Progetto di

Programmazione Avanzata

Papersoccer



Simone Francalancia

Andre Di Ubaldo

Riccardo Mendicino

L31 Informatica 2019

Introduzione 2

view View 3

controller StartPanelController 4

view Startpanel 6

controller InteractiveMatchController 7

controller RandomMatchController 9

model.core Ball 10

model.core Cell 10

model.core CellLocaton 11

model.core Directions 11

model.core MatrixSoccerField 12

model.core Size 16

model.match Match 16

moder.match InteractiveMatch 17

model.match RandomMatch 19

model.ruleset DefaultRuleSet 20

model.player Player 23

model.player InteractivePlayer 24

model.player RandomPlayer 25

view.FrameCampo Campo 28

view.FrameCampo ghiera 29

view.FrameCampo playerboxtext 30

view.FrameCampo tracciato 31

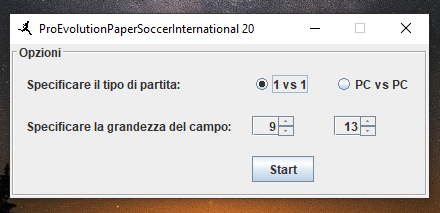
controller EndPanelController 32

view Endpanel 33

Test 34

**Introduzione a PaperSoccer**

Paper soccer è un semplice gioco di calcio rappresentato su una griglia dove si vince se il giocatore rosso va in porta blu o viceversa. Appena avvia davanti a noi troveremo una finestra che ci permette di scegliere il tipo di match e le dimensioni del campo.

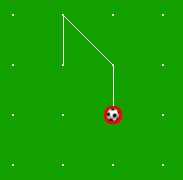
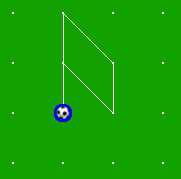


Ci sono due tipi di match, **1 vs 1** e **PC vs PC:**

* Nella modalità 1vs1 per giocare basta cliccare il punto del campo dove si vuole spostare la palla, l’idea è quindi quella di alternarsi il mouse con un amico e di fare una partita assieme.
* Nella modalità PCvsPC saremo invece spettatori di una partita tra due bot che calceranno la palla ogni volta che viene premuta la **barra spaziatrice**.

**Regole del gioco (1 vs 1)**

Le regole sono semplici:

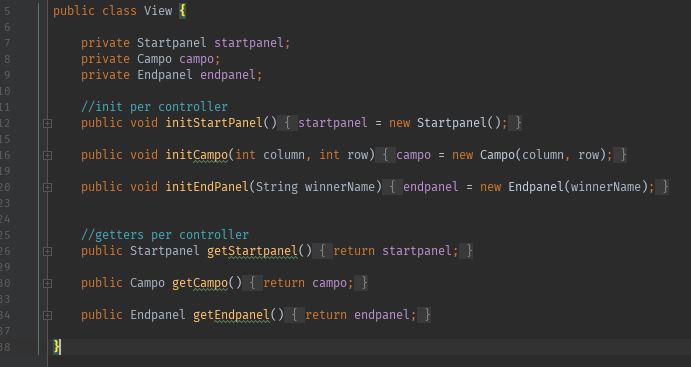
* Non si può ripercorrere la propria scia.
* Ogni giocatore può di base effettuare una mossa per turno, a meno che non prenda un angolo di rimbalzo
* Si può ritirare la palla per ogni rimbalzo preso.
* Se si va in una casella dove successivamente non ci saranno mosse disponibili, il gioco termina.

**view.View**

si occupa di racchiudere e

inizializzare le classi grafiche del programma

* Startpanel
* Campo
* Endpanel



È stato scelto di aggiungere questa classe poiché avevamo bisogno di qualcosa che si occupasse di inizializzare i vari frame in base alle chiamate effetuate, poiché i 3 frame principali, che sono quelli sopra elencati vengono tutti evocati in momenti differenti per offrire una migliore visione del programma ed esperienza in generale.

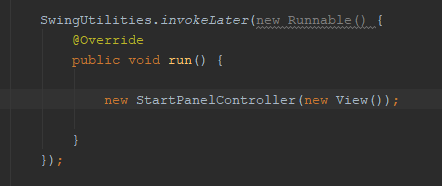
È composta di 6 metodi, 3 per inizialiazzare i JFrame (12-22) e altri 3 che sono i getters per il controller (26-36).

**controller.StartPanelController**

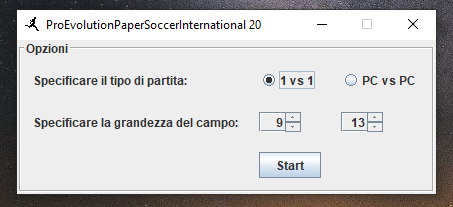
Il **controller**, nel modello MVC, è quell’elemento che “fa da collante” tra la grafica e la logica in modo tale da renderle indipendenti tra di loro.

Nel Main l’unico comando presente crea un oggetto StartPanelController con argomento un oggetto View.

*Immagine del main:*



Il package controller contiene diversi tipi di controller, uno per ogni “elemento interattivo” presente nel programma, quindi StartPanelController è il controller associato all’ oggetto Startpanel cioè la prima finestra che appare quando si avvia PaperSoccer.

*Immagine dello Startpanel:*  


StartPanelController ha il compito di ascoltare lo Startpanel e di passare i relativi input al model; è definito come segue:

**Attributi:**

* private View **view**; -> per gestire gli elementi grafici
* protected static int **row**, **column;** -> per gestire le dimensioni del campo

**Costruttore:**

* StartPanelController(View view);

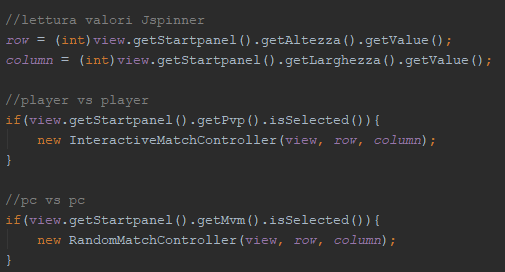
**Metodi:**

* addStartBtn(); -> assegna l’action listener al bottone Start

Il concetto è semplice: il costruttore vuole come argomento un oggetto di classe View in modo tale da chiamare il metodo **initStartPanel();** che permette di creare la grafica della finestra.

Successivamente viene invocato il metodo **addStartBtn();** che aggiunge un action listener (chiamato startgame) che permette la lettura di tutti gli input che l’utente passerà al model tramite gli elementi grafici della view.

Quindi addStartBtn premette di scegliere la dimensione del campo e di selezionare il tipo di partita (creando quindi nuovi oggetti controller per gestire i relativi tipi di match).

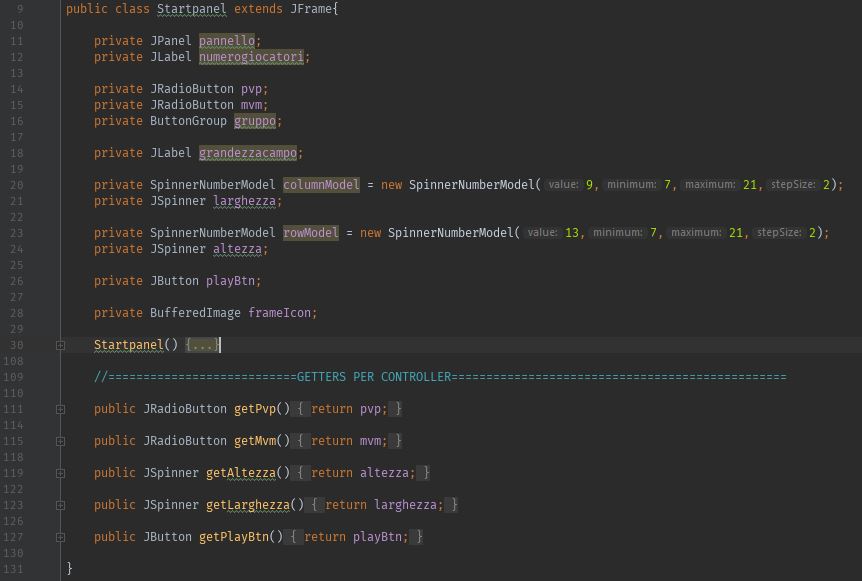


Appena premuto **Start** i valori verranno passati al model e sia startPanelController sia Startpanel verranno distrutti.

**view.Startpanel**

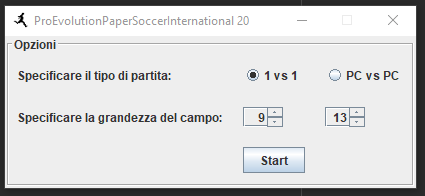
il pannello che si trova di fronte l’utente appena inviato il programma. Si occupa di ricevere le impostazioni della partita specificate dall’utente e di inviarle indietro a StartPanelController .

Abbiamo deciso di impostare delle dimensioni minime e massime in modo tale che il campo rientri nei confini dello schermo (in caso max) e abbia senso giocare (in caso min) (r20 e r23).

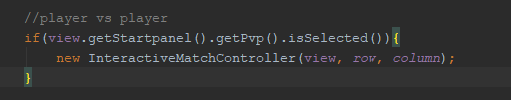


La costruzione della classe consiste in un JFrame contentente un Jpanel con all’interno radiobuttons e spinners per le impostazioni della partita.

I rispettivi getters si occuperanno di comunicare con il controller per salvare le impostazioni date dall’utente.



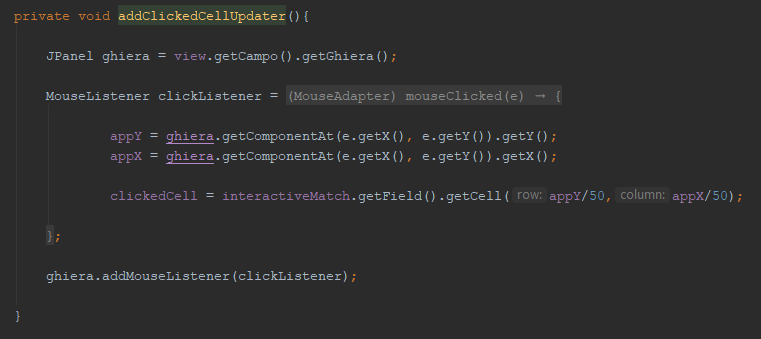
**controller.InteractiveMatchController**



Se si seleziona 1vs1 nel menù, viene creato un oggetto di classe **InteractiveMatchController** che si occuperà dell’interazione tra la view e il model per gestire la partita tra due persone, della creazione del campo delle dimensioni desiderate e della sua rappresentazione a video.

Il suo ruolo da qui in poi sarà quello di:

* ottenere la posizione della cella sotto il click del mouse tramite il listener (clickListener) implementato nel metodo addClickedCellUpdater();



appY e appX tramite questo metodo conterranno le coordinate in pixel del centro della cella della ghiera. Per ottenere la cella della matrice invece andremo a dividere quei valori per la grandezza delle immagini usate, cioè 50 pixel. Quindi se clicco il componente in Y: 150px e X:100px otterremo la cella della matrice (3,2).

* controllare la validità della mossa



* permettere la mossa  
  
* aggiornare la view tramite il metodo **draw**(appY, appX) presente nella classe Tracciato (che aggiunge i punti cliccati ad un arrayList per muovere graficamente la palla e disegnare la sua scia)  
    
  e il metodo **cambioturno** che muove la freccia dei turni (playerboxtext) in basso  
  
* controllare se la partita è terminata

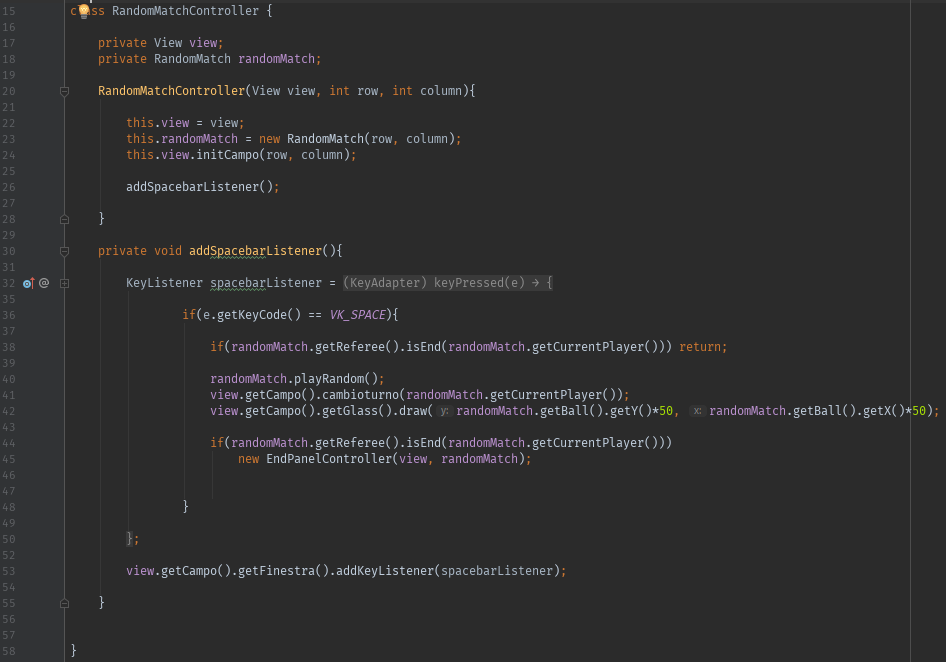
**controller.RandomMatchController**

Nel caso invece in cui l’utente abbia selezionato PCvPC sarà il turno del **RandomMatchController**.

Come L’interactiveMatchController ha il compito della creazione del campo e della gestione della partita; questa volta non tra due persone ma tra due **randomPlayer**.

Quindi invece di passare al model la cella ottenuta tramite il click del mouse, gli passerà le direzioni scelte dai due **bot** (che non hanno bisogno di cliccare la ghiera di gioco).

Quando viene premuta la **BARRA SPAZIATRICE** il bot corrente effettuerà la mossa, alternando i turni e così via permettendo all’utente di osservare la partita.

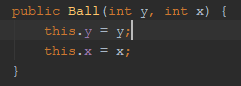


**model.core.Ball**

Semplicissima classe per tenere traccia delle coordinate della palla; è costituita quindi solamente da due attributi **y** e **x**:



Il costruttore è altrettanto semplice e serve per inizializzarla nelle coordinate desiderate:



I suoi unici metodi sono i getters e setters per utilizzare i due attributi.

**model.core.Cell**

Classe che definisce la “casella della scacchiera” sopra la quale si può muovere la palla. Ha come attributi un oggetto **loc** di classe **cellLocation** e un arrayList di direzioni (**Directions**).



* **loc:** oggetto che ha come attributi **row** e **column**; serve per identificare la posizione della cella sulla ghiera.
* **arrayLoc:** arrayList che serve per contenere le direzioni **non disponibili** della cella.

Quindi se l’arrayLoc di una cella X è vuoto, significa che da quella casella X possiamo spostare la palla in una direzione qualsiasi. Mentre se per esempio arrayLoc contenesse NORTH, NORTH\_EAST e EAST non sarebbe possibile spostare la palla in quelle direzioni ma potremmo spostarla in tutte le altre.

**Costruttori:**

* Cell(int row, int column)
* Cell(cellLocation loc)

Inizializzano la cella nella coordinata specificata

**Metodi:**

* arrayLocIsFull() -> ritorna true se arrayLoc contiene 8 elementi (cioè è **pieno**)
* addListDirections(Directions… d) -> aggiunge tutte le direzioni passate come argomento all’arrayLoc della cella.
* **matchDirection(cell Dest)** -> prende come argomento la cella di destinazione (cioè dove si vuole spostare la palla) e ritornerà quella direzione, altrimenti ritornerà null. Ad esempio se voglio muovere la palla a NORTH ritornerà NORTH se disponibile.

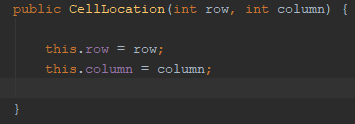
**model.core.CellLocation**

Gli oggetti di questa classe servono solamente come attributo per gli oggetti di classe Cell. Un oggetto CellLocation identifica la posizione di una cella sulla matrice di classe MatrixSoccerField.

E’ composta dai due seguenti attributi:



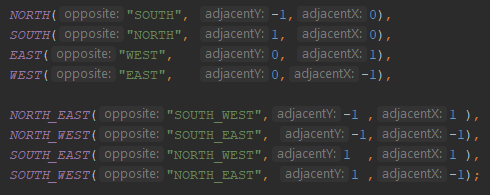
e dal seguente costruttore:



I metodi di questa classe sono i getters e i setters degli attributi e un semplice toString.

**model.core.Directions**

Classe **enum** che serve per specificare le direzioni/spostamenti che la palla può fare, sono i punti cardinali NORTH, SOUTH, EAST, WEST, NORTH\_EAST, NORTH\_WEST, SOUTH\_EAST e SOUTH\_WEST per un totale di 8 direzioni disponibili.

*Immagine della classe Directions:*

Ad ogni direzione è associata una stringa che contiene la direzione opposta (per comodità) e due int.

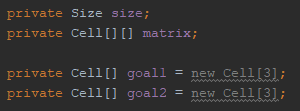
I due interi sono stati introdotti per facilitarci nello sviluppo del bot: sono la Y e la X da sommare alle coordinate attuali della palla per ottenere la cella di destinazione della mossa che il computer vuole fare.

Ad esempio, se la palla si trova in (Y: 5, X: 5) e il bot vuole spostare la palla verso sud (SOUTH), per ottenere la cella di destinazione basterebbe sommare l’**adjacentY** di SOUTH (+1) alla Y della palla e l’**adjacentX** di SOUTH (+0) alla X della palla. Quindi ora sappiamo che la cella di destinazione si trova in (Y: 6, X: 5)

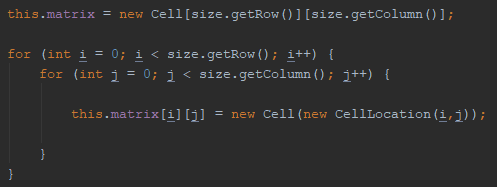
***Lo spostamento sull’asse delle Y è specchiato perché punto centrale del JFrame (0,0) si trova in alto a sinistra, per questo per spostarci in giù verso sud si somma 1 invece di sottrarlo.***

**model.core.MatrixSoccerField**

Come suggerisce il nome, è la classe che costituisce il campo da gioco. Ha un attributo size per definirne la dimensione in termini di row e column ed formata da una **matrice di celle** per il campo da gioco e da 2 array (sempre di celle) di dimensione fissa 3 per le porte.

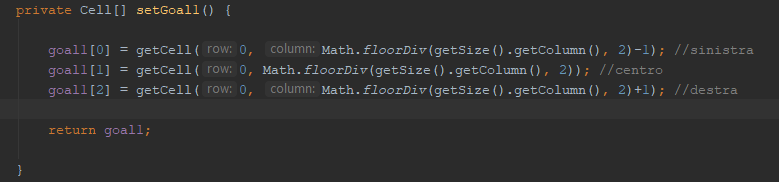


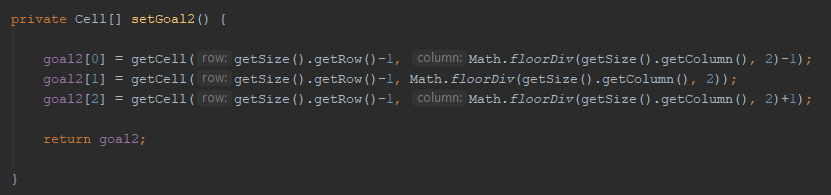
Il costruttore, che richiede come parametro solamente un oggetto di classe Size, si occupa di istanziare ed inizializzare le celle che vanno a costituire il campo



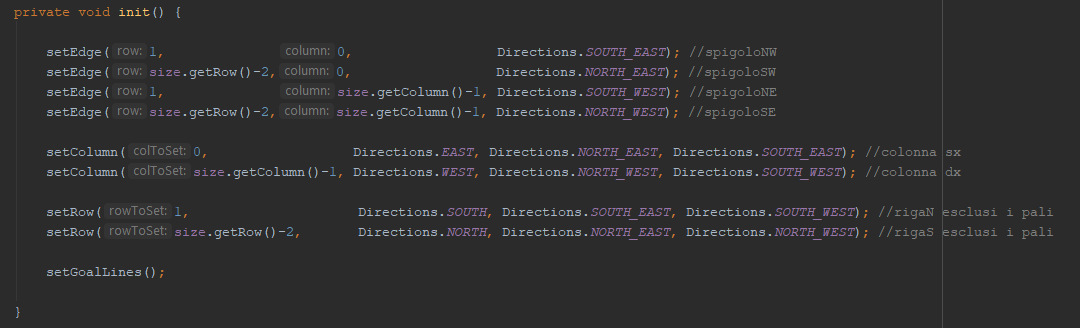
e quelle che vanno ad impostare le porte richiamando gli appositi metodi **setGoal1();** per la porta rossa e **setGoal2();** per la porta blu







L’ultimo metodo ad essere invocato nel costruttore è **init();**



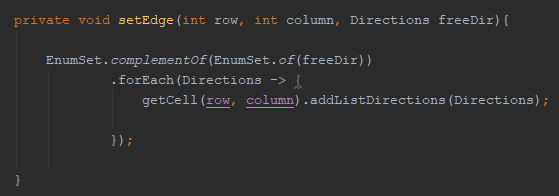
metodo **FONDAMENTALE** in quanto imposta le “direzioni occupate” delle celle che costituiscono gli **angoli** e i **bordi** del campo in modo tale da non permettere mosse al di fuori di esso. Il suo scopo è quello di bloccare le direzioni dove non vogliamo che la palla vada scrivendo quelle direzioni dentro l’arrayLoc della cella interessata. Ad esempio: non voglio che la palla possa uscire dal bordo destro del campo. Scrivo quindi EAST dentro l’arrayLoc della cella a bordo campo, bloccando quindi quella mossa.

Per fare questo init si appoggia su 4 metodi:

* **setEdge**(int row, int column, Directions freeDir);

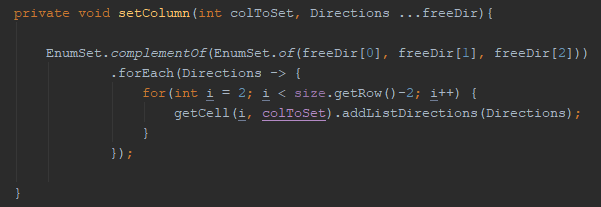
si occupa di impostare le direzioni degli angoli. Gli angoli hanno solo una sola direzione disponibile (se si va lì si perde la partita) quindi setEdge scriverà

nell’ arrayLoc della cella all’angolo **tutte** le direzioni della classe enum Dir ections **lasciando** però **libera** quella da noi interessata (fa quindi il complemento).



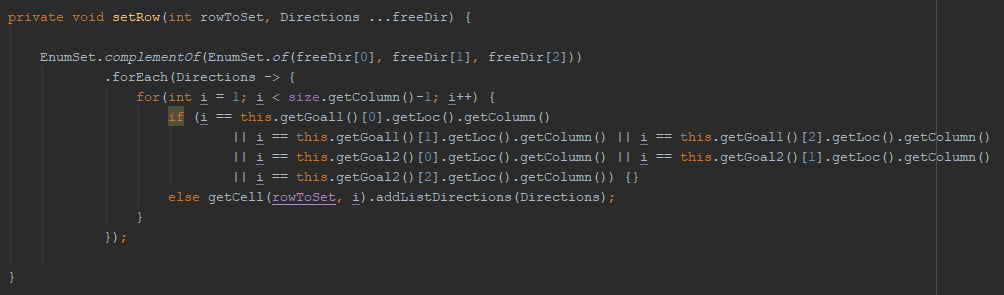
* **SetColumn**(int colToSet, Directions… freeDir);

Stesso concetto di setEdge, ma applicato alla colonna specificata come argomento e con più argomenti Directions passabili come parametro.



* **setRow**(int rowToSet, Directions… freeDir);

stesso concetto di sempre, ma applicato alle righe a nord e sud quindi richiede qualche controllo in più visto che sono presenti gli angoli delle porte



* **SetGoalLines**();

imposta gli angoli interni di fronte alle due porte

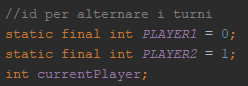
**model.core.Size**

E’ la classe che tiene i valori row e column inseriti dallo startpanel e che viene chiamata da **InteractiveMatch** o dal **RandomMatch** e **MatrixSoccerField** per restituire i valori delle dimensioni del campo.

**model.match.Match**

Classe astratta che serve per implementare le funzionalità e i metodi in comune che useranno entrambe le sue sottoclassi (non gestisce quindi la partita). Contiene i seguenti attributi:

* Id per scandire i turni tra un giocatore e l’altro (sia umani sia bot) durante la partita:



* Campo da gioco e palla:



* Regolamento contenente i metodi per effettuare i vari controlli durante la partita



Ed i seguenti metodi:

* **otherPlayer();**

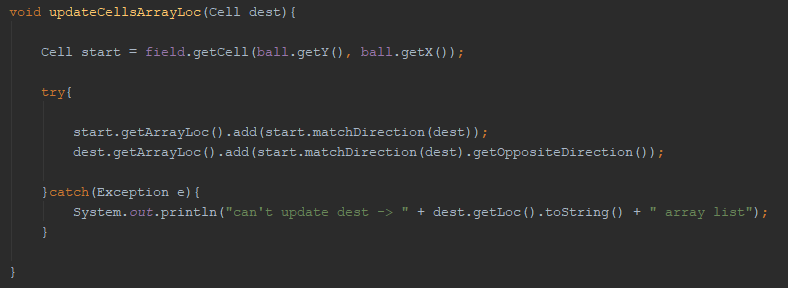
ritorna l’id dell’altro giocatore (quello non giocante) richiamato in genere alla fine dei turni per alternarli



* **updateCellsArrayLoc(Cell dest);**

metodo *sovraccaricato* in modo da poter essere sfruttato anche con le Directions (quindi per i bot).

Aggiorna l’arrayLoc delle celle “sotto” lo spostamento della palla



Ad esempio: se dalla cella (5,5) mi sposto a NORTH, sull’ arrayLoc della cella **start** (5,5) il metodo scriverà NORTH mentre sulla cella di destinazione **dest** (6,5) scriverà SOUTH. Questo perché dalla cella di partenza mi sposto verso nord e dalla cella di destinazione arrivo da sud.

* **updateCellsArrayLoc(Directions dir);**

Come il metodo precedente ma con l’aggiunta di questa riga di codice per ottenere la cella di destinazione



**model.match.InteractiveMatch**

Classe estesa da Match quindi implementa tutti i suoi attributi e i suoi metodi.

Aggiunge in più un array di InteractivePlayers come attributo che serve per contenere ed alternare i 2 players durante la partita (usato in combinazione al metodo otherPlayer già descritto sopra)



e i seguenti due metodi:

* **playInteractive(Cell Dest);**

che è il metodo che gestisce la partita tra due persone



**Breve descrizione dell’idea dietro il funzionamento del metodo:**

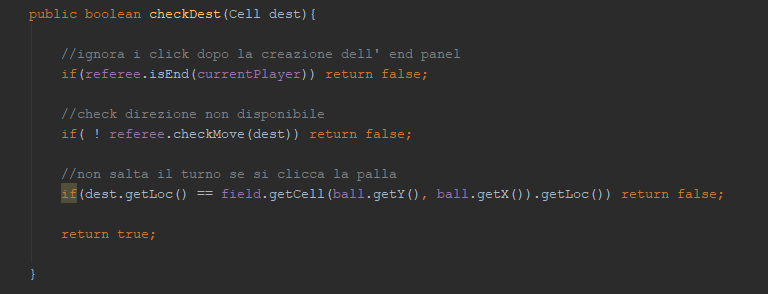
se viene richiamato questo metodo significa che la cella cliccata con il mouse è disponibile (grazie a checkDest() ), quindi se il player ha almeno una mossa viene **aggiornato l’arrayLoc** delle celle start e dest, si **sposta la palla** nella cella desiderata, si **controlla** se è stato effettuato un **rimbalzo** (se sì il contatore delle mosse del player corrente si incrementa di 1) e si **decrementa** il contatore delle **mosse** del player (perché almeno una mossa è stata fatta). Alla fine del turno, quando il giocatore corrente avrà terminato le mosse, gli verrà assegnata una mossa (per non farlo iniziare con il contatore a 0 nel suo turno successivo) e verranno **scambiati i player** in modo tale da permettere all’ altra persona di giocare.

* **checkDest(Cell dest);**

Questo metodo è richiamato nell’InteractiveMatchController



perché essendo la view e il model indipendenti, la view “non può sapere” se la mossa effettuata è corretta o sbagliata, quindi se questo controllo fosse stato messo nel metodo precedente (cioè nel model) il model non si aggiornerebbe mentre la view sì; **quindi graficamente la palla si troverebbe in una posizione dove in realtà non è.**

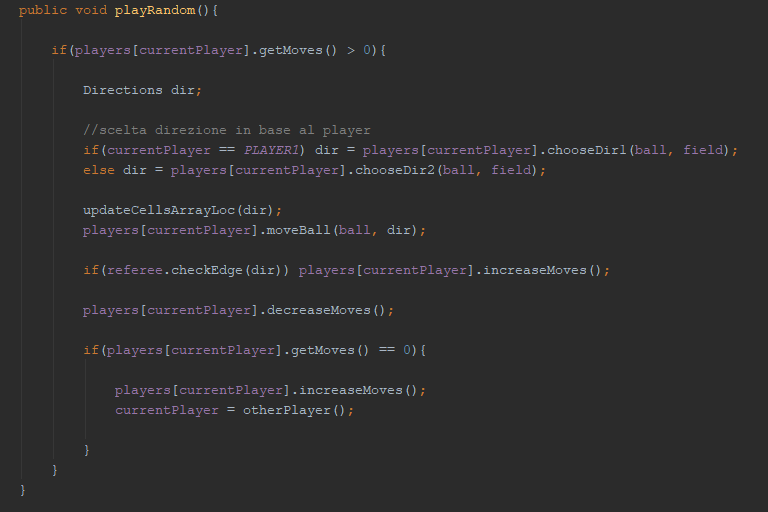


**model.match.RandomMatch**

Ha il compito di gestire le partite tra i randomPlayer, cioè tra i bot. Come InteractiveMatch aggiunge un attributo: un array di RandomPlayer



e il metodo che permette di ai bot di giocare



La differenza tra playRandom e playInteractive sta nella scelta della direzione tramite i metodi **chooseDir1** e **chooseDir2** (definiti entrambi nella classe RandomPlayer) e nel metodo che sposta la palla, cioè moveBall (overload del metodo), che ha come argomento la palla e la direzione dove spostarla invece della cella direttamente data dal click dell’utente.

**ruleSet.DefaultRuleSet**

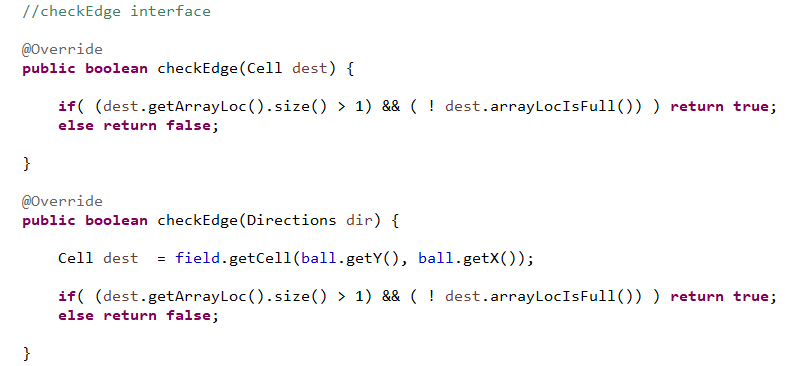
La classe DefaultRuleSet contenuta nel package it.unicam.cs.pa.paperSoccer.model.ruleSet è la classe dell’unico regolamento presente nel progetto, che implementa le interfacce checkEdge, checkMove e checkEnd. Nelle prime due sono presenti due metodi che sono simili, ma che sono destinati al match di tipo InteractiveMatch nel caso dei primi e al match di tipo RandomMatch nel caso dei secondi. È necessario che eventuali regolamenti implementino entrambi per permettere il funzionamento di entrambi i tipi di Match.

L’interfaccia checkEdge che dichiara i metodi per verificare la presenza dell’edge, ossia la presenza di una o più direzioni diverse da quella che si aggiunge con lo spostamento, uno che prende una cella e uno che prende la direzione dello spostamento della palla come parametro.

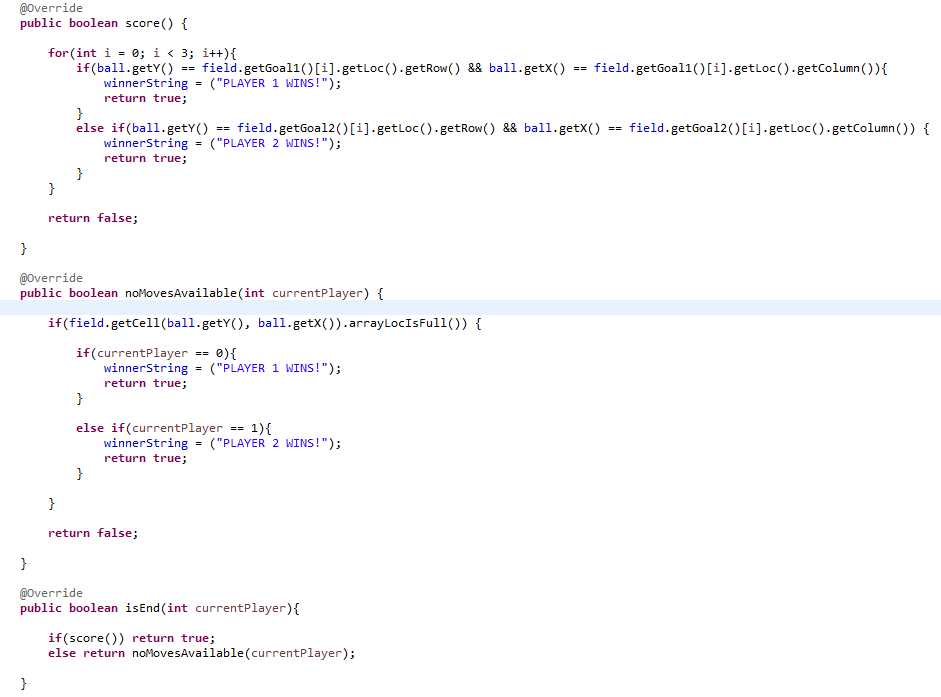
L’interfaccia checkMove dichiara i metodi per il controllo del movimento, ossia che la cella, nel caso del metodo checkMove, o la direzione, nel caso del metodo checKMove siano liberi di ricevere la palla.

L’interfaccia checkEnd che verifica le condizioni per la fine del match, ossia lo score, attraverso il metodo score, o la mancanza di direzioni libere all’interno della cella dove si trova attualmente la palla, attraverso il metodo noMovesAvailable che prende il campo intero currentPlayer. Dichiara inoltre il metodo isEnd per verificare la soddisfazione di una delle due condizioni di fine partita.

Nel DefaultRuleSet sono definiti i metodi dell’interfaccia checkEdge, ossia checkEdge di tipo boolean che prende una cella come parametro, utilizzata per controllare che il suo array di direzioni arrayLoc non sia pieno e che contieni una o più direzioni e il checkEdge che prende la direzione come parametro, che poi converte in cella per poi seguire lo stesso meccanismo del primo.

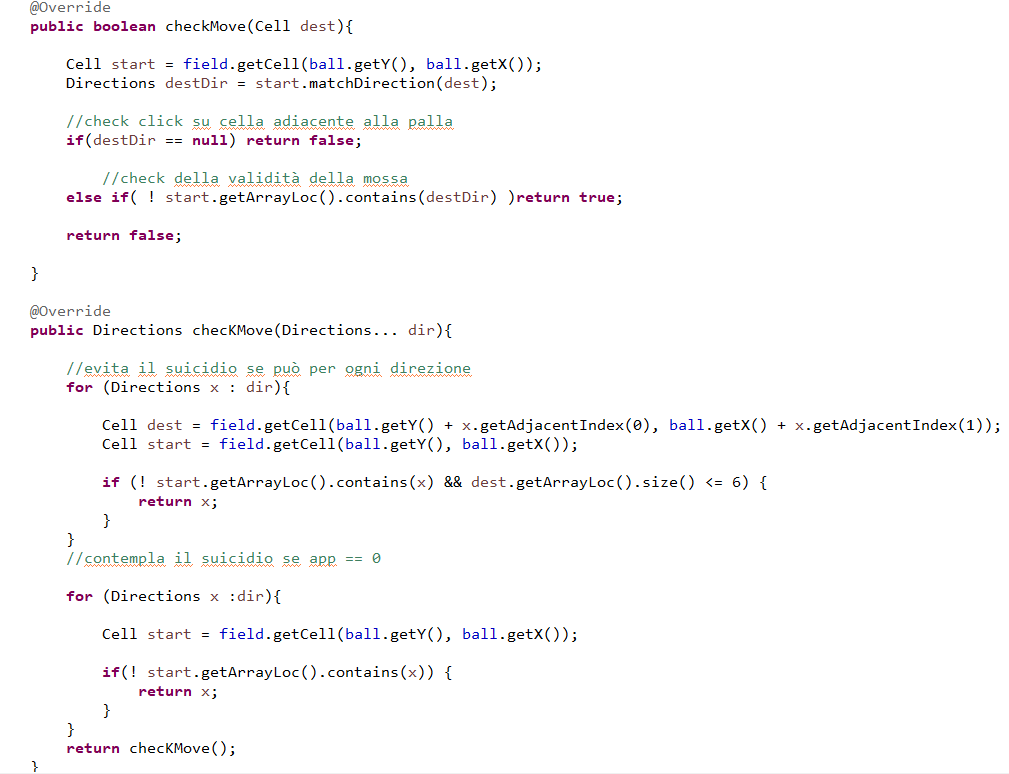


Implementa i metodi dell’interfaccia checkEnd ossia, score che attraverso la struttura di controllo iterativa controlla se la palla sia in una delle tre celle di una delle due porte, restituendo un messaggio di vittoria e un valore booleano true, che servirà poi nel metodo isEnd come condizione di fine partita, e il metodo noMovesAvailable che controlla se l’arrayList di direzioni della cella è pieno oppure no, restituendo messaggio di vittoria in base al giocatore attuale e un valore booleano true in caso di fine partita, false altrimenti. Il metodo isEnd controlla se il metodo score restituisce true, portando alla fine della partita, altrimenti restituisce il risultato di noMovesAvailable, true o false a seconda della fine della partita o meno.



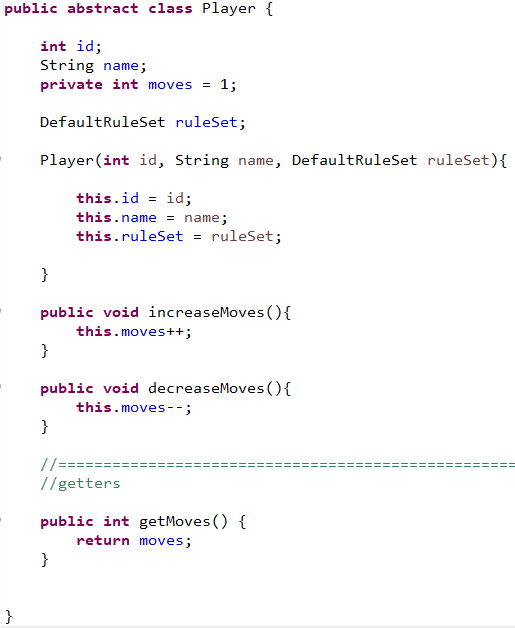
La classe implementa inoltre i metodi dell’interfaccia checkMove: checkMove che prende una cella ricavandone la direzione e controllando se non sia nulla (in caso che la cella sia uguale a quella di partenza) e se non sia presente nell’array delle direzioni della cella di partenza, e checKMove che prende un insieme di direzioni poiché cerca per ognuna di queste quella più conveniente, ovvero quella che porta ad una cella che ha un array di direzioni minore o uguale a sei e che non contiene la direzione; se ciò non è possibile si segue lo stesso discorso per checkMove.

Infine abbiamo il metodo getter per prendere la stringa di ritorno in caso di vittoria e i metodi setter per l’impostazione del campo e della palla.



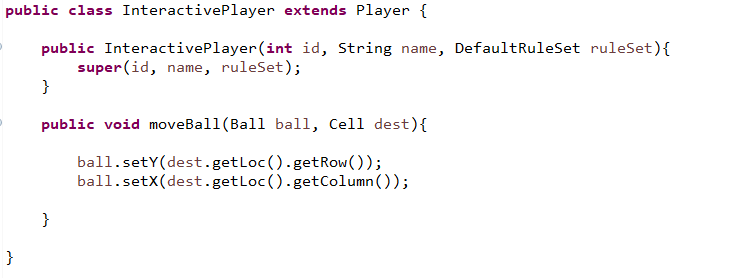
**model.player.Player**

La classe Player contenuta nel package it.unicam.cs.pa.paperSoccer.model.player è una classe astratta che ogni classe di Player specifica deve implementare, poiché essa contiene un attributo di tipo Intero chiamato **Id** necessario per identificare un giocatore, un attributo di tipo String per il **nome**, un attributo per gli spostamenti di palla disponibili di tipo Intero chiamato **moves**, e il regolamento che il giocatore deve rispettare di tipo DefaultRuleSet chiamato **ruleSet**. Tra i metodi troviamo il costruttore definisce i primi tre campi delle istanze che lo istanziano, i metodi increaseMoves e decreaseMoves per aumentare e diminuire i passaggi di palla disponibili nel turno del rispettivo giocatore, secondo il regolamento specificato e il metodo getMoves per acquisire il valore del campo moves, in quanto privato come tutti i campi dell’oggetto per il principio dell’incapsulazione.



**model.player.interactivePlayer**

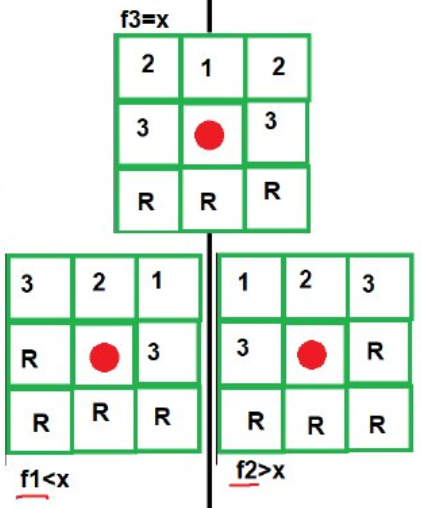
La classe InteractivePlayer contenuta nel package it.unicam.cs.pa.paperSoccer.model.player è la classe del player interattivo, ossia di ciascun utente che attraverso il click del mouse e la grafica riesce a giocare la partita 1vs1 contro il rispettivo avversario. Essa definisce un costruttore che attraverso il metodo super richiama il costruttore della classe player per impostare i valori dei campi id, nome e ruleSet e un metodo chiamato moveBall che ha un oggetto ball di tipo Ball e la cella di destinazione dello spostamento dell come parametri. Quest’ultimo si occupa dello spostamento della palla, ossia dell’impostazioni delle nuove coordinate della palla che effettuano lo spostamento, ma non si occupa dell’aggiornamento degli array delle direzioni delle celle coinvolte nello spostamento, cosa di cui si occupa il metodo updateCellsArrayLoc nella classe Match.



**model.player.RandomPlayer**

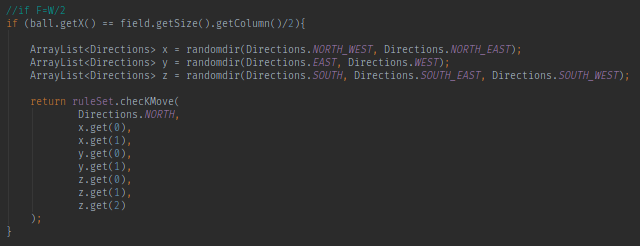
La classe RandomPlayer si occupa dello spostamento da parte del computer.

Invece che abbandonarci ad un semplice movimento random giusto per inizializzare il “giocatore computer”, abbiamo deciso di fingere una “pseudo intelligenza” che permetta al computer di fare scelte in base a quello che anche un giocatore reale farebbe.



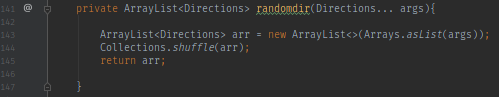
L’immagine sopra raffigura la nostra idea di “mossa migliore” in base alla posizione in cui la palla si trova dove **f1** rappresenta il lato sinistro del campo, **f3** la corsia centrale, **f2** il lato destro del campo.

I numeri dentro ogni quadrato indicano l’ordine di preferenza della mossa in ordine ascendente, confrontato con il codice il risultato diventa questo:

*Esempio di movimento, caso f3*

Per capire come agisce dobbiamo prima spiegare la funzione **randomdir** e ricordare l’utilizzo di **rulSet.checKMove.**

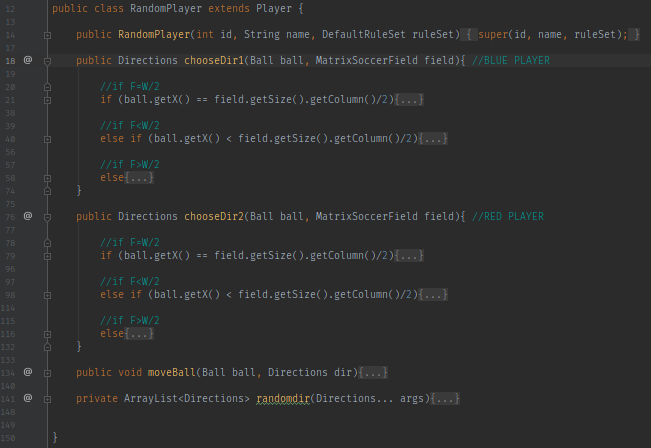
In breve il cechKMove permette di capire se la mossa è valida e si può attuare, mentre il randomdir, preso un array di direzioni restituisce lo stesso array shufflato.



Quindi:

vengono stabiliti degli array che racchiudono le direzioni in cui la preferenza è uguale, (per esempio in corsia centrale se il nord fosse occupato sceglierebbe in maniera randomica se andare a nordest o nordovest), e successivamente viene stilata dentro al metodo checkmove la lista delle direzione riarrangiata randomicamente ma rispettando le preferenze. L’unione delle varie posizioni della palla in base al campo vanno a formare **ChooseDir1** per il player blu, e **ChooseDir2** per il player rosso.

Il **match.RandomMatch**, che specifichiamo è anche l’utilizzatore dei chooseDir sopra spiegati, dopo le rispettive verifiche, sposterà la palla tramite il metodo **moveBall()** di RandomPlayer.

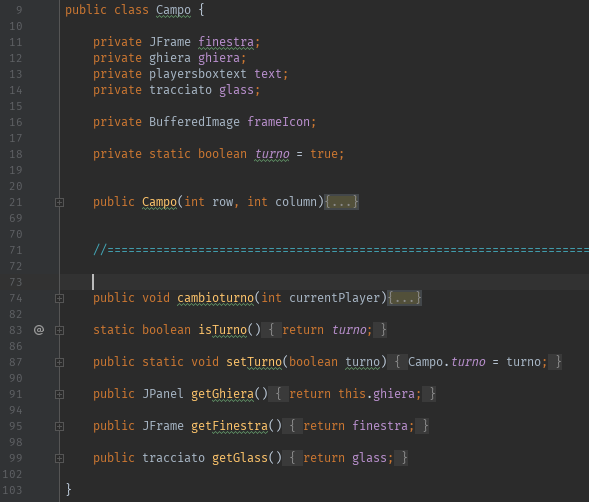
Tradotto in codice:

**view.FrameCampo.Campo**

successivamente alla selezione delle impostazioni il campo comparirà sullo schermo. Il campo identifica un frame contenente vari JPanel con ruoli differenti, tali jpanel sono:

* ghiera
* playerboxtext
* tracciato

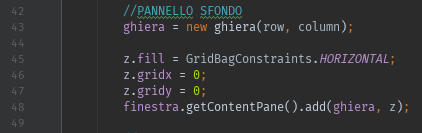
la classe si occupa della rappresentazione grafica della partita e contiene le chiamate alle ulteriori classi citate sopra di cui andremo a commentare i ruoli di seguito.



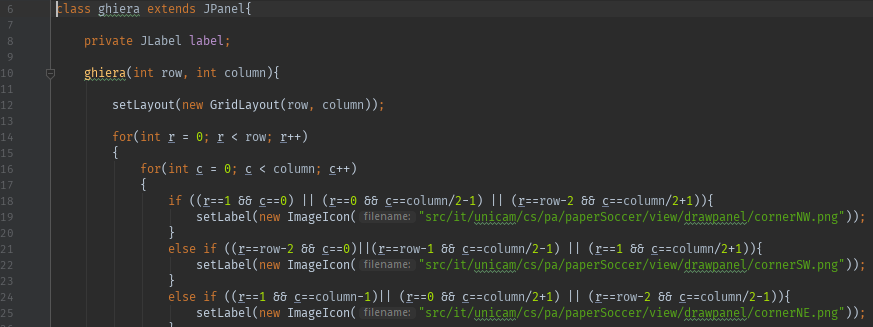
*La classe campo*

**view.FrameCampo.ghiera**

la classe fondamentale che si occupa della rappresentazione grafica delle caselle di gioco. è un semplice doppio ciclo for che date le coordinate specificate prenderà dalla folder "drawpanel" le icone corrispondenti. raffiugranti alla fine del doppio ciclo il campo completo.

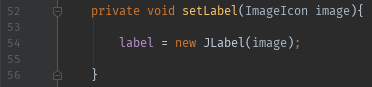


*ghiera dentro a Campo*



*La classe ghiera*

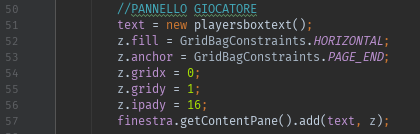
Abbiamo tagliato la foto per questioni di spazio, tuttavia il funzionamento è abbastanza chiaro. Alla fine della classe è presente il metodo **SetLabel()** il cui scopo non è altro che quello di creare le nuove immagini.



**view.FrameCampo.playerboxtext**

la classe che si occupa di mostrare graficamente il turno del giocatore ai piedi del frame.

il cambio del turno viene gestito dal Campo tramite il metodo "**isTurno()**" dato poi a “**setttext()**”.



Playerboxtext dentro a Campo.



La classe playerboxtext.

**view.FrameCampo.tracciato**

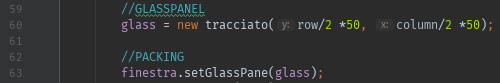
classe raffigurante il tracciato della palla, il colore della palla, e il movimento della stessa durante la partita. la peculiarità sta nel fatto di essere un jpanel reso invisibile nella classe Campo (tranne che il paintComponent) dalla chiamata setGlassPane. Il funzionamento viene reso possibile grazie all’InteractiveMatchController/RandomMatchController che ad ogni spostamento, aggiungono all’array dei punti la nuova destinazione e disegna la traccia tramite "paintComponent(Graphics g)" poichè il metodo grafico non lavora con le coordinate delle caselle del campo ma con i pixel di quest'ultimo.

Le coordinate per il disegno vengono date

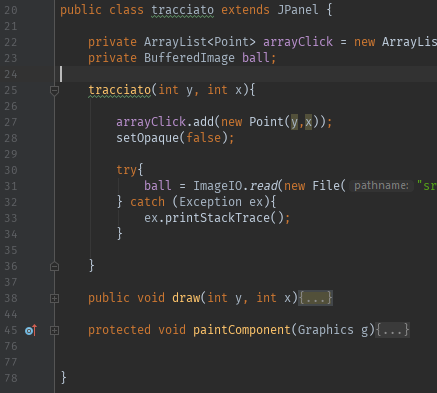
**1)** selezionando la casella

**2)** moltiplicando la X e la Y della casella selezionata per 50 (otterremo in pixel l’angolo in alto a sinistra della casella selezionata)

**3)** marcandone il centro aggiungendo la metà della dimensione di una cella (50/2 =25) sia in altezza che larghezza.



T*racciato dentro a Campo. I primi due punti inseriti equivalgono alla partenza della palla, a centrocampo.*



*Tracciato e il suo PaintComponent*

**controller.EndPanelContoller**

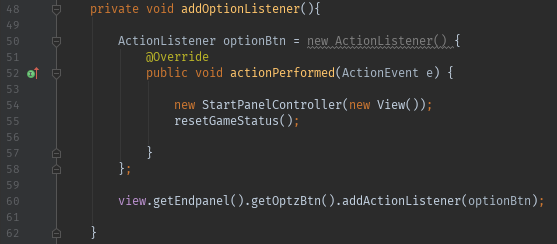
Invocato sia da InteractiveMatchController che da RandomMatchController alla fine di una partita si occupa di gestire la finestra che compare quando termina la partita.

Il suo costruttore richiede una view ed un **matchtype**: la view serve di consueto per l’aggiornamento della grafica mentre il matchtype (ottenuto dalla partita appena terminata) serve nel caso in cui si clicchi **Rematch** in modo tale da far partire istantaneamente “la rivincita” con le stesse impostazioni della partita terminata.

se il giocatore selezionerà di giocare ancora l’endpanel verrà chiuso e il campo resettato (**addRematchListener()** >> resetGameStatus() >> spawn del campo tramite interactive/random controller) (**pic1**), che creerà controller nuovi e view nuove in base al tipo di partita ma mantenendo appunto le dimensioni (per questo motivo row e column sono dichiarate **statiche** in startPanelController).

altrimenti, cliccando sul tasto opzioni si ritornerà allo Startpanel, invocato dallo StartPanelController (**addOptionListener()** >> StartPanelController()>>resetGameStatus())(**pic2**)

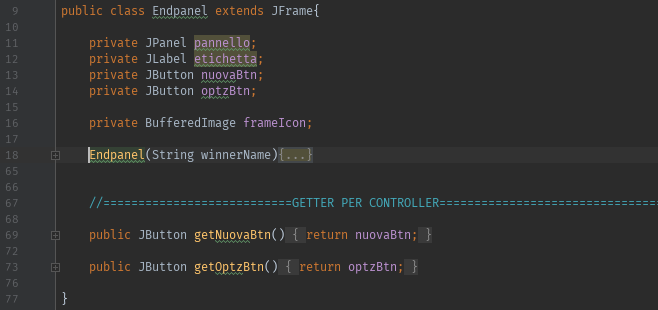
*pic1*

*pic2*

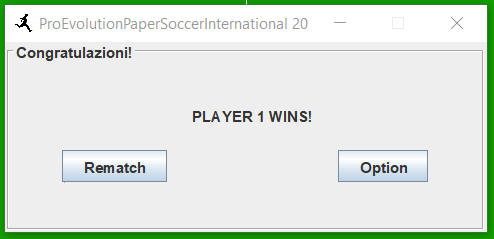
**view.Endpanel**

a partita finita comparirà il frame che annuncierà il vincitore e la possibilità di giocare ancora o di cambiare le impostazioni. le opzioni scelte dall’utente verranno comunicate all’EndPanelController che chiamerà di conseguenza i metodi adatti.

Come per lo StartPanel si tratta solo di una classe con componenti grafici quali 2 JButton e i rispettivi getter.



*La classe EndPanel*



*Schermata Endpanel*

**-Testing-**

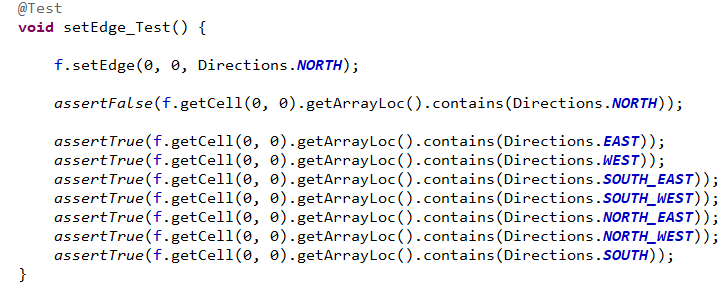
Per testare le componenti del programma si utilizza il framework Java JUnit, che ci consente di creare metodi di unit test con annotazioni utili a distinguerli dai metodi testati presenti nei package it.unicam.cs.pa.paperSoccer.model e it.unicam.cs.pa.paperSoccer.controller. Si è deciso di separare I metodi di testing da quelli testati mettendoli nel package it.unicam.cs.pa.paperSoccer.tests della cartella tests, per evitare di affollare le classi sorgente e agevolare le modifiche delle classi. Le classi di Testing sono composti da JUnit test, in particolare da Assert Method, che consentono di verificare la corrispondenza tra I valori prodotti dai metodi testati e I valori previsti, quali assert, assertTrue (che restituisce true se il suo argomento è true) e assertFalse(il contrario di assertTrue). I metodi di Testing sono:

-**Cell\_Test**: questa classe testa I metodi della classe Cell nel package it.unicam.cs.pa.paperSoccer.model, tra cui il metodo arrayLocIsFull, il quale si occupa di verificare se un array di tipo Directions con tutte le direzioni possibili viene considerato pieno, il metodo addDirections che testa se una direzione aggiunta nell’array di direzioni viene effettivamente aggiunta e il metodo matchDirections per controllare che una direzione che collega due celle è quella che ci si aspetta.

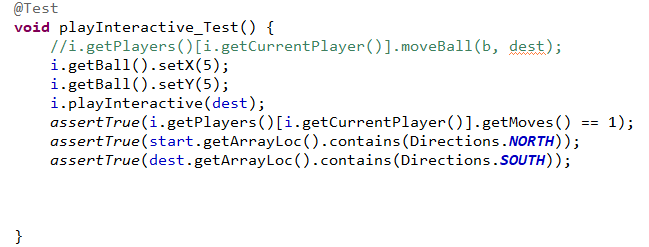
-**DefaultRuleSet\_Test**: questa classe si occupa di testare il metodo checkEdge\_Cell che controlla se la palla viene spostata in una cella che ha già una direzione nell’array delle direzioni, I metodi checkMove\_Dir e checkMove\_Cell che testano se lo spostamento della palla è consentito o no (ovvero se presente la direzione nei rispettivi array di direzioni), il metodo score\_Test che controlla se la palla è entrata in una porta e il metodo noMovesAvailable\_Test, che controlla se la palla è in una cella con l’array di direzioni pieno e che quindi non può muoversi.

-**Directions\_Test**: questa testa il metodo getOppositeDirections per vedere se data una direzione viene restituita l’opposta. Ha solo questo metodo perchè ritenuto più interessante degli altri poichè la classe contiene metodi Getters.

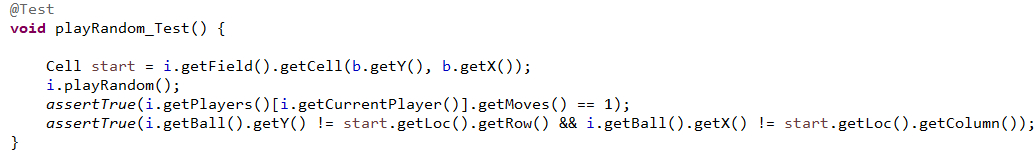
-**MatrixSoccerField\_Test**: questa classe prova il funzionamento dei metodi per l’impostazione delle celle nella matrice del campo, quindi il metodo setEdge\_Test che testa sela cella all’angolo passato come argomento contiene tutte le direzioni eccetto quella data, stessa cosa per le celle sul bordo destro, sinistro, quelle sul superiore e inferiore e per le celle davanti alle porte.



-**InteractiveMatch\_Test**: questa classe testa I metodi della classe InteractiveMatch nel package del model con il playInteractive\_Test, per testare il funzionamento del meccanismo sposta la palla, aggiorna gli array di direzioni delle celle interessate e gestione dei turni, e il checkDest per il testing del movimento della palla, controllando se il movimento è consentito con due assert che risultano veri nel caso del metodo poichè si è all’inizio della partita.

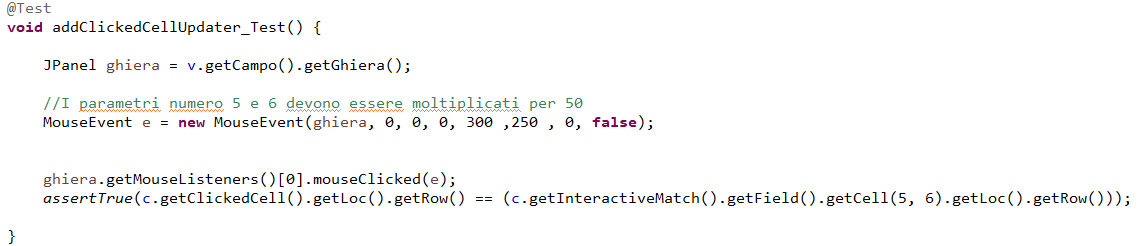


-**RandomMatch\_Test**: questa classe testa il metodo di gioco del Match randomico con playRandom\_Test, controllando se il turno sia terminato con il corretto spostamento della palla con due assertTrue.

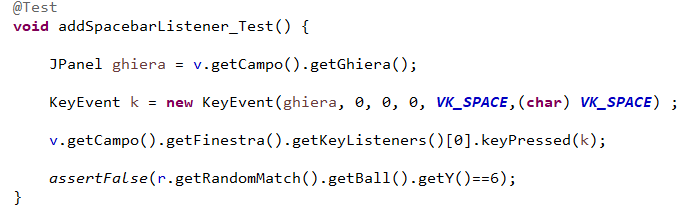


-**StartPanelController\_Test**: questa classe si occupa di verificare il funzionamento del play button della classe startPanel nel package it.unicam.cs.pa.paperSoccer.controller, verificando che le dimensioni del campo corrispondano a quelle passate in questo caso con variabili “fisse” e che inizi la partita randomica se l’utente ha cliccato la spunta PCvsPC o quella normale se ha cliccato sulla spunta 1vs1.

-**InteractiveMatchController\_Test**: questa classe testa il funzionamento dei metodi responsabili del controllo del click del Mouse convertito in clickedCell, verificando che la riga di essa corrispondi alla riga che ci si aspetta con assertTrue (metodo addClickedCellUpdater\_Test) e del controllo del gioco e aggiornamento grafica solo se il movimento è valido (in questo caso è il primo movimento quindi è ovviamente valido) con assert che assicurano lo spostamento della palla nella cella di destinazione.



-**RandomMatchController\_Test**: questa classe prova il funzionamento del metodo di spostamento della palla e aggiornamento grafica, non è presente il controllo della cella cliccata poichè il pc sposta la palla solo dove può andare e in questo caso la palla si sposta nella cella superiore a quella della palla quindi l’assert è rispettato.



-**EndPanelController\_Test**: questa classe si occupa di testare la classe EndPanelController con il metodo addRematchListener\_Test, controllando che si crei una nuova partita del tipo selezionato con un assert per ogni tipo di match, e con il metodo addOptionListener che crea uno startPanel nel caso l’utente dovesse selezionare l’opzione Option.