classe MessageCarrier.

Quando arriva un messaggio dal Client al Server attraverso Socket, **MessagingChannelStrings** lo pone nella coda da Server a Client, e poi ordina a **MessageCarrier** di eseguire notifyObservers().

Quando arriva la notifica da MessageCarrier, **AsyncUserListener** controlla se il messaggio arrivato è uno di quelli che devono avere una risposta asincrona immediata, e in tal caso chiama i metodi di altre classi (GameMaster, UserMessagesReporter) che inviano la risposta.

- **Subject :**
  - interfaccia `MessagingChannelInterface,` con i metodi `public void addObserver(Observer o)` e `public void killConnection();`
  - interfaccia predefinita Observable
- **Observer:** interfaccia predefinita Observer
- **ConcreteSubject:**
  - `public class MessagingChannelStrings extends Observable implements MessagingChannelInterface`
  - `public class MessageCarrier extends Observable`
- **ConcreteObserver:** `public class AsyncUserListener implements Observer`

Caso 2:

Nel Server, quando il Controller chiama **AnnouncerStrings** per mandare un messaggio in broadcast ai Client connessi tramite Socket, AnnouncerStrings crea il messaggio appropriato a seconda del metodo chiamato, lo conserva nella sua variabile `String message` e esegue `notifyObservers();`

Tutti gli oggetti di tipo **Connection** associati ai vari Client vengono notificati; eseguono quindi `announcerStrings.getMessage()` e inviano il messaggio ai Client sul Socket.

- **Subject:** interfaccia predefinita Observable

- **Observer :**interfaccia predefinita Observer
- **ConcreteSubject :** `public class AnnouncerStrings extends Observable`
- **ConcreteObserver :** `public class Connection implements Observer`

Caso 3:

Nel Client, la classe **Updater**, all'interno del Controller, osserva la classe **ClientSocketChannel** nella Connection, che riceve i messaggi (le stringhe) inviate dal Server.

Quando un messaggio arriva, l'Updater viene notificato, e invoca il metodo processMessage(message); questo metodo provocherà dei cambiamenti nella View e/o nel Model, a seconda del messaggio

- **Subject:** interfaccia predefinita Observable
- **Observer :**interfaccia predefinita Observer
- **ConcreteSubject :** `public class ClientSocketChannel extends Observable`
- **ConcreteObserver :** `public abstract class Updater implements Observer`

Caso 4:

Nel Client, la classe **SmartSwingView**, all'interno del package view.gui, osserva la classe **ModelForGUI** nel package model.

Il Controller (più precisamente, la classe Updater) effettua una serie di modifiche in ModelForGUI, dopodichè invoca il metodo :

`public void finishedUpdating() {...notifyObservers();},` e in risposta la SwingView aggiorna i suoi pannelli estraendo i dati del ModelForGUI.

- **Subject:** interfaccia predefinita Observable
- **Observer :** interfaccia predefinita Observer
- **ConcreteSubject :** `public class ModelForGUI extends Observable`
- **ConcreteObserver:** `public class SmartSwingView extends DumbSwingView implements Observer`

5.4 Singleton

Assicura che una classe possa istanziare un solo oggetto, e provvede un unico punto di accesso.

Partecipanti:

- **Singleton:** definisce un'operazione che permette di creare un'unica istanza di un oggetto, e di accedervi.

Elenco di Singleton:

- Nel Server, la classe che costruisce il Server per gestire le connessioni con i Client attraverso Socket.

  `public class ServerSocketCore ,` con i metodi
  `public static ServerSocketCore createServerInstance(GlobalSettings locals)`
  e `public static ServerSocketCore getServerInstance()`

- Nel Server, allo stesso modo, la classe che gestisce le connessioni con i Client attraverso RMI.

  `public class ServerRMICore,` con il metodo
  `public static ServerRMICore getServerInstance(GlobalSettings locals)`

5.5 Proxy

Un oggetto agisce da intermediario e/o sostituto per un altro oggetto. Le ragioni possono essere varie (evitare l'istanziazione di un oggetto pesante in termini di risorse, controllare l'accesso a un oggetto, gestire alcune richieste in un contesto locale invece di richiedere di gestirle in remoto, etc.)

In particolare, un **remote proxy** fornisce un rappresentante locale per un oggetto remoto.

Partecipanti:

- **Subject:** l'interfaccia comune al Proxy e al Real Subject
- **Proxy:** implementa l'interfaccia Subject (può quindi essere usato al posto del Real Subject); mantiene un riferimento al Real Subject (può inviargli richieste se occorre).
- **Real Subject**: l'oggetto rappresentato dal Proxy

Utilizzo:

Nella comunicazione Client-Server di tipo RMI, è un oggetto locale sul client che è in grado di inviare al Server delle richieste, e inoltre invia al Server un riferimento a se stesso, provvedendo così all'autenticazione.

Qui è presente una sottile variazione sul tema rispetto al proxy standard, dato che la funzionalità di autenticazione è fornita attraverso metodi overloaded, in modo da semplificare il compito del chiamante.

- **Subject:** `public interface ServerRemoteInterface.` Definisce tutti i metodi che il client può invocare sul Server da remoto.
- **Proxy:** `public class ProxyToServer implements ServerRemoteInterface`
- **Real Subject**: `public class ServerRMICore implements ServerRemoteInterface`

5.6 State

Permette a un oggetto di cambiare il proprio comportamento quando il proprio stato interno cambia, usando diverse classi.

Partecipanti:

Context: questo oggetto mantiene un riferimento a un oggetto ConcreteState, che definisce lo stato corrente.

State : interfaccia / superclasse per i differenti ConcreteState

ConcreteState: sottoclassi di State; ciascuna definisce un diverso comportamento.

Impiego:

Nel Server, esistono gli strumenti per generare una mappa manualmente, inserendo le celle e i loro tipi in linea di comando (package server.model.gamemap.qdgenerator).

Ruoli:

- **Context:** `public class CommandLine,`
  che all'interno del metodo `addCellsData()`
  ha il riferimento `CellAcquirer current = new SafeCellAcquirer();`
- **State :** `public abstract class CellAcquirer`
- **ConcreteState:**
  - `public class HumanStartingCellAcquirer extends CellAcquirer`
  - `public class AlienStartingCellAcquirer extends CellAcquirer`
  - `public class EscapeCellAcquirer extends CellAcquirer`
  - `public class SafeCellAcquirer extends CellAcquirer`
  - `public class DangerousCellAcquirer extends CellAcquirer`

5.7 Iterator – iterable

Il metodo Iterator ritorna un oggetto “iterator” che permette di attraversare gli elementi di una collezione.

Partecipanti:

- **Iterator**: interfaccia che definisce i metodi per attraversare una collezione di elementi: next(), hasNext(), remove()
- **Iterable**: interfaccia che definisce il metodo iterator(), che ritorna un oggetto Iterator
- **Concrete Iterator :** Implementa l'interfaccia Iterator, è in grado di percorrere gli elementi di una collezione.
- **Concrete Iterable:** Implementa l'interfaccia Iterable, può quindi creare un'oggetto Concrete Iterator.

Impiego:

La classe mapLoader permette di caricare la mappa di gioco da un file. La mappa di gioco è memorizzata come file Json, contenente le proprietà di base (nome, dimensioni) e un array con la posizione e il tipo di ciascuna casella.

Il mapLoader deve essere il più possibile indipendente da come poi la mappa sarà effettivamente immagazzinata negli oggetti che la utilizzano (server model / schermata di swing). Per fare questo, mapLoader è Iterable, e fornisce un iterator sulle celle esagonali caricate da file.

Ruoli delle classi:

- **Iterator:** l'interfaccia predefinita `Iterator<>`
- **Iterable:** l'interfaccia predefinita `Iterable<>`
- **Concrete Iterator:** `public class MapIterator implements Iterator<Cell>,` che contiene i metodi: `boolean hasNext(), Cell next(),` e `void remove()` (quest'ultimo non ha nessun effetto)

- **Concrete Iterable:** `public class MapLoader implements Iterable<Cell>,` contiene il metodo `public MapIterator iterator()`

Illustration Index