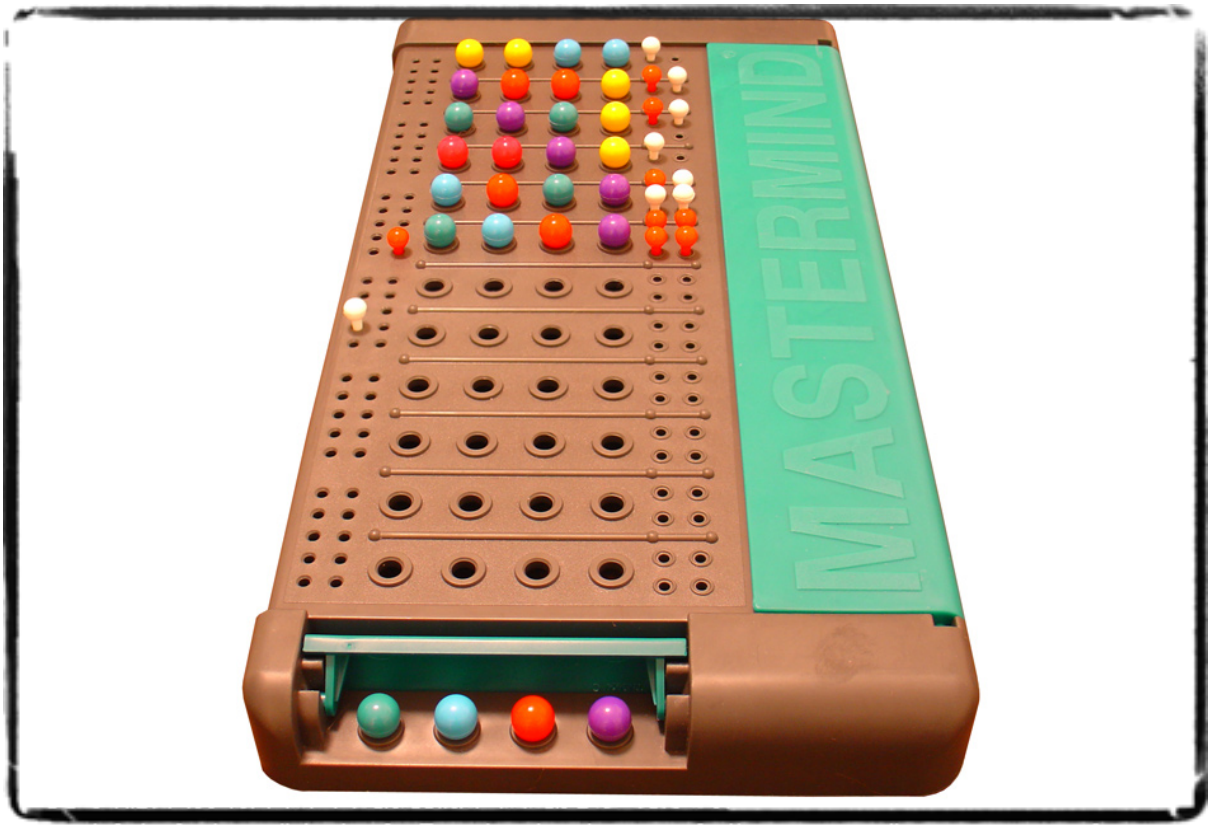
Relazione del Progetto di Programmazione Avanzata

Mastermind



Edoardo Papa

Anno Accademico 2018-2019

Corso di Laurea in Informatica

Università di Camerino

Prof. Michele Loreti

Introduzione

Che cos'è mastermind?

Il MasterMind è un gioco nato nel 1970 e, benché molto semplice da comprendere e da giocare, necessita di un'elevata concentrazione e un'ottima capacità di ragionamento.
Due sono i giocatori: il primo ha a disposizione sei piroli di colori diversi e quattro spazi da riempire. Senza farsi vedere (sul tavolo da gioco c'è un'apposita parte nascosta), sceglie quali colori utilizzare e in quale ordine. L'altro giocatore deve indovinare la serie esatta inserita. L'avversario gli comunica quanti colori corretti sono al posto giusto, quanti colori giusti ma al posto sbagliato.
Solitamente si hanno a disposizione dieci tentativi per indovinare, dopodiché il gioco termina.

Ecco le istruzioni essenziali che illustreranno come giocare a MasterMind:

- Uno dei due giocatori sceglie e inserisce i colori da utilizzare e l'ordine in cui compaiono, utilizzando l'apposito spazio nascosto all'avversario. La scelta del colore e della posizione dei pinoli è puramente casuale.

- Il secondo giocatore deve provare ad ipotizzare la serie creata.
L'altro, gli comunica quanti colori sono giusti e al posto corretto, quanti colori giusti ma al posto sbagliato.

Piccolo Esempio:
Se per esempio la serie è rosso-blu-verde-giallo e il giocatore che deve indovinare ipotizza rosso-bianco-blu-nero, il primo dovrà dirgli: "Un colore giusto nella posizione corretta, un colore giusto nella posizione sbagliata".
Successivamente il giocatore che deve indovinare continuerà a cercare la combinazione giusta fino a che o indovinerà il codice o terminerà i tentativi disponibili.

Introduzione

Per questo progetto è stato utilizzato il design Pattern Model-View-Controller (MVC).

Come si intuisce facilmente dal nome, tale pattern è costituito da 3 strati :

- MODEL
- VIEW
- CONTOLLER

Il model è il componente centrale de pattern che gestisce i dati, la logica dell'applicazione.

La view può essere qualsiasi rappresentazione di output delle informazioni, come un grafico, diagramma ecc.

Il controller prende l'input dalla vista e lo converte in comandi/dati per il modello.

Nel mio progetto:

CONTROLLER

- Controller.java
- Coordinator.java
- Initializatior.java

VIEW

- MainView.java

MODEL

- Game.java
- Row.java
- SetRules.java

Responsabilità e Architettura delle Classi

Controller.java

La classe controller definita all'interno del package *it.unicam.cs.pa.controller* è una classe che è in grado di prendere degli input provenienti dalla view.

La classe controller è utilizzata quando vogliamo che l'utente ci restituisca dei dati. In questa classe sono presenti piu metodi per la richiesta di dati da parte dell'utente. Ognuno di questi ha un compito ben preciso, come ad esempio la restituzione di un INT oppure di un LISTA ecc.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe Controller.java ha al suo interno:

Attributi:

- private ViewInterface view

  Rappresenta l'interfaccia grafica del gioco

Metodi:

- public Controller(ViewInterface view)

  Rappresenta il costruttore della classe controller che ha come parametro formale l'interfaccia grafica

- public ViewInterface getView()

  Questo metodo permette di restituire l'interfaccia grafica.

- public ArrayList<String> getDataFromPlayer()

  Questo metodo ci permette di restituire una lista di dati inseriti dall'utente. Tipicamente è utilizzata quando l'utente inserisce l'insieme dei "piroli".(ESEMPIO DI DATO INSERITO DALL'UTENTE : *R,V,G,B*)

- public int getDataFromConsole(int maxRange)

  Questo metodo ci permette di restituire il numero intero inserito dall'utente. *MaxRange* definisce il massimo numero che l'utente potrà inserire (numeri da *0* a *maxRange*).

- public String getStringFromConsole()

Questo metodo ci permette di restituire una stringa inserita dall'utente. Tipicamente è utilizzata quando l'utente deve inserire il nome di un giocatore.

Coordinator.java

La classe coordinator definita all'interno del package *it.unicam.cs.pa.controller* è una classe che coordina il turno della partita.

La classe Coordinator è utilizzata quando, dopo aver inizializzato la partita con le varie regole, tipi di giocatori e la creazione della riga da decodificare, vogliamo far eseguire il turno successivo di gioco.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe Coordinator.java ha al suo interno:

Attributi:

- private Player[] players

Rappresenta l'insieme dei giocatori.

- private int turno

Rappresenta il numero del turno attuale.

- private StrategyInterface strategy;

Rappresenta la strategia che il giocatore-codebreaker PC andrà ad utilizzare. Il tipo di strategia è scelto dall'utente.

- private GameDifficulty difficulty;

Rappresenta la difficoltà del gioco. La difficoltà è scelta dell'utente.

- private Game game;

Rappresenta la parte computazionale del nostro gioco, la quale ci permetterà di aggiungere righe al nostro tavolo da gioco, verificare la correttezza della riga inserita.

- private ViewInterface view

Rappresenta l'interfaccia grafica del gioco.

Metodi:

- public Coordinator(Game game, ViewInterface view)

Rappresenta il costruttore della classe coordinator che ha come parametro formale il game e l'interfaccia grafica.

- public boolean nextTurn()

Questo metodo permette di eseguire il turno di una partita. In particolare, identifico se ho come giocatore-codebreaker il giocatore PC, in caso affermativo, a seconda della strategia che può adottare mi ricavo la riga da aggiungere al tavolo da gioco, in caso negativo, sarà l'utente a digitare la riga da inserire.

- public int getTurno()

Questo metodo permette di restituire in numero del turno attuale.

Initializatior.java

La classe initializatior definita all'interno del package *it.unicam.cs.pa.controller* permette di inizializzare la partita con tutte le informazioni basi definite dall'utente.

La classe Initializatior è utilizzata per inizializzare la partita. All'interno di questa classe verranno definiti i tipi di giocatore *(UMANO vs PC / PC vs PC / UMANO vs UMANO)* , la difficoltà del gioco (attraverso la quale verranno definiti in particolare il massimo numero di tentativi e i colori disponibili) e la strategia del giocatore PC se scelto il giocatore PC come codebreaker.

Tutte le informazioni precedentemente descritte sono scelte solo ed esclusivamente dall'utente.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe Initializatior.java ha al suo interno:

Attributi:

- private Player player1

Rappresenta il giocatore 1

- private Player player2

Rappresenta il giocatore 2

- private GameDifficulty difficulty;

Rappresenta la difficoltà del gioco. La difficoltà è scelta dell'utente.

- private Controller IOComunication

Rappresenta il controller che ci permetterà di comunicare con l'utente.

- private Row mainKey

Rappresenta la riga di "piroli" che il codebreaker dovrà decodificare.

- private StrategyInterface strategy;

Rappresenta la strategia che il giocatore-codebreaker PC andrà ad utilizzare. Il tipo di strategia è scelto dall'utente.

- private Game game;

Rappresenta la parte computazionale del nostro gioco, la quale ci permetterà di aggiungere righe al nostro tavolo da gioco, verificare la correttezza della riga inserita.

Metodi:

- public Initializatior(Controller controll, Game game)

Rappresenta il costruttore della classe initializatior che ha come parametro formale il game e il controller. All'interno del costruttore andremo a creare i tipi di giocatori scelti dall'utente, la difficoltà scelta dall'utente e infine a seconda del tipo di giocatore codebreaker andremo o meno a selezionare la strategia (solo il giocatore PC può avere una strategia).

- private void createGame()

Questo metodo permette di creare la partita a seconda delle scelte che l'utente ha deciso. In particolare ci permette di creare : *UMANO vs PC / PC vs PC / UMANO vs UMANO.* Definendo anche chi sarà il *CODEMAKER* e chi il *CODEBREAKER.*

- private void PcVsPc()

Questo metodo permette di creare una partita con soli giocatori PC. Quindi sarà il PC a creare il codice da decodificare e sarà sempre il PC a decodificarlo a seconda della strategia scelta dall'utente.

- private void humanVsHuman()

Questo metodo permette di creare una partita con soli giocatori PLAYER. Quindi sarà il PLAYER a creare il codice da decodificare e sarà sempre un PLAYER a decodificarlo.

- private void pcVsHuman()

Questo metodo permette di creare una partita con un giocatore PLAYER e un giocatore PC. A seconda della scelta dell'utente verrà definito chi dei due sarà il *CODEMAKER* e chi il *CODEBREAKER.*

- private void setMainKey()

Questo metodo permette all'utente di inserire la riga da decodificare.

- private void setDifficulty()

Questo metodo permette all'utente di selezionare la difficoltà.

- private void getPlayer()

Questo metodo permette di mostrare l'elenco dei giocatori *CODEMAKER* e *CODEBREAKER.* I giocatori vengono creati attraverso il metodo *createPlayer*. Nel caso in cui il

CODEBREAKER fosse il PC permette all'utente di decidere la strategia. (La lista dei giocatori disponibili è definita all'interno di *PlayerArchive*)

- private void setStrategy()

Questo metodo permette di mostrare l'elenco delle strategie per il giocatore *CODEBREAKER* di tipo PC. La strategia da adottare viene creata attraverso il metodo *createStrategy*. (La lista dei giocatori delle strategie disponibili è definita all'interno di *StrategyArchive*)

- private StrategyInterface createStrategy(List<String> strategyType)

Questo metodo permette di creare la strategia per il giocatore *CODEBREAKER* di tipo PC a seconda della strategia scelta dall'utente. (La lista dei giocatori delle strategie disponibili è definita all'interno di *StrategyArchive*)

- private Player createPlayer(int i, List<String> playerTyper, TypeGamePlayer type)

Questo metodo permette di creare il tipo di giocatore facendolo selezionare dall'utente. (La lista dei giocatori disponibili è definita all'interno di *PlayerArchive*)

- public Row getMainKey()

Questo metodo permette di ritornare la riga (l'insieme di "piroli") da decodificare.

Observable.java

La classe observable definita all'interno del package *it.unicam.cs.pa.designPattern* permette di definire una relazione uno-a-molti tra gli oggetti senza rendere gli oggetti strettamente accoppiati. Observable è una classe che può essere osservata e all'interno di ogni Observable possiamo avere più osservatori.

La classe Observable<T extends ViewInterface> è utilizzata per definire una classe che può essere osservabile. Utilizzo il designPattern Observer-Observable per creare una relazione tra la classe Game e la classe Mainview. Questo perché quando io all'interno della mia classe game, vado ad eseguire l'aggiunta di una riga al mio "tavolo da gioco", devo andare ad aggiungere tale riga anche nell'interfaccia grafica. Observable<T extends ViewInterface> può accettare qualsiasi tipo di oggetto T purché sia del sottotipo ViewInterface.

Generalmente quando una determinata istanza di un oggetto Observable cambia e si vuole notificare questo cambiamento, si richiama il metodo *setChanged()* e successivamente *notifyObservers()*.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe Observable.java ha al suo interno:

Attributi:

- private boolean change

  Identifica se è avvenuto un cambiamento

- private List<T> listOfObserver

  Rappresenta la lista degli observer che osservano l'oggetto observable.

Metodi:

- public Observable()

  Rappresenta il costruttore della classe Observable.

- public void addObserver(T obs)

  Questo metodo permette di aggiungere un Observer alla lista interna degli observer.

- public void deleteObserver(Observer obs)

  Questo metodo permette di eliminare dalla lista interna degli observer un observer passato come parametro formale.

- public int countObserver()

  Questo metodo permette di conoscere quanti observer sono presenti della lista interna.

- public void setChanged()

  Questo metodo permette di impostare a TRUE il valore del change che identifica se è avvenuto un cambiamento.

- public void clearChanged()

  Questo metodo permette di impostare a FALSE il valore del change che identifica se è avvenuto un cambiamento.

- public void hasChanged()

  Questo metodo permette di far ritornare il valore del flag change che identifica se è avvenuto un cambiamento.

- public void notifyObservers(ArrayList<Row> arg)

  Questo metodo controlla il flag change per verificare che sia avvenuto un cambiamento e successivamente, se questo è avvenuto, richiama il metodo *update()* di tutti gli observer presenti nella lista interna.

Observer.java

L'interfaccia observer definita all'interno del package *it.unicam.cs.pa.designPattern* permette di definire una relazione uno-a-molti tra gli oggetti senza rendere gli oggetti strettamente accoppiati. Observer è un'interfaccia che definisce un oggetto che può osservare.

L'interfaccia Observer è utilizzata per definire un oggetto osservatore. Utilizzo il designPattern Observer-Observable per creare una relazione tra la classe Game e la classe Mainview. Questo perché quando io all'interno della mia classe game vado ad eseguire l'aggiunta di una riga al mio "tavolo da gioco", devo eseguire questa aggiunta anche nell'interfaccia grafica.

A seguire troveremo la spiegazione di ogni metodo.

L'interfaccia Observer.java ha al suo interno:

Metodi:

- public void update(ArrayList<Row> arg)

Questo metodo verrà richiamato quando si sarà verificato un cambiamento nello stato di un observable.

BadFileException.java

La classe BadFileException definita all'interno del package *it.unicam.cs.pa.exception* permette di definire un'eccezione che verrà lanciata nel momento in cui il file preso in considerazione da una classe non rispetti determinati parametri. Ad esempio se all'interno del file per definire le regole "/rulesData/DefaultRules" abbiamo un label di questo genere:

```
LABEL=NOME_DIFFICOLTA
NUMBER_OF_COLORS=8
AVAIABLE_COLORS=R,B,G,Y,V,O,C,W,Q,X,J
NUMBER_OF_TRY=9
```

Verrà sicuramente lanciata questo tipo di eccezione poiché abbiamo scritto in *"AVAIABLE_COLORS"* 11 colori, mentre nel campo *"NUMBER_OF_COLORS"* abbiamo detto che ne abbiamo solamente 8.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe BadFileException.java ha al suo interno:

Attributi:

- private static final long serialVersionUID = 1L;

Metodi:

- public BadFileException(String message)

FileOpenErrorException.java

La classe FileOpenErrorException definita all'interno del package *it.unicam.cs.pa.exception* permette di definire un'eccezione, che verrà lanciata nel momento in cui non riusciremo ad aprire il file preso in considerazione da una classe.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe FileOpenErrorException.java ha al suo interno:

Attributi:

- private static final long serialVersionUID = 1L;

Metodi:

- public FileOpenErrorException(String message)

MismatchException.java

La classe MismatchException definita all'interno del package *it.unicam.cs.pa.exception* permette di definire un'eccezione, che verrà lanciata nel momento in cui inseriamo dei dati non corretti. Ad esempio se si sta attendendo un *INT,* ma ci viene passato uno *STRING*, viene lanciata questa eccezione.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe MismatchException.java ha al suo interno:

Attributi:

- private static final long serialVersionUID = 1L;

Metodi:

- public MismatchException(String message)

RegistrySysntaxException.java

La classe RegistrySysntaxException definita all'interno del package *it.unicam.cs.pa.exception* permette di definire un'eccezione, che verrà lanciata nel momento in cui non sia rispettato il formato dei file testuali personificabili.

File testuali personificabili :

1) /rulesData/
2) /playerDataCreation/
3) /strategyDataCreation/

All'interno del file /standard/info.txt sono definiti i vari formati dei file precedenti.

Quindi se non dovesse essere rispettato tale formato viene lanciata questa eccezione.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe RegistrySysntaxException.java ha al suo interno:

Attributi:

- private static final long serialVersionUID = 1L;

Metodi:

- public RegistrySysntaxException(String message)

GeneralException.java

La classe GeneralException definita all'interno del package *it.unicam.cs.pa.exception* permette di definire un'eccezione generale. Tutte le eccezioni precedentemente descritte estendono questa classe GeneralException che definisce una tipologia di eccezione generale.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe GeneralException.java ha al suo interno:

Attributi:

- private static final long serialVersionUID = 1L;

Metodi:

- public GeneralException(String message)

GameDifficulty.java

L'interfaccia GameDifficulty definita all'interno del package *it.unicam.cs.pa.mode* permette di definire un contratto, che indica le caratteristiche principali che devono essere rispettate da ogni difficoltà di gioco.

A seguire troveremo la spiegazione di ogni metodo.

L'interfaccia GameDifficulty.java ha al suo interno:

Metodi:

- public String getName()

  Questo metodo ritorna il nome della difficoltà

- public int getSlots()

  Questo metodo ritorna il numero massimo di tentativi che possono essere fatti per risolvere il codice.

- public List<String> getAvaiableColors()

  Questo metodo ritorna la lista dei colori disponibili.

- public String getAvaiableColorsToString()

  Questo metodo ritorna la lista dei colori disponibili sotto forma di stringa.

GameMode.java

La classe GameMode definita all'interno del package *it.unicam.cs.pa.mode* permette di definire l'insieme delle informazioni come -numero_di_tentativi, -colori_disponibili,-nome_della_difficolta a seconda della modalità di gioco che l'utente vuole creare.

La classe GameMode estende l'interfaccia GameDifficulty, quindi deve rispettare il contratto definito da tale interfaccia.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe GameMode.java ha al suo interno:

Attributi:

- private String name

  Identifica il nome della difficoltà.

- private int numberOfColor

  Identifica il numero dei colori disponibili

- private int numberOfTry

  Identifica il numero di tentativi che si possono effettuare per risolvere il codice.

- private List<String> colors

  Identifica la lista di tutti i colori disponibili.

Metodi:

- public GameMode(String string, int numberOfcolor, List<String> colors, int numberOfTry)

Questo metodo rappresenta il costruttore della classe che prende come parametri formali il nome della difficoltà di gioco, il numero di colori disponibili, la lista dei colori disponibili e il numero di tentativi per risolvere il codice.

Gli altri metodi di questa classe sono i metodi definiti dall'interfaccia *GameDifficulty* quindi la loro descrizione è stata trattata precedentemente.

Game.java

La classe Game definita all'interno del package *it.unicam.cs.pa.model* permette di definire la parte computazionale del nostro gioco.

Tale classe permette di aggiungere al nostro tavolo da gioco nuove righe inserite dall'utente se rispettano determinati criteri. Ad esempio se la lunghezza della riga inserita è minore o maggiore di 4, allora la riga non viene inserita. Oppure, se la riga inserita ha al suo interno colori non definiti dall'oggetto creato di tipo *GameMode,* allora la riga non viene inserita.

La classe Game estende Observable<ViewInterface>. Questo definisce che la classe Game è un oggetto che può essere osservato e potrà essere osservato da observer che sono di tipo ViewInterface.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe Game.java ha al suo interno:

Attributi:

- private GameDifficulty mode

  Identifica la difficoltà del gioco.

- private ArrayList<Row> gameBoard

  Identifica il tavolo da gioco.

- private Row mainKey

  Identifica il codice da dover andare a decodificare

- private Player player1

  Identifica il giocatore 1

- private Player player2

  Identifica il giocatore 2

- private StrategyInterface strategy

  Identifica la strategia del giocatore PC

- private Controller controller

  Identifica il controllore del gioco che è in grado di prendere dati dall'interfaccia grafica.

Metodi:

- public Game(Controller controller)

  Questo metodo è il costruttore del Game che prende come parametro formale un controller.

- public boolean checkCorrectRow(Row row)

  Questo metodo permette di analizzare la riga inserita dall'utente e valutare se la riga inserita ha al suo interno valori non ammessi. Tale metodo ritorna TRUE se la riga è corretta, altrimenti FALSE.

- public boolean addRow(Row actualyRow, Row key)

  Questo metodo permette analizzare e di verificare che la riga actualyRow è una riga corretta e in caso affermativo l'aggiunge al tavolo da gioco e notifica tutti gli observer del cambiamento.

- public Row getMainKey()

  Questo metodo permette di ritornare la riga da decodificare.

- public gameDifficulty getMode()

  Questo metodo permette di ritornare la difficolta del gioco.

- public int getSizeOfGameBoard()

  Questo metodo permette di ritornare la grandezza del tavolo da gioco.

- public boolean isSolved()

  Questo metodo permette di capire se la riga da decodificare è stata risolta o meno.

- public Player getPlayer1()

  Questo metodo permette di ritornare il giocatore 1.

- public Player getPlayer2()

  Questo metodo permette di ritornare il giocatore 2.

- public StrategyInterface getStrategy()

  Questo metodo permette di ritornare la strategia del giocatore PC.

- public ArrayList<Row> getGameBoard()

Questo metodo permette di ritornare il tavolo da gioco.

- public void setMode(GameDifficulty difficoltà)

Questo metodo permette di impostare la difficoltà del gioco.

- public void setPlayer(Player player1, Player player2)

Questo metodo permette di impostare i due giocatori.

- public void setStrategy(StrategyInterface stra)

Questo metodo permette di impostare la strategia che dovrà adoperare il giocatore PC.

- public void setMainKey(Row mainKey)

Questo metodo permette di impostare la riga da decodificare.

Row.java

La classe Row definita all'interno del package *it.unicam.cs.pa.model* permette di definire la riga contenente i "piroli".

Tale classe ci definisce una riga contente i piroli e memorizza il numero dei colori corretti ma che si trovano nella posizione sbagliata e dei colori corretti che si trovano nella posizione giusta.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe Row.java ha al suo interno:

Attributi:

- private int numberOfCorrectColor

Identifica il numero dei colori corretti ma che si trovano nella posizione sbagliata.

- private int numberOfCorrectPositionColor

Identifica il numero dei colori corretti che si trovano nella giusta posizione.

- private ArrayList<String> holes

Identifica i piroli all'interno della riga.

Metodi:

- public Row(ArrayList<String> holes)

Questo metodo è il costruttore della ROW che permette di aggiungere i piroli alla riga passandogli direttamente una lista "prefatta".

- public Row check(Row key)

Questo metodo permette di analizzare la riga this e la riga passata come parametro formale (riga da decodificare) e vedere quanti colori abbiamo indovinato senza però indovinare la posizione e di quanti piroli abbiamo indovinato sia il colore sia la posizione.

- public int size()

Questo metodo permette di ritornare la grandezza della riga.

- public String toString()

Questo metodo permette di ritornare una stringa che rappresenta la nostra riga.

- public int getNumberOfCorrectPosition()

Questo metodo permette di ritornare il numero di colori che abbiamo indovinato sia il colore che la posizione.

- public int getNumberOfCorrectColor()

Questo metodo permette di ritornare il numero di colori che abbiamo indovinato il colore ma non la posizione.

- public ArrayList<String> getHoles()

Questo metodo permette di ritornare la lista dei piroli.

Player.java

L'interfaccia Player definita all'interno del package *it.unicam.cs.pa.player* permette di definire un contratto che definisce le caratteristiche principali che devono essere rispettate da qualsiasi tipo di giocatore.

A seguire troveremo la spiegazione di ogni metodo.

L'interfaccia Player.java ha al suo interno:

Metodi:

- public String getNameOfPlayer()

Questo metodo restituisce il nome del giocatore.

- public TypePlayer getTypeOfPlayer()

Questo metodo restituisce l'enum *TypePlayer* che identifica il tipo di giocatore *(PC / PLAYER)*.

- public TypeGamePlayer getTypeGamePlayer()

Questo metodo restituisce l'enum TypeGamePlayer che identifica la tipologia di giocatore *(CODEMAKER / CODEBREAKER)*.

PlayerFactory.java

L'interfaccia PlayerFactory definita all'interno del package *it.unicam.cs.pa.player* permette di definire un'interfaccia funzionale che ci permetterà di creare i giocatori.

A seguire troveremo la spiegazione di ogni metodo.

L'interfaccia Player.java ha al suo interno:

Metodi:

- public Player createPlayer(String name,TypeGamePlayer type)

Questo metodo permette di restituire il giocatore creato. Prende come parametri formali il nome del giocatore e la tipologia del giocatore *(CODEMAKER / CODEBREAKER)*.

HumanPlayer.java

La classe HumanPlayer definita all'interno del package *it.unicam.cs.pa.player* permette di definire un giocatore di tipo umano con le caratteristiche di qualsiasi giocatore, infatti questa classe implementa l'interfaccia precedentemente descritta *Player.*

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe HumanPlayer.java ha al suo interno:

Attributi:

- private String name

Identifica il nome del giocatore.

- private final TypePlayer TYPE_OF_PLAYER

Identifica il tipo di giocatore *(PC / PLAYER)*.

- private final TypeGamePlayer TYPE_GAME_PLAYER

Identifica la tipologia di giocatore *(CODEMAKER / CODEBREAKER)*.

Metodi:

I metodi di questa classe sono i metodi definiti dall'interfaccia *Player;* la loro descrizione è stata trattata precedentemente.

PcPlayer.java

La classe PcPlayer definita all'interno del package *it.unicam.cs.pa.player* permette di definire un giocatore di tipo computer con le caratteristiche di qualsiasi giocatore, infatti questa classe implementa l'interfaccia precedentemente descritta *Player.*

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe PcPlayer.java ha al suo interno:

Attributi:

- private String name

  Identifica il nome del giocatore.

- private final TypePlayer TYPE_OF_PLAYER

  Identifica il tipo di giocatore *(PC / PLAYER)*.

- private final TypeGamePlayer TYPE_GAME_PLAYER

  Identifica la tipologia di giocatore *(CODEMAKER / CODEBREAKER)*.

Metodi:

I metodi di questa classe sono i metodi definiti dall'interfaccia *Player;* la loro descrizione è stata trattata precedentemente.

N.B.: Queste due classi *HumanPlayer* e *PcPlayer* sono apparentemente identiche a parte per il fatto che una ha come *TypePlayer* PLAYER (HumanPlayer) e l'altra ha come *TypePlayer* PC (PcPlayer). Queste due classi sono state volontariamente separate perché in un futuro si potrebbero aggiungere a tali classi delle caratteristiche o delle funzionalità in più. Ad esempio per la classe HumanPlayer si potrebbero immagazzinare più informazioni oltre che al nome, come ad esempio il cognome, il nikename e magari a seconda del nikename inserito verificare quante volte tale nikename ha vinto.

HumanPlayerFactory.java

La classe HumanPlayerFactory definita all'interno del package *it.unicam.cs.pa.player* permette di creare un giocatore di tipo HumanPlayer. Questa classe implementa l'interfaccia funzionale PlayerFactory.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe HumanPlayerFactory.java ha al suo interno:

Metodi:

- public Player createPlayer(String name,TypeGamePlayer type)

  Questo metodo restituisce il nuovo giocatore creato di tipo PLAYER.

PcPlayerFactory.java

La classe PcPlayerFactory definita all'interno del package *it.unicam.cs.pa.player* permette di creare un giocatore di tipo PcPlayer. Questa classe implementa l'interfaccia funzionale PlayerFactory.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe PcPlayerFactory.java ha al suo interno:

Metodi:

- public Player createPlayer(String name,TypeGamePlayer type)

  Questo metodo restituisce il nuovo giocatore creato di tipo Pc.

PlayerArchive.java

La classe PlayerArchive definita all'interno del package *it.unicam.cs.pa.player* definisce un archivio che contiene tutti i tipi di giocatore che possono essere creati. Tali giocatori sono "presi" dal file di testo "/playerDataCreation/data.txt".

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe PlayerArchive.java ha al suo interno:

Attributi:

- private static PlayerArchive instance

  Identifica l'istanza della classe.

- private Map<String,PlayerFactory> archive

  Identifica un contenitore che racchiude tutti i giocatori disponibili. Tale archivio di coppie di elementi sono contraddistinti da una chiave e dal valore ad essa associato. La chiave è ciò che identifica univocamente un elemento, quindi non è possibile avere due elementi nel nostro archivio con lo stesso valore di chiave.

- private List<String> nameTypeOfPlayer

Identifica la lista contenente i nomi di tutti i giocatori disponibili.

Metodi:

- private PlayerArchive()

Questo è il costruttore della nostra classe il quale ci permette di far "partire" l'inizializzazione del nostro archivio.

- private void inizializate()

Questo metodo permette di inizializzare l'archivio con i dati contenuti all'interno del path "/playerDataCreation/data.txt".

- public static PlayerArchive getInstance()

Questo metodo permette di restituire l'istanza della mia classe. Nel caso fosse nulla, ne creo una nuova.

- public PlayerFactory getPlayer(String nameTypeOfPlayer)

Questo metodo restituisce il giocatore all'interno del nostro archivio a seconda del nome inserito come parametro formale.

- public void loadPlayer(File file)

Questo metodo permette di caricare un file testuale, contenente i giocatori, passato come parametro formale.

- private void register(String registryLine)

Questo metodo permette di "registrare" un giocatore. Tale metodo richiama a sua volta un altro metodo per la registrazione del giocatore all'interno dell'archivio, infatti questo metodo si limita a prendere come parametro formale una stringa e di dividerla in due sottostringhe, che verranno poi passate al successivo metodo per la registrazione. La stringa passata come parametro formale, deve avere la seguente caratteristica : "nome_giocatore nome_classe". Quindi tra il nome del giocatore e il nome della classe deve esserci solamente uno spazio vuoto che li divide. Nel caso in cui non sia rispettato il formato definito precedentemente e descritto all'interno del file "/standard/info.txt" viene lanciata un'eccezione e il gioco termina.

- private void register(String name, String className)

Questo metodo permette di "registrare" un giocatore a seconda delle due stringhe passate come parametro formale. La prima identifica il nome del giocatore, la seconda il nome della classe. All'interno di questo metodo abbiamo un'ulteriore chiamata a un altro metodo che effettuerà la vera e propria registrazione all'interno dell'archivio dei giocatori.

- private void register(String name, PlayerFactory newInstance)

Questo metodo permette di aggiungere all'archivio dei giocatori un nuovo giocatore. Inoltre, ci permette di aggiungere alla lista dei nomi dei giocatori il nome del giocatore aggiunto.

- public List<String> getNameTypeOfPlayer()

Questo metodo restituisce la lista che contiene i nomi dei giocatori disponibili.

TypeGamePlayer.java

L'enum TypeGamePlayer definita all'interno del package *it.unicam.cs.pa.player* definisce la tipologia dei giocatori.

La tipologia dei giocatori :

- *CODEBREAKER*
- *CODEMAKER*

Il codebreaker è colui che dovrà decodificare la riga.

Il codemaker è colui che dovrà creare la riga da decodificare.

TypePlayer.java

L'enum TypePlayer definita all'interno del package *it.unicam.cs.pa.player* definisce il tipo dei giocatori.

Tipo di giocatori :

- *PC*
- *PLAYER*

Il PC è il computer.

Il PLAYER è qualsiasi giocatore umano.

Rules.java

L'interfaccia Rules definita all'interno del package *it.unicam.cs.pa.rules* permette di definire un contratto, che indica le caratteristiche principali che devono essere rispettate da qualsiasi classe in grado creare delle regole di gioco.

A seguire troveremo la spiegazione di ogni metodo.

L'interfaccia Rules.java ha al suo interno:

Metodi:

- public List<String> getAvaiableColors()

  Questo metodo restituisce la lista contenente i colori disponibili.

- public int getNumberOfSlots()

  Questo metodo restituisce il numero di tentativi che è possibile fare.

SetRules.java

La classe SetRules definita all'interno del package *it.unicam.cs.pa.rules* definisce l'insieme delle regole adottate durante la partita. Tali regole possono essere scelte dall'utente selezionando quale file utilizzare, tra i file presenti all'interno della cartella "/rulesData/".

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe SetRules.java ha al suo interno:

Attributi:

- private static SetRules instance

  Identifica l'istanza della classe.

- private int numberOfColor

  Identifica il numero di colori disponibili.

- private List<String> colors

  Identifica la lista contenente i colori disponibili.

- private int numberOfTry

  Identifica il numero massimo di tentativi che possono essere fatti.

- private List<String> mode

  Identifica la lista contenente le varie difficoltà.

- private int numberOfFileSelected

  Identifica il numero del file selezionato dall'utente.

- private String[] listOfFile

  Identifica la lista con tutti i file contenuti della cartella "/rulesData/"

- private ViewInterface view

  Identifica l'interfaccia grafica.

Metodi:

- private SetRules(ViewInterface passedView)

Questo è il costruttore della classe che mi permette di leggere, impostare e far selezionare all'utente le regole di gioco presenti nel path "/rulesData/".

- private void readAndSelectFile()

Questo metodo mi permette di leggere tutti i file presenti della cartella "/rulesData/"

- private void userSelectFile()

Questo metodo permette di far selezionare all'utente un file tra quelli presenti nella cartella "/rulesData/".

- private static void createInstance(ViewInterface view)

Questo metodo permette di creare l'oggetto SetRules.

- public static SetRules getInstance(ViewInterface view)

Questo metodo permette di ritornare l'istanza dell'oggetto SetRules. N.B.: Nel caso in cui l'istanza fosse nulla allora viene creato un nuovo oggetto.

- public static SetRules getInstance()

Questo metodo restituisce l'istanza dell'oggetto SetRules.

- public List<String> getMode()

Questo metodo restituisce la lista delle varie difficoltà

- public GameDifficulty setDifficulty(int data)

Questo metodo restituisce la difficoltà del gioco a seconda del numero del file selezionato dall'utente. Permette inoltre di controllare se ci sono delle incoerenze nel file scelto ed è in grado di sollevare un'eccezione.

- private void readFileAndSetData(BufferedReader fileIn, int data)

Questo metodo permette di leggere i dati provenienti dal file selezionato dall'utente e setta le regole di gioco a seconda della LABEL scelta dall'utente.

- private BufferedReader openFileFromString(String path)

Questo metodo permette di aprire un file attraverso il path passato come parametro formale e inoltre permette di restituire il flusso di dati contenuti all'interno del file. Questo metodo può lanciare un'eccezione se non è in grado di trovare/aprire il file.

Ulteriori metodi presenti in questa classe sono i metodi definiti dall'interfaccia *Rules;* la loro descrizione è stata trattata precedentemente.

StrategyInterface.java

L'interfaccia StrategyInterface definita all'interno del package *it.unicam.cs.pa.strategy* permette di definire un contratto, che indica le caratteristiche principali che devono essere rispettate da qualsiasi tipo di strategia.

A seguire troveremo la spiegazione di ogni metodo.

L'interfaccia StrategyInterface.java ha al suo interno:

Metodi:

- public Row getStrategy()

  Questo metodo restituisce la riga risultante a seconda della strategia applicata.

StrategyFactory.java

L'interfaccia StrategyFactory definita all'interno del package *it.unicam.cs.pa.strategy* definisce un'interfaccia funzionale con la quale possiamo creare le strategie.

A seguire troveremo la spiegazione di ogni metodo.

L'interfaccia StrategyFactory.java ha al suo interno:

Metodi:

- public StrategyInterface createStrategy()

  Questo metodo permette di creare una strategia.

KnuthStrategy.java

La classe KnuthStrategy definita all'interno del package *it.unicam.cs.pa.strategy* definisce la strategia, proposta da Knuth, che dovrà applicare il giocatore PC per trovare un'ipotetica soluzione per decodificare la riga.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe KnuthStrategy.java ha al suo interno:

Attributi:

- private int CODE_LENGHT

  Identifica il numero di "piroli" che posso inserire in una Row (riga).

- private int NUM_COLORS

  Identifica il numero di colori disponibili.

- private int NUM_OF_POSSIBLE_GUESSES

  Identifica il numero di possibili combinazioni.

- private int[] key

  Identificala riga da decodificare.

- private int[] guess

  Identifica il codice ottenuto come risultato dell'algoritmo.

- private int[2] score

  Identifica il risultato del codice tentato ([0] —> colore giusto e posizione giusta, [1] —> colore giusto ma posizione sbagliata).

- private int numOfGuesses

  Identifica il numero di tentativi fatti dall'algoritmo.

- private int[] [] potAns

  Identifica il contenitore di tutte le possibili combinazioni.

- private HashSet<Integer> usedGuess

  Identifica l'insieme delle ipotesi utilizzate.

- private HashSet<Integer> candidate

  Identifica l'insieme delle possibili soluzioni.

Metodi:

- private Row encriptyKey(int[] guess, GameDifficulty difficult)

  Questo metodo permette di convertire i dati generati dall'algoritmo di Knuth in una Row.

- private void continueKnuthAlgo()

  Questo metodo permette di continuare l'esecuzione dell'algoritmo di knuth.

- private void initKnuthAlgo()

  Questo metodo permette di inizializzare l'algoritmo di Knuth.

- private int[] decriptKey(Row Key, GameDifficulty difficult)

  Questo metodo permette di convertire una Row passata come parametro formale in dati utilizzabili dall'algoritmo di Knuth.

- private void generateS()

Questo metodo permette di generare tutte le possibili combinazioni per la risoluzione del gioco.

- private void condense(int[] inputGuess, int[] comparedResult)

Questo metodo permette di eliminare tutte le soluzioni non corrette dalla lista delle possibili soluzioni.

- private int[] nextGuess()

Questo metodo permette di generare un'ipotesi in base a quello che è stato già eliminato dall'insieme delle possibili soluzioni.

- private int nextCondenseCounter(int[] potGuess)

Questo metodo permette di ottenere il minimo numero di ipotesi.

- private int[] compareGuess(int[] correct,int[] guess)

Questo metodo permette di confrontare l'ipotesi generata dall'algoritmo con la riga da decodificare.

Ulteriori metodi presenti in questa classe sono i metodi definiti dall'interfaccia *StrategyInterface;* la loro descrizione è stata trattata precedentemente.

RandomStrategy.java

La classe RandomStrategy definita all'interno del package *it.unicam.cs.pa.strategy* definisce la strategia Random per la risoluzione della riga da decodificare (i colori che compongono l'ipotetica soluzione sono scelti in maniera random).

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe RandomStrategy.java ha al suo interno:

Metodi:

All'interno di questa classe è presente soltanto un metodo che permette la generazione dell'ipotetica soluzione in maniera totalmente casuale. Tale metodo è definito dall'interfaccia *StrategyInterface;* la sua descrizione è stata trattata precedentemente.

KnuthStrategyFactory.java

La classe KnuthStrategyFactory definita all'interno del package *it.unicam.cs.pa.startegy* permette di creare una strategia di tipo KnuthStrategy. Questa classe implementa l'interfaccia funzionale StrategyFactory.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe KnuthStrategyFactory.java ha al suo interno:

Metodi:

- public Player createStrategy()

Questo metodo permette di restituire una strategia di tipo KnuthStrategy da far adottare al giocatore PC.

RandomStrategyFactory.java

La classe RandomStrategyFactory definita all'interno del package *it.unicam.cs.pa.startegy* permette di creare una strategia di tipo RandomStrategy. Questa classe implementa l'interfaccia funzionale StrategyFactory.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe RandomStrategyFactory.java ha al suo interno:

Metodi:

- public Player createStrategy()

Questo metodo permette di restituire una strategia di tipo RandomStrategy da far adottare al giocatore PC.

StrategyArchive.java

La classe StrategyArchive definita all'interno del package *it.unicam.cs.pa.strategy* definisce un archivio che contiene tutti i tipi di strategie che possono essere create. Tali strategie sono "prese" dal file di testo "/strategyDataCreation/strategyData.txt".

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe StrategyArchive.java ha al suo interno:

Attributi:

- private static StrategyArchive instance

  Identifica l'istanza della classe.

- private Map<String,StrategyFactory> archive

  Identifica un contenitore che raccoglie tutte le strategie disponibili. Tale archivio di coppie di elementi sono contraddistinti da una chiave e dal valore ad essa associato. La chiave è ciò che identifica univocamente un elemento, quindi non è possibile avere due elementi nel nostro archivio con lo stesso valore di chiave.

- private List<String> nameTypeOfStrategy

  Identifica la lista contenente i nomi di tutte le strategie disponibili.

Metodi:

- private StrategyArchive()

  Questo è il costruttore della nostra classe il quale ci permette di far "partire" l'inizializzazione del nostro archivio.

- private void inizializate()

  Questo metodo permette di inizializzare l'archivio con i dati contenuti all'interno del path "/strategyDataCreation/strategyData.txt".

- public static StrategyArchive getInstance()

  Questo metodo restituisce l'istanza della mia classe. Nel caso fosse nulla, ne creo una nuova.

- public StrategyArchive getStrategy(String nameTypeOfStrategy)

  Questo metodo restituisce la strategia all'interno del nostro archivio a seconda del nome inserito come parametro formale.

- public void loadStrategy(File file)

  Questo metodo permette di caricare un file testuale, contenente le strategie, passato come parametro formale.

- private void register(String registryLine)

  Questo metodo permette di "registrare" una strategia. Tale metodo richiama a sua volta un altro metodo per la registrazione della strategia all'interno dell'archivio, infatti questo metodo si limita a prendere come parametro formale una stringa e di dividerla in due sottostringhe che verranno poi passate al successivo metodo per la registrazione. La stringa passata come parametro formale deve avere la seguente caratteristica: "nome_strategia nome_classe". Quindi tra il nome della strategia e il nome

della classe deve esserci solamente uno spazio vuoto che li divide. Nel caso in cui non sia rispettato il formato definito precedentemente e descritto all'interno del file "/standard/info.txt" viene lanciata un'eccezione e il gioco termina.

- private void register(String name, String className)

Questo metodo permette di "registrare" una strategia a seconda delle due stringhe passate come parametro formale. La prima identifica il nome della strategia , la seconda il nome della classe. All'interno di questo metodo abbiamo un'ulteriore chiamata a un altro metodo che effettuerà la vera e propria registrazione all'interno dell'archivio delle strategie.

- private void register(String name, StrategyFactory newInstance)

Questo metodo permette di aggiungere all'archivio delle strategie una nuova strategia. Inoltre, ci permette di aggiungere alla lista dei nomi delle strategie il nome della strategia aggiunta.

- public List<String> getNameTypeOfStrategy()

Questo metodo restituisce la lista che contiene i nomi delle strategie disponibili.

ViewInterface.java

L'interfaccia ViewInterface definita all'interno del package *it.unicam.cs.pa.view* permette di definire un contratto che indica le caratteristiche principali che devono essere rispettate da qualsiasi tipo di interfaccia che vogliamo implementare. ViewInterface estende l'interfaccia Observer che permette di definire che, qualsiasi oggetto ViewInterface è un oggetto osservatore, quindi che può osservare e può essere notificato di un determinato cambiamento avvenuto in un'istanza.

A seguire troveremo la spiegazione di ogni metodo.

L'interfaccia ViewInterface.java ha al suo interno:

Metodi:

- public ArrayList<String> getDataFromPlayer()

Questo metodo permette di restituire una lista di dati inseriti dall'utente. Tipicamente è utilizzata quando l'utente inserisce l'insieme dei "piroli". (ESEMPIO DI DATO INSERITO DALL'UTENTE: *R,V,G,B*)

- public int getDataFromConsole(int maxRange)

Questo metodo permette di restituire il numero intero inserito dall'utente. *MaxRange* definisce il massimo numero che l'utente potrà inserire (numeri da *0* a *maxRange*).

- public String getStringFromConsole()

Questo metodo permette di restituire una stringa inserita dall'utente. Tipicamente è utilizzata quando l'utente deve inserire il nome di un giocatore.

- public void print(String str)

Questo metodo permette di mostrare "a video" una stinga passata come parametro formale.

- public void info()

Questo metodo permette di mostrare "a video" tutte le informazioni iniziali atte per descrivere all'utente come si svolgerà i gioco.

- public void startGameMessage()

Questo metodo permette di mostrare "a video" il messaggio iniziale che permette l'inizio della partita da parte del Decodificatore.

- public void printTheSolution(Game game)

Questo metodo permette di mostrare "a video" quel è la combinazione che doveva essere indovinata. Tale metodo va richiamato solo alla fine della partita, sia se il vincitore è il *CODEMAKER* sia se il vincitore è il *CODEBREAKER*.

MainView.java

La classe MainView definita all'interno del package *it.unicam.cs.pa.view* definisce l'interfaccia grafica del tipo "CONSOLE". Questa classe implementa l'interfaccia ViewInterface che definisce le caratteristiche principali che ogni interfaccia grafica deve rispettare.

A seguire troveremo la spiegazione di ogni metodo e di ogni attributo.

La classe MainView.java ha al suo interno:

Attributi:

- private static final Scanner scan

Mi permette di ottenere tutto quello che l'utente inserisce sulla console.

Metodi:

- public void println(String str)

Questo metodo permette di rappresentare messaggi con l'aggiunta di '\n' finale.

Mastermind.java

La classe Mastermind definita all'interno del package *it.unicam.cs.pa.MASTERMIND* definisce il main del gioco.

La classe Mastermind.java ha al suo interno:

Metodi:

- public static void main(String[] args)

  Questo metodo permette far iniziare il gioco.

Estendibilità del codice

Il codice può essere esteso in maniera molto semplice e concisa.

Possiamo estendere e personalizzare i seguenti oggetti :

- **Giocatori**
- **Regole**
- **Strategie**
- **Interfaccia grafica**

Giocatori :

Possiamo aggiungere degli ulteriori giocatori e personalizzare le loro funzionalità nella seguente maniera:

La classe del nostro nuovo giocatore deve implementare l'interfaccia *Player* che definisce il contratto che tutti i giocatori devono rispettare. Successivamente bisogna creare una classe "factory" in grado di poter ritornare una nuova istanza dell'oggetto del nuovo giocatore. Tale classe dovrà implementare l'interfaccia *PlayerFactory*.

Infine, bisognerà aggiungere il nuovo giocatore al file di testo che contiene tutti i giocatori disponibili. Il file di testo è contenuto nel path "/playerDataCreation/data.txt".

N.B.: I dati inseriti nel file di testo devono assolutamente rispettare lo standard definito in "standard/info.txt".

Strategie :

Possiamo aggiungere ulteriori strategie per il giocatore PC che dovrà decodificare il codice e personalizzare le funzionalità nella seguente maniera:

La classe della nostra nuova strategia deve implementare l'interfaccia *StrategyInterface*, che definisce il contratto che tutte le strategie devono rispettare. Successivamente bisogna creare una classe "factory" in grado di poter restituire una nuova istanza dell'oggetto della nuova strategia. Tale classe dovrà implementare l'interfaccia *StrategyFactory*.

Infine, bisognerà aggiungere la nuova strategia al file di testo che contiene tutte le strategie disponibili. Il file di testo è contenuto nel path "/strategyDataCreation/strategyData.txt".

N.B.: I dati inseriti nel file di testo devono assolutamente rispettare lo standard definito in "standard/info.txt".

Interfaccia Grafica :

Possiamo aggiungere ulteriori interfacce grafiche e personalizzare le loro funzionalità nella seguente maniera:

La classe della nostra nuova interfaccia grafica deve implementare l'interfaccia *ViewInterface* che definisce il contratto che tutte le interfacce devono rispettare.

Regole :

Possiamo modificare semplicemente le regole del gioco, come ad esempio il numero di tentativi, i colori disponibili ecc. nella seguente maniera:

All'interno del file di testo "/rulesData/" possiamo inserire quanti file vogliamo contenenti le regole di gioco. All'interno del file creato possiamo inserire più modalità di gioco, come ad esempio *FACILE, DIFFICILE, HARD, ecc*. Tutte le modalità che aggiungiamo devono assolutamente rispettare lo standard definito in "standard/info.txt".