Davide Morelli
1849054
Presenza canale M-Z

SAPIENZA
UNIVERSITÀ DI ROMA

Progetto di Metodologie di programmazione in Java

Juno

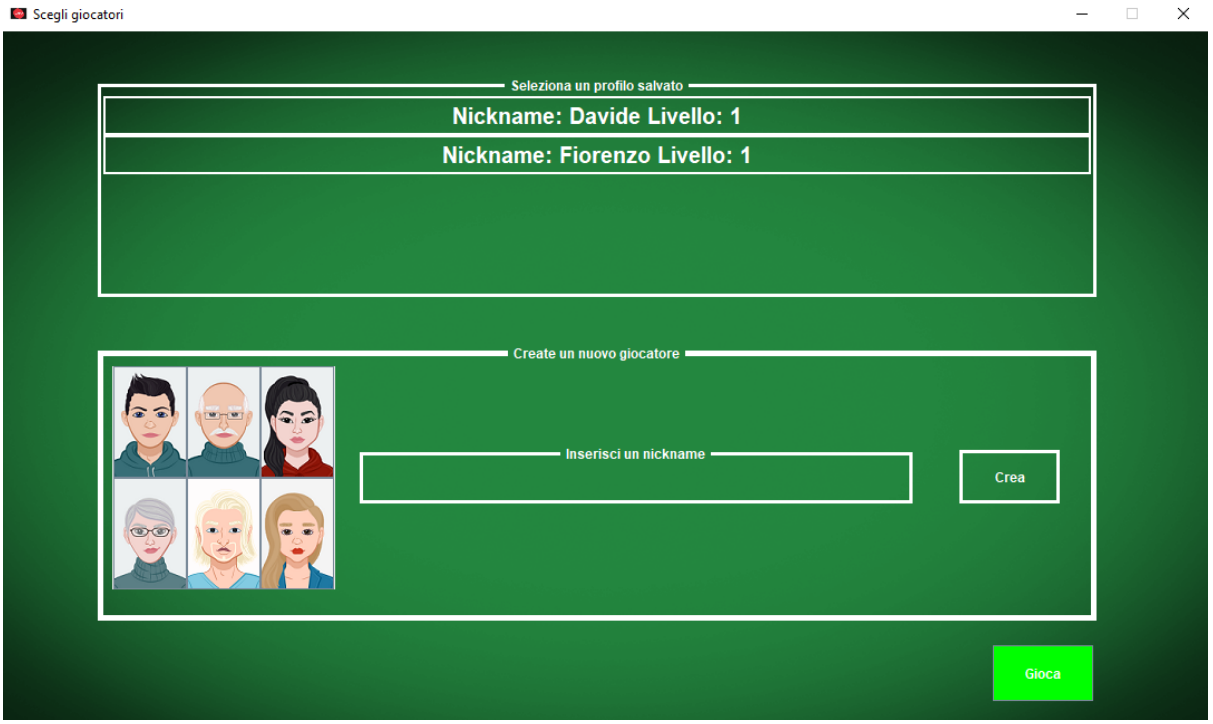
Decisioni di progettazione relative alle specifiche

Gestione profilo utente, nickname, avatar, partite giocate, vinte/perse, livello

Utente

Tutto ciò che riguarda la gestione del profilo utente si trova nel package *Model.GestioneUtente.*
Nel progetto ho cercato di porre particolare attenzione sull'estendibilità del codice, per questo motivo ho modellato l'utente con una classe *User* astratta. Questa scelta mira a dare la possibilità di creare altri possibili utenti, oltre il *Real player* e *Bot player*, più specifici per nuove modalità a cui si vuole estendere il codice.
La classe astratta "obbliga" le sue sottoclassi a creare un utente a partire da un nickname con cui identificare il giocatore, un path sottoforma di String con cui associare un avatar e un oggetto della classe *Statistics* che contiene le statistiche del giocatore.
Nella classe astratta *User* non ho esplicitato un costruttore e ho lasciato alle sottoclassi il compito di crearne uno in base alle loro esigenze, infatti nella sottoclasse *RealPlayer* ho scelto di costruire l'oggetto a partire da un nickname e un path per l'avatar, in quanto ritengo giusto far scegliere a chi sta giocando il nome e l'avatar da associare al suo profilo; allo stesso tempo nella sottoclasse *BotPlayer* ho un diverso costruttore, esso infatti non costruisce la figura del giocatore artificiale facendo scegliere un nickname e un avatar, ma sceglie in maniera casuale un nome e un avatar da una lista prestabilita.

Scelta giocatori



La creazione del giocatore avviene mediante questa schermata. L'utente può scegliere di utilizzare un vecchio profilo scegliendolo dalla lista nella parte alta della schermata oppure può crearlo da zero, scegliendo un avatar e immettendo un nickname. Una volta creato il nuovo giocatore, lo si potrà scegliere nella parte alta della schermata.
L'aggiornamento della lista dei profili salvati avviene mediante il pattern Observable-Observer.

Statistiche giocatori

Le statistiche dei giocatori sono modellate mediante una classe *Statistics*. L'oggetto di questa classe viene creato ogni qualvolta si crea un giocatore reale e non un giocatore artificiale, in quanto ho ritenuto opportuno non salvare le statistiche dei bot.
Le statistiche comprendono:

- **LIVELLO:** ogni nuovo giocatore parte dal livello 1. Il livello viene incrementato automaticamente mediante il metodo *checkStatistics(),* nel momento in cui il giocatore avrà maturato una determinata esperienza.
- **ESPERIENZA:** ogni nuovo giocatore parte con esperienza 0. L'esperienza viene incrementata ogni volta che il giocatore vince una partita, ed è quella componente che incrementa il livello del giocatore.
- **PARTITE VINTE O PERSE:** Le partite vengono aggiornate automaticamente nel profilo del giocatore al termine di un game. Sono

gli unici campi che possono essere incrementati all'esterno della classe, infatti i loro metodi *set* sono public; ho fatto questa scelta in quanto ho ritenuto importante non dare la possibilità all'esterno di incrementare i dati più "sensibili" come il livello e l'esperienza, dati che possono cambiare sensibilmente il profilo di un giocatore, e rendere il loro incremento automatico e possibile solo all'interno della classe.
All'interno di questa classe non è presente un campo "PARTITE GIOCATE" poiché ridondante essendo un'informazione che si può ricavare dalla somma delle partite vinte e perse.

Salvataggio profilo

Il salvataggio delle statistiche e più in generale del profilo di un giocatore è affidato alla classe *SaveLoadData*; è una classe singleton in quanto credo non sia opportuno creare più istanze di questo oggetto e può essere usato anche in nuove partite poiché il suo utilizzo non influisce in alcun modo sul comportamento della partita.
Questa classe salva un *RealPlayer* ogni qualvolta viene creato, mediante il metodo *SaveNewUtente().*
I dati vengono salvati su un file.txt, che si trova nella cartella FileData.
La classe salva tutti i dati all'interno di una mappa. La mappa contiene come chiave il nickname dell'utente e come valore l'istanza dell'utente. Ho deciso di salvare l'istanza del giocatore e non delle stringhe che rappresentassero le statistiche del giocatore perchè ritengo più agevole leggere dal file l'istanza del giocatore e aggiornare le statistiche accedendo direttamente all'oggetto *Statistics* con i metodi get e set della classe *Statistics.*
Il salvataggio dell'istanza del giocatore ha comportato l'utilizzo dell'interfaccia *Serializable,* per questo sia la classe *User* e sia la classe *Statistics* implementano *Serializable.*
N.B L'unico campo che non viene salvato all'interno del file.txt è l'istanza della classe SaveLoadData, per questo nella classe *User* il campo che contiene il riferimento all'istanza di questa classe oltre all'indicatore di visibilità è caratterizzato anche dal termine "transient".

In questa schermata è possibile visualizzare i profili salvati con le relative statistiche, aggiornate grazie al pattern Observable-Observer al termine della partita.

Gestione di una partita completa in modalità classica con un giocatore umano contro 3 giocatori artificiali

Implementazione del deck, carte

Tutte le classi che riguardano la modellazione del deck e delle carte si trova nel package *Model.GestioneDeck.*
Tutte le carte sono state modellate a partire dalle enum così da rendere il codice più estendibile possibile, infatti:

- Enum Color: contiene i colori classici delle carte e consente di aggiungere degli ulteriori colori qualora si volessero creare deck per altre modalità;
- Enum Action: contiene tutte le carte azione, quindi *reverse, skip, drawTwo;*
- Enum Value: contiene i simboli delle carte valore, quindi i numeri che vanno da 1 a 9.
- Enum Wild: contiene i simboli più comunemente conosciuti come “Jolly”, quindi pesca 4 o cambio colore.

Ogni enum ha un costruttore privato che associa ad ogni carta il valore in punti, nel caso in cui si volesse estendere il gioco alla modalità a punti.

Le carte vengono create tutte a partire dalla superclasse *Card.*
La suddetta classe contiene i campi riguardanti le enum, con i relativi metodi get e set, inoltre contiene un metodo di utilità con visibilità public usato per settare il colore della carta (questo metodo è stato creato per settare il colore delle carte jolly, “cambia colore” o “pesca 4”, carte che inizialmente non hanno un colore prestabilito).
Alle sottoclassi è lasciata l’implementazione dei metodi *toString()*, *getTypeCard()* che ritorna il tipo di carta e *gerCardPoints()*, tutti metodi che devono essere implementati in maniera specifica per ogni sottoclasse.
Nella classe card non è esplicitato un costruttore, in quanto la creazione della carta cambia a seconda del tipo: le carte valore vengono create con il colore e il valore, le carte wild sono create con il loro simbolo ed infine le carte azione che vengono create mediante la loro azione e il colore. La scelta di modellare le carte come sottoclassi fa sì che qualora volessi creare nuovi tipi di carte per altre modalità di gioco io possa estendere la classe *card* ed accettare il suo “contratto”.

Nota: Potenzialmente avrei potuto usare il *Builder pattern* per creare le istanze delle carte ma non l’ho ritenuto opportuno in quanto le carte non hanno costruttori particolarmente complessi o tante e diverse componenti da associare.

Deck e mazzo degli scarti

Anche per il deck vale lo stesso principio che ho utilizzato nella creazione delle carte, cioè consentire la realizzazione di diversi deck a seconda della modalità di gioco. Ho implementato una classe deck astratta che stipula il “contratto” a cui le sottoclassi devono sottostare. A differenza di quanto fatto finora, nella classe astratta ho esplicitato un costruttore che inizializza il tipo di deck che viene creato, infatti ogni sottoclasse dovrà richiamare il super costruttore ed esplicitare di che tipo è il deck creato (tipo contenuto nella enum DeckType). Il deck è una semplice ArrayList di <Card>, gli unici metodi che tutte le sottoclassi hanno in comune sono quelli per:

- mischiare il deck;
- estrarre la carta in cima al mazzo;
- rimuovere la carta in cima al mazzo.

L’unico deck creato è il deck classico. Esso è popolato mediante diversi metodi privati ed ha un numero fissato di carte.

Il mazzo degli scarti è stato implementato in modo similare al deck, con l'unica differenza che essendo una componente priva di varianti non è stata resa estendibile in alcun modo.

Gestione partita

La gestione della partita è affidata al package *Model.GameStage.*
La classe astratta *GameStage* fornisce le componenti essenziali alle sottoclassi verso la creazione di una partita. Anche la scelta di creare sottoclassi per ogni partita mira a consentire l'ideazione di diverse modalità di gioco pur mantenendo il "contratto" della superclasse.

La classe *GameStage* è composta da due riferimenti:

- Deck
- Discard cards

Contiene inoltre una ArrayList di User, denominata players, un int che conserva il turno del giocatore corrente, una boolean che determina la direzione del giro (orario o antiorario) e una TreeMap contenente nickname-Lista di carte nella mano del giocatore.
La suddetta classe astratta implementa un costruttore che deve necessariamente concernere l'istanza di un giocatore reale, un'istanza del tipo di deck che si vuole utilizzare a seconda della modalità di gioco, ed infine il numero di giocatori; nel costruttore viene istanziato il deck, creato il nuovo mazzo degli scarti, creata la nuova lista dei giocatori e la TreeMap contenente la mano di ogni giocatore, setta il giro verso sinistra (false) ed infine aggiunge alla lista dei giocatori i *BotPlayer* mancanti, creandoli runtime.
Una caratteristica essenziale ai fini della partita: il giocatore reale viene sempre aggiunto alla posizione 0 all'interno della lista degli User, così da sapere a priori a quale indice riferirsi quando si parla di *RealPlayer.*
La superclasse rende noti alle sottoclassi tutti i get e set dei campi ed alcuni metodi comuni a tutte le possibili modalità di gioco (dunque alle sottoclassi) come:

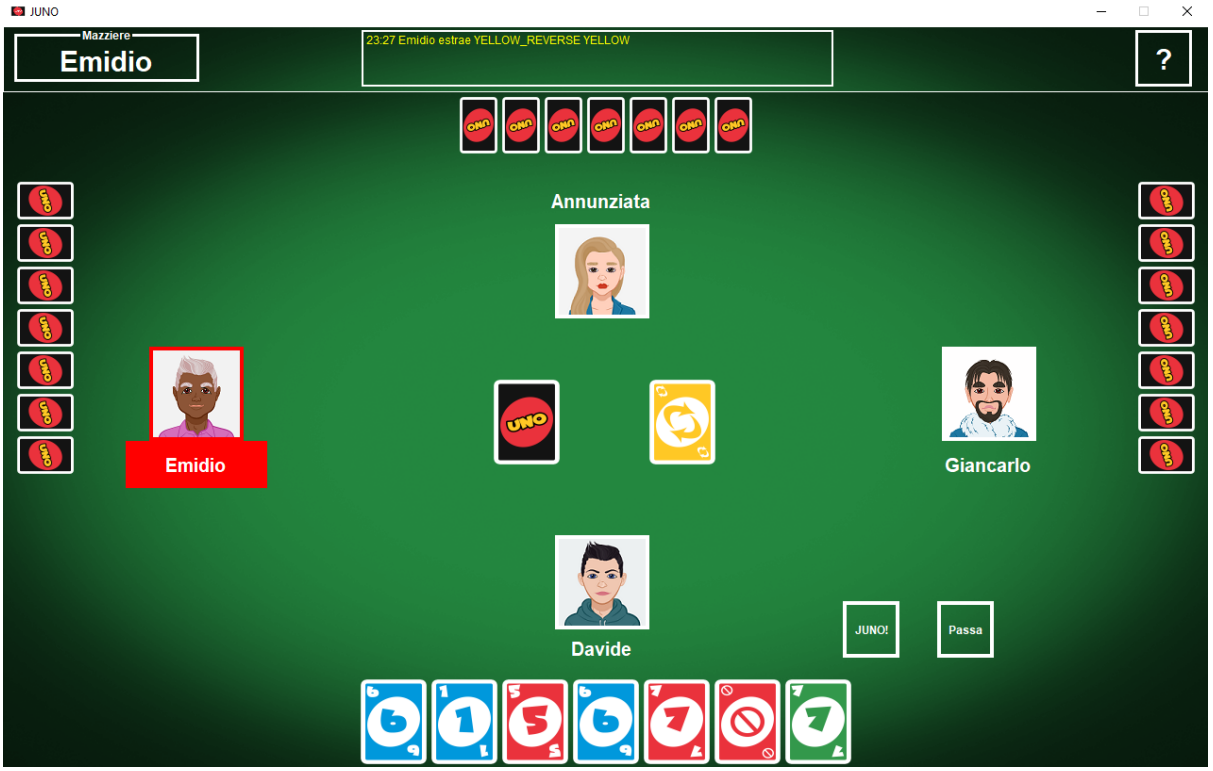
- setCurrentPlayer(): che setta il turno del prossimo giocatore;
- dealtCards(int numbersOfCards): che distribuisce le carte ad ogni giocatore mettendo in input il numero di carte da distribuire, qualora nelle diverse modalità si volessero distribuire un numero di carte >7;
- I metodi per aggiornare gli observer;
- Il metodo per pescare carte dal deck.

Tuttavia “obbliga” le sottoclassi ad implementare i seguenti metodi:

- metodo per stabilire il mazziere;
- metodo per stabilire la prima carta estratta;
- metodo riguardante gli effetti delle carte;
- metodo per far giocare il bot o il giocatore reale;
- metodo per controllare la possibile vittoria di un giocatore;
- metodo per controllare le possibili carte giocabili.

Questi metodi possono cambiare a seconda delle diverse modalità di gioco.

Schermata della partita



Al centro troviamo il mazzo degli scarti e il deck da cui pescare le carte (premendo su di esso).
In alto troviamo una barra contenente:

- un textarea che tiene traccia di tutte le mosse fatte durante la partita, le cui righe cambiano colore a seconda del colore in vigore in quel momento;
- un textfield contenente il nome del mazziere del game;
- un bottone che mostra le istruzioni per giocare.

Vicino le carte del real player ci sono due bottoni:

- Juno!: per gridare “Juno” quando si ha una sola carta in mano;
- Passa: per passare il turno in assenza di carte da giocare.

Il turno di ogni giocatore è evidenziato dal suo nickname di colore rosso, inoltre ogni turno è dettato da un timer di 5 secondi che rende il gioco più agevole da seguire.

Design pattern adottati

Nel progetto ho ritenuto opportuno utilizzare due design pattern:

- Observable-observer
- Singleton

Singleton

Ho utilizzato questo pattern nelle seguenti classi:

- Classe *SaveLoadData* nel package *Model.GestioneUtente.*
  Ho usato il pattern in questa classe poiché non servono ulteriori istanze per accedere al mio “database” rappresentato da un file.txt, ulteriori istanze occuperebbero solo spazio.
- ActionListener nel package Controller.ActionListener
  Ho creato 4 classi di action listener. Queste classi non hanno bisogno di ulteriori istanze in quanto il loro funzionamento rimane lo stesso in ogni istanza della modalità di gioco che ho implementato.
  Esempio pratico:
  Nel package vi è la classe *ActionListenerPlayerHand,* il listener che gestisce la mano del giocatore reale.
  Ogni carta è un bottone e ogni carta ha bisogno di un’istanza del listener per essere comandata. In una partita la mano di un player può contenere innumerevoli carte, pertanto non avrebbe senso creare altrettante innumerevoli istanze dello stesso listener, quando il funzionamento è il medesimo per ogni listener.

- Observable-Observer

Nel progetto ho deciso di utilizzare questo pattern per comunicare eventuali cambiamenti alla view.
Ho collegato tutti gli Observable ai propri observer nelle classi del package *Controller.Controllers* mentre gli observer sono stati implementati nel package *Controller.Observers.*
Nel progetto ci sono due classi che estendono la classe Observable:

Observable	Observer
SaveLoadData	*ObserverStatistics*
GameStage	*ObserverGameStage*

1. *SaveLoadData- ObserverStatistics*
   Questa collaborazione viene utilizzata per aggiornare la view in due casi:
   1. Quando viene creato un nuovo *RealPlayer* la lista dei giocatori creati nel *FramePlayers* si aggiorna all'istante rendendo possibile la scelta del giocatore con cui giocare il prossimo game. (Immagine pagina 4)
      Terminata la creazione del giocatore esso viene salvato nel file.txt, di conseguenza viene chiamato il metodo *updatePlayersDataOB(),* che manda alla view la mappa aggiornata dei players presenti nel file.txt ed aggiorna la parte alta del *FramePlayers.*

   2. Quando un game termina, le statistiche del giocatore reale che ha preso parte al game vengono aggiornate automaticamente, pertanto la view deve mostrare le statistiche aggiornate all'ultimo game nel *FrameStatistics*.
      Terminato il game e aggiornate le statistiche nel model, viene chiamato il metodo *updatePlayersDataOB()* (nella classe *saveLoadData*), esso manda le nuove informazioni alla view che aggiorna la propria schermata.

2. *GameStage - ObserverGameStage*

Questa collaborazione viene utilizzata per aggiornare la view in tre casi:

1. Ogni volta che viene giocata una carta da qualsiasi giocatore oppure vi è un evento di gioco che modifica il numero di carte in mano ai giocatori o le carte in cima al mazzo degli scarti, nella classe
   GameStageClassic viene chiamato il metodo *updateOb()*.
   Esso manda alla view ogni volta l'istanza aggiornata della classe GameStageClassic cosicchè la view possa estrarre da essa i dati necessari ad aggiornare il *FrameViewGS* nel migliore dei modi.

2. Aggiornare la barra superiore.
   Quando viene eseguita un'azione dai giocatori, il nuovo evento viene passato alla textarea che si trova nella barra superiore oppure all'inizio del game viene passata l'informazione relativa al dealer.

3. A ogni turno viene effettuato un check dal metodo *checkWin()* per controllare se un giocatore ha vinto il game, nel caso in cui l'esito sia affermativo, l'informazione viene passata alla view tramite il metodo *updateWinOb()* e di conseguenza viene creato un JOptionPane che mostra il vincitore del game.

Uso degli stream

All'interno del progetto ho principalmente utilizzato gli stream all'interno del metodo *checkHand()* nella classe *GameStageClassic* nel package *Model.GameStage.*

```java
/**
 * This method checks which cards the player can play according to the rules of the mode
 * @return playable cards
 */
@Override
public List<Card> checkHand() {
    Card lastDiscard= discards.getLastCard();
    Color color=lastDiscard.getColor();
    var card=lastDiscard.getTypeCard().equals("Action")? lastDiscard.getAction() :
        lastDiscard.getTypeCard().equals("Wild")? lastDiscard.getWild():lastDiscard.getValue();


    var carteGiocabili= this.getHands().get(getPlayers().get(currentPlayer).getNickname())
    .stream().filter(x-> x.getValue()==card || x.getAction()==card || x.getColor()==color || x.getTypeCard()=="Wild").collect(Collectors.toList());


    return carteGiocabili.stream().filter(x->x.getColor()==discards.getLastCard().getColor()).count()>0?
              carteGiocabili.stream().filter(x->x.getWild()!=Wild.WILDDRAWFOUR).collect(Collectors.toList()):carteGiocabili;
}
```

Questo metodo è il cardine di ogni mossa dei giocatori.
Controlla le carte giocabili dalla mano di ogni giocatore e restituisce la lista di carte giocabili relativa alla carta in cima al mazzo degli scarti.
Il metodo assume l'informazione relativa alla carta scartata precedentemente, su di essa filtra le carte nella mano del giocatore a seconda del valore o dell'azione o del colore o del tipo di carta ed infine restituisce mediante *.collect(Collectors.toList())* la lista delle carte giocabili.
Prima di restituire la lista, viene fatto un ulteriore check, infatti la carta "Pesca 4" può essere giocata se e solo se in mano non si hanno carte dello stesso colore della carta scartata, per cui viene filtrata nuovamente qualora non ci fossero le condizioni adeguate a giocare la carta wild.

Altre note progettuali e di sviluppo

Alcune delucidazioni relative alla view e al controller.

View

Per realizzare la view ho scritto interamente il codice in Java Swing, senza utilizzare alcun tool.
Ho due package relativi alla view:

- *View.Home*: contiene il frame relativo alla home, alla scelta del giocatore, alle statistiche dei giocatori e alle istruzioni
- *View.GameStage*: contiene tutte le informazioni relative alla creazione e all'aggiornamento della schermata partita.

Principalmente ogni frame contiene una JLabel, quest'ultima imposta il background del frame e può contenere altri componenti come ad esempio altre JLabel, buttons oppure JTextArea.

In tutti i frame del package *View.Home* ho scelto di creare delle inner class per quanto riguarda la label del background, in quanto quest'ultima è utilizzata per quello specifico frame e non ha nessuna utilità al di fuori di esso.

Il frame che contiene tutte le componenti della partita e composto da:

- *MainLabelGS:* essa imposta il background ed a sua volta contiene altre JLabel ordinate mediante il BorderLayout.
    1. *LabelCenterGS:* è la parte centrale della schermata. Contiene il JButton relativo al deck, da cui pescare le carte e la JLabel relativa alle carte scartate.
    2. *LabelPlayerHand:* nella parte bassa dello schermo, ovvero la JLabel contenente la mano del *RealPlayer.*
    3. *LabelPlayerBot:* questa label rappresenta la mano del giocatore artificiale mostrando solo il retro delle carte.

- *LabelTopGS:* questa JLabel è posta nella parte superiore del *FrameViewGS.*
  Contiene una JTextArea che mostra ogni evento relativo al game sottoforma di stringa:
  orario - nickname - carta giocata - colore carta
  Il colore del testo di quest'area cambia a seconda del colore in vigore sino a quel momento.
  Questa label contiene un bottone per le istruzioni del gioco e una label che contiene informazioni relative al mazziere.

All'interno di questo package vi è un'interfaccia. Essa implementa due metodi statici utilizzati per ridimensionare le icone all'interno di JButton o JLabel a seconda delle dimensioni di queste componenti.

Controller

Il controller è caratterizzato da tre package:

1. *Controller.ActionListener:* esso contiene tutti gli *actionPerformed* relativi a:
    - *Home*
    - *LabelCenter* ovvero i bottoni “Deck”, “Juno!”, “Passa”
    - *PlayerHand* ovvero i bottoni relativi alle carte in mano al giocatore reale
    - *Players* tutti i bottoni relativi alla creazione e alla scelta del personaggio con cui giocare il game.

Come spiegato precedentemente, i listener utilizzano il pattern singleton.

2. *Controller.Controllers:* esso contiene:
    - *ControllerGameStageClassic*, questo controller crea l’istanza del GameStageClassic, degli observer (creando i vari collegamenti con gli observable) ed infine tutti i listener utilizzati in questa fase. Il controller istanzia inoltre il *FrameViewGS* a cui passa tutti i dati (per iniziare la partita) provenienti dal model mediante l’observer.

    - *ControllerPlayers / ControllerStatistics*, svolgono lo stesso identico compito svolto dal *ControllerGameStageClassic.* Questi controllers creano le istanze dei relativi frame, degli observer (creando i vari collegamenti con gli observable) e infine tutti i listener utilizzati in questa fase.
    - La classe *Juno* contenente il *main* per avviare il progetto.

3. *Controller.Observers:* esso contiene i due observer, spiegati precedentemente, che sono rispettivamente:
    - *ObserverDataPlayers*
    - *ObserverGameStage*

Come giocare

Facendo il run del main contenuto nella classe Juno si apre la Home.
Nella home ci sono tre scelte:

1. “Gioca” con cui si inizia il game
   Si apre una schermata con cui il giocatore può scegliere di giocare con uno dei profili salvati o crearne uno nuovo.
2. “Statistiche” in cui sono contenute tutte le statistiche dei profili salvati.
3. “Come giocare” spiega le regole del gioco e come iniziare una partita.

Fase di gioco:
Valgono tutte le regole classiche di Uno.
Il giocatore avrà dinanzi a sé le sue carte da giocare, il deck da cui pescare, il mazzo degli scarti e due bottoni con cui passare il turno o gridare “Juno!”.
Il giocatore non può:

- pescare carte fuori dal suo turno;
- utilizzare i bottoni “passa” e “Juno!” fuori dal suo turno;
- giocare le sue carte fuori dal suo turno;
- giocare carte errate.

Tutto ciò è controllato da un sistema che fa apparire a schermo degli avvisi.
Finita la partita, apparirà a schermo un avviso con il nome del vincitore e verranno aggiornate automaticamente le statistiche del player.
Il giocatore può giocare una nuova partita tornando alla Home.