

Progetto di gruppo

Programmazione Avanzata Java e C

Stefano Valloncini

Yuhang Ye

Luigi Amarante

Per il corso del professor

Massimiliano Redolfi

Università degli Studi di Brescia

Ultimo aggiornamento: 6 luglio 2022

Indice

1	Introduzione	1
1.1	Il gioco di carte	1
1.2	Il mazzo di carte	1
1.3	Successione di gioco	3
1.4	Semplificazioni	4
1.5	Logo del gioco	4
1.6	Modalità di gioco	5
1.6.1	Gioco multiplayer offline	6
1.6.2	Gioco multiplayer LAN	6
1.6.3	Gioco in singplayer offline	9
2	Pattern Model-View-Controller	10
2.1	Funzionamento generale	10
2.2	Classi rappresentative	11
2.3	Immagine rappresentativa pattern MVC implementato nel progetto	12
3	View	13
3.1	CardView	13
3.1.1	Metodo <code>fillBackground</code>	14
3.1.2	Metodo <code>drawWhiteOvalInCenter</code>	14
3.1.3	Metodo <code>drawValueInCenter</code>	14
3.1.4	Metodo <code>drawValueInCorner</code>	15
3.1.5	Metodo <code>showHoverEffect</code>	15
3.1.6	Metodo <code>removeHoverEffect</code>	16
3.1.7	Metodo <code>paintComponent</code>	16
3.1.8	L'antialiasing	17
3.2	CardBackView	18
3.3	TableView	21
3.4	Componenti	21
3.4.1	Pannello informazioni giocatore	21
3.4.2	Pannello chat	23
3.5	Pannello di gioco	25
4	Model	27
4.1	GameModel	28
4.1.1	Attributi	28
4.2	Metodi	28
4.2.1	Gestore mosse	28
4.2.2	Controllo giocatore con una sola carta	30

4.2.3	Controllo se il gioco è terminato	31
4.2.4	Controllo chi è il vittore	31
4.2.5	Gestore dei turni	32
4.2.6	Metodo <code>getNextIndex</code>	32
4.3	Controller	33
4.4	Rungame	34
5	Thread e Executor Service	36
5.1	Thread	36
5.2	Executor Service	36
5.3	Thread utilizzati	37
5.4	Sincronizzazione eventi di gioco	38
5.5	Implementazione molteplici client	38
6	Architettura Client/Server	39
6.1	Client	39
6.1.1	Attributi	39
6.2	Server	40
6.2.1	Attributi	40
6.2.2	Attributi	41
6.2.3	Gestore client	41
7	Intelligenza artificiale	46
7.1	Metodo <code>determineNextMossa</code>	46
8	Conclusioni finali	49

1 Introduzione

Questo documento è la relazione del progetto realizzato per il corso di *Programmazione avanzata Java e C* dell’Università degli Studi di Brescia. L’obiettivo del progetto è stato realizzare in Java il gioco *Uno*, rispettando tutti i requisiti richiesti dal docente del corso.

Prerequisiti del progetto

- essere realizzato in Java
- avere un’architettura Client/Server applicativo
- avere un’interfaccia grafica basata su *Java Swing* o *JavaFX* e rispettare il pattern *MVC*
- utilizzare i thread (ad esempio per la comunicazione oppure per la gestione di calcoli complessi/simulazioni)

In modo particolare, abbiamo usufruito per la maggior parte dell’interfaccia grafica, del framework **Swing**, che verrà descritta approfonditamente successivamente nella sezione 3.

1.1 Il gioco di carte

Iniziamo con il vedere le regole generale del gioco. A tutti i giocatori vengono consegnate, casualmente, 8 carte. **Il gioco si svolge a turni.**

In un turno un giocatore è libero di pescare o posizionare sul campo di gioco una carta, la quale deve essere ovviamente valida, cioè rispettare le regole del gioco. Una carta non valida non può essere posizionata sul campo di gioco. Il giocatore che per primo riesce a esaurire tutte le carte che ha nella sua mano vince la partita. L’altro giocatore, perde.

Quando un giocatore raggiunge un numero di carte in suo possesso pari a uno, deve attivare il pulsante **UNO!**. In caso contrario, se non effettua questa operazione, gli verranno assegnate due carte, e quindi, gli viene impedito di vincere la partita in quel turno (nel caso in cui il giocatore in questione avesse scelto di posizionare la sua ultima carta sul campo da gioco).

Non si prevede di continuare il gioco una volta che uno dei giocatori ha completato il gioco: esiste solo un vincitore, e tutti gli altri giocatori sono perdenti.

1.2 Il mazzo di carte

Un mazzo di carte di *Uno* è composto da 108 carte: carte numeriche, carte *action-card*, carte *wild-card*. Tutte le carte, a parte le carte *wild-card*, sono caratterizzate da un particolare colore (e un tipo, che caratterizza il tipo di carta che rappresentano). Le carte *wild-card* non hanno colori, ma una volta posizionate richiedono che sia selezionato un nuovo colore a propria scelta. Nel caso una carta

wild-card venga selezionata come prima carta di gioco, allora essa verrà considerata come speciale e potrà essere posizionato un qualunque colore sul terreno di gioco. L'intero mazzo di carte è presentato nella figura 1¹

- 19 carte di colore Blu, numerate dall'1 al 9 (2 serie) più uno 0
- 19 carte di colore Rosso, numerate dall'1 al 9 (2 serie) più uno 0
- 19 carte di colore Verde, numerate dall'1 al 9 (2 serie) più uno 0
- 19 carte di colore Giallo, numerate dall'1 al 9 (2 serie) più uno 0

Inoltre, come carte *speciali*:

- 8 carte **Pesca Due** dei quattro colori sopracitati (tipo action-card)
- 8 carte **Cambio giro** dei quattro colori sopracitati (tipo action-card)
- 4 carte **Cambio colore** (tipo wild-card)
- 4 carte **Pesca Quattro e Cambio colore** (tipo wild-card)

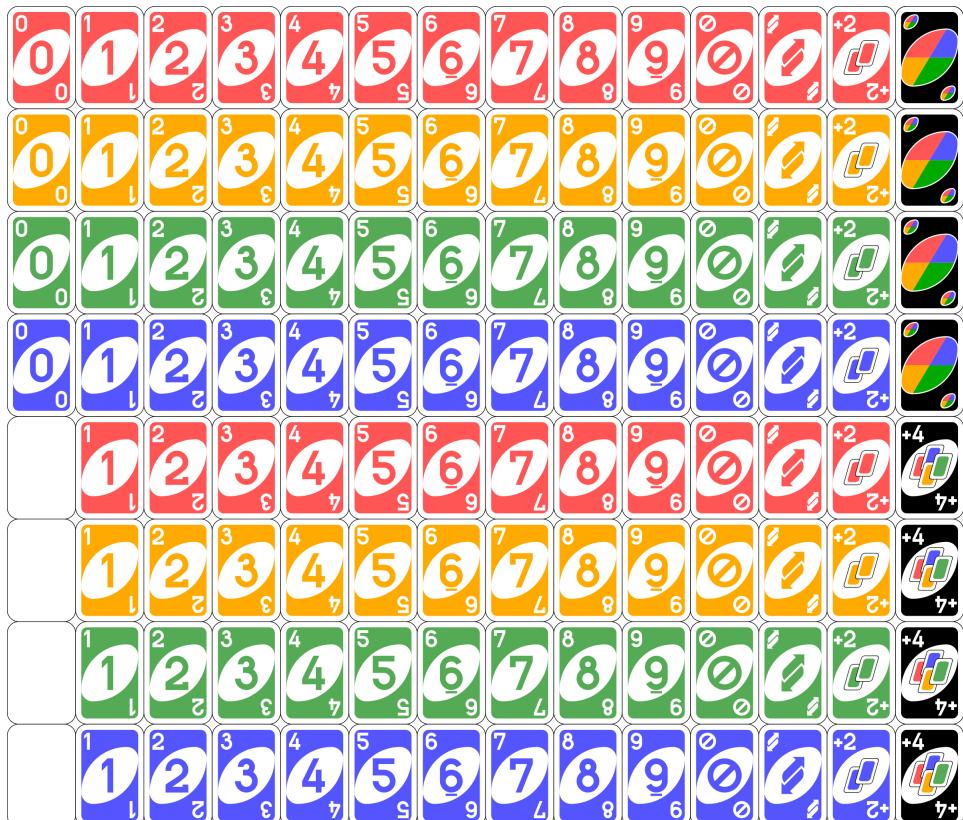


Figura 1: L'intero set di carte di uno

¹Fonente: Pagina Wikipedia uno

1.3 Successione di gioco

Il funzionamento del gioco è lineare, le uniche carte che alterano lo scorrere del gioco sono le carte di **stop** e **inverti**. Il gioco può essere riassunto nei seguenti passi:

- Inizio del gioco
- Turno giocatore
- $\uparrow\downarrow$ finché il gioco finisce
- Prossimo turno
- Fine del gioco

Inoltre gli effetti delle carte speciali sono i seguenti:

- Una carta **Pesca quattro** obbliga l'altro giocatore a pescare quattro carte e a decidere un nuovo colore di carta che può essere posizionato sul terreno di gioco;
- Una carta **Pesca due** obbliga l'altro giocatore a pescare due carte;
- Una carta **Cambia colore** permette al giocatore di cambiare colore.

Inizio gioco

Dopo aver consegnato le carte come descritto nella sezione precedente, inizia un giocatore. L'estrazione del giocatore che inizia per primo è fatta in modo casuale. Viene inoltre estratta dal mazzo una carta in modo casuale, che sarà considerata come la prima carta posizionata sul tavolo. Le carte che vengono date sono otto, a ciascun giocatore. Le carte del mazzo sono esattamente quelle descritte nella sezione 1.2.

Turno giocatore

Il giocatore deve decidere in base alle sue carte se pescare una nuova carta dal mazzo, oppure se utilizzare una carta in base a quella presente sul tavolo. Si è limitato il numero di carte a 30 contemporaneamente in mano ad un giocatore, perché la probabilità che non sia disponibile una carta valida tra le 30, è statisticamente impossibile. Questo impedisce ad un giocatore di giocare solo di mosse banali, cioè pescando solo carte. Inoltre alcune carte hanno come effetto il far pescare più carte ad un giocatore. In tal caso, l'altro giocatore deve pescarle e non ha modo di evitare questo.

Prossimo turno

Una volta effettuata una qualunque delle due scelte ammissibili, cioè o pescare una carta o posizionarne una sul tavolo da gioco, il turno passa al giocatore successivo (in senso orario, a sia cambiato da una carta). Unica eccezione è fatta nel momento in cui viene utilizzata dal giocatore una carta "cambio giro" o "stop", in uno di questi due casi il turno viene passato *saltando* il turno successivo per il giocatore prossimo o *invertendo* (se il senso corrente è orario, allora diventerà antiorario, se invece

è antiorario, diventerà orario). i turni successivi, come descritto successivamente.

Fine del gioco

Il gioco prosegue finché un giocatore non termina le sue carte nella sua mano. Quando un giocatore arriva ad avere solo una carta nella sua mano, allora deve mobilitarsi per indicare di avere solo una carta a tutti gli altri giocatori. Giocando in persona questo si ottiene esclamando "uno", nel gioco da noi realizzato in Java occorre premere il pulsante "Uno!". Il giocatore che con una sola carta compie una qualunque operazione (pescare una carta o selezionarne una), è penalizzato con l'aggiunta di due carte alla sua mano. Un giocatore che indica correttamente "uno", e seleziona la sua ultima carta è vincitore.

1.4 Semplificazioni

Come già descritto in precedenza abbiamo costruito la grafica del gioco in modo che supportasse due giocatori. Questa è una limitazione, in quanto nel gioco reale è possibile giocare da 2 a 10 giocatori. Tuttavia, abbiamo di fatto creato tutte le classi e gli strumenti necessari affinché fosse possibile di fatto creare una partita con più giocatori. Di fatto però, è possibile giocare soltanto in due persone, in quanto a livello visivo non sono supportati più giocatori. A livello tecnico tutti i concetti tecnici verranno descritti nelle sezioni successive.

1.5 Logo del gioco

Il logo ufficiale del gioco²:



Figura 2: Logo ufficiale del gioco

²Fonte: Pagina Wikipedia uno

1.6 Modalità di gioco

Abbiamo creato una serie di modalità di gioco, in modo da creare diversità nei temi di sviluppo:

- **Multiplayer LAN:** attraverso l'utilizzo di specifiche classi (che verranno descritte in seguito), abbiamo permesso a due utenti di collegarsi e giocare una partita su due diversi computer, collegati sulla stessa rete locale.
- **Multiplayer locale:** abbiamo permesso di far giocare due giocatori sullo stesso computer, che non dispongono di due diversi computer o di una connessione LAN. Entrambi i due giocatori vedono le carte dell'altro. Questa è una sorta di limitazione, ma che può portare ad essere una modalità di gioco divertente.
- **Singleplayer:** abbiamo implementato una funzione di singleplayer (ovviamente in locale) che permette di giocare contro il computer, o di simulare una partita gestita interamente dal computer. Abbiamo creato dunque una *sottospecie* di intelligenza artificiale molto semplice e che dato lo scenario di gioco fosse in grado di decidere la scelta migliore (o una delle migliori) per il giocatore.

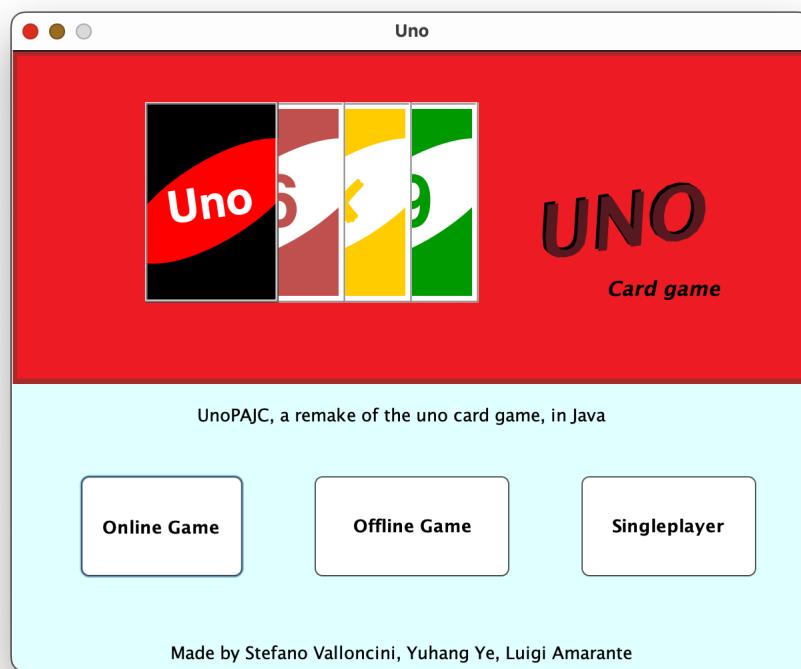


Figura 3: Schermata principale del gioco da cui selezionare la modalità di gioco

1.6.1 Gioco multiplayer offline

La prima modalità di gioco è il giocatore multiplo in locale. Dunque, i due giocatori giocano entrambi sulla stessa interfaccia grafica, e le carte dei due giocatori non sono nascoste. Questa funzione del gioco mima la versione del gioco uno in tutte le sue regole (descritte nella sezione 1), ma a carte scoperte.

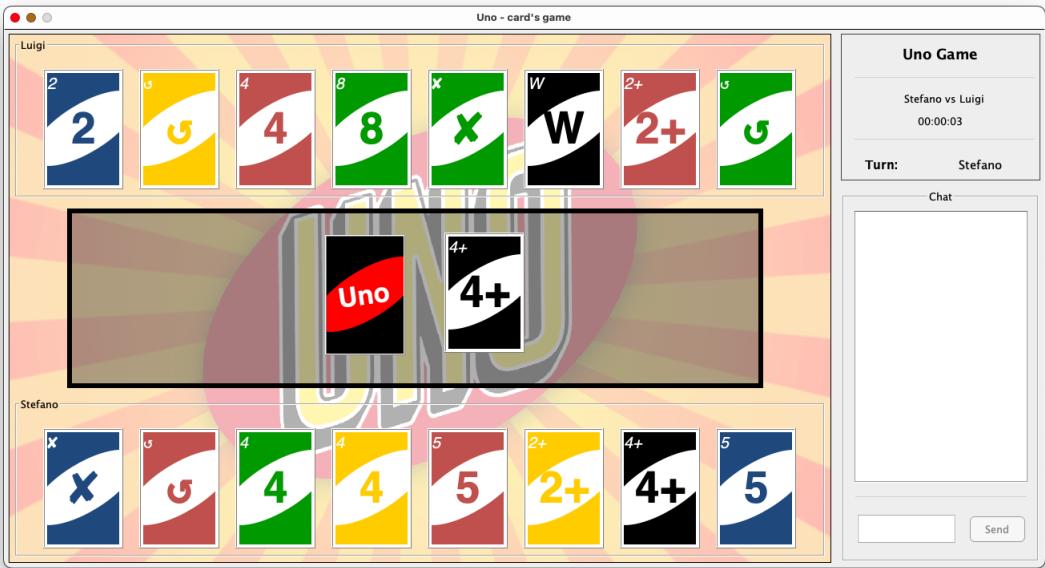


Figura 4: Il gioco in modalità offline: entrambi i giocatori hanno le carte scoperte

Ovviamente i giocatori potranno compiere mosse soltanto durante il loro turno. La chat di gioco viene disabilitata, in quanto la comunicazione non è necessaria tra i due giocatori, dato che la partita è in locale.

1.6.2 Gioco multiplayer LAN

La seconda modalità di gioco è di giocatore multiplo su rete locale. Dunque è necessario che due utenti abbiano due computer collegati alla stessa rete LAN per giocare una partita insieme.

Le carte del giocatore avversario sono coperte, ed è possibile vedere solo il retro delle carte dell'avversario, proprio come se fosse una partita tra due giocatori nella vita reale. Le regole del gioco sono esattamente quelle riportate nella sezione 1.

Non sono presenti limitazioni particolari, ed è possibile scrivere messaggi nella chat di gioco, e vedere di chi è il turno di gioco. È possibile giocare in due giocatori in multiplayer.

Nella figura 5 si vede lo screenshot a gioco appena iniziato per il client, mentre per il server nella figura 6. Se i due giocatori entrambi posizionano due carta valide, allora i risultati saranno: 7 e 8.

Finestra di inizio gioco:

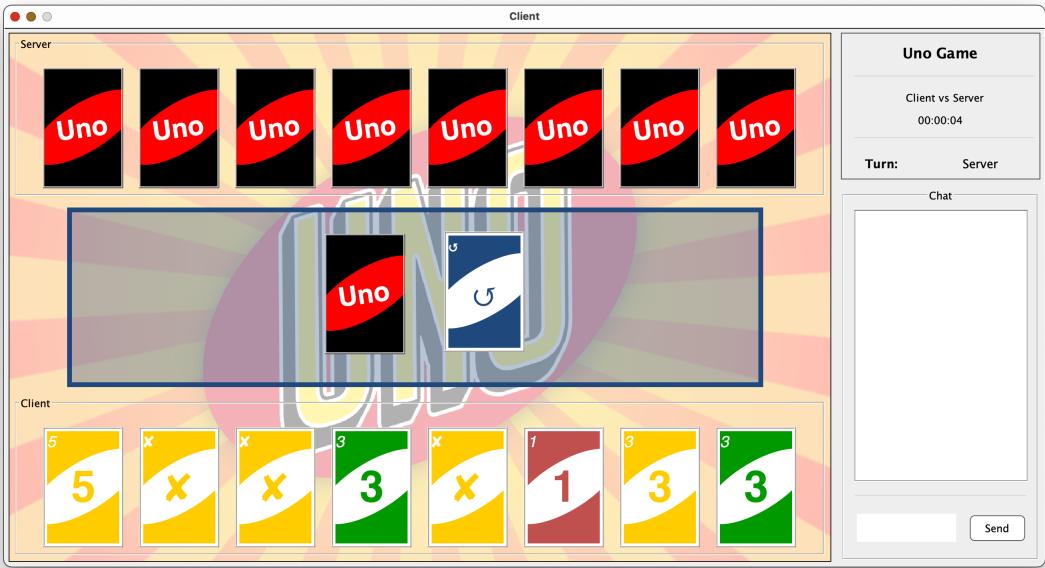


Figura 5: Finestra di gioco per il client

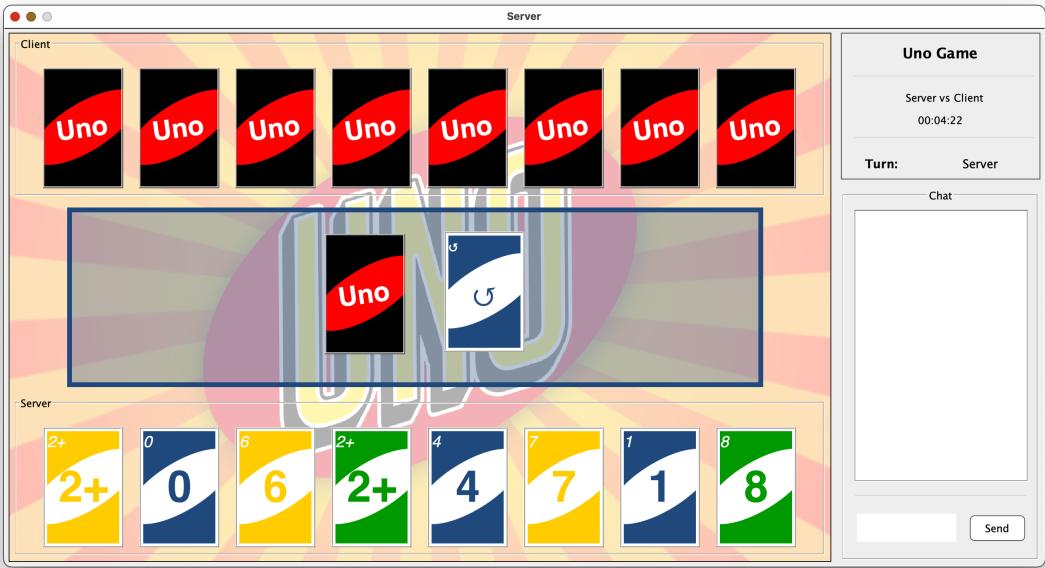


Figura 6: Finestra di gioco per il server

Le finestre sono uguali, ma ovviamente la carte mostrate nei due diversi host sono diverse. L'aggiornamento della grafica e di tutti i suoi componenti avviene ogni volta che avviene una mossa compiuta da un giocatore.

Dopo un’ipotetica prima mossa, dove entrambi i giocatori piazzano una carta:

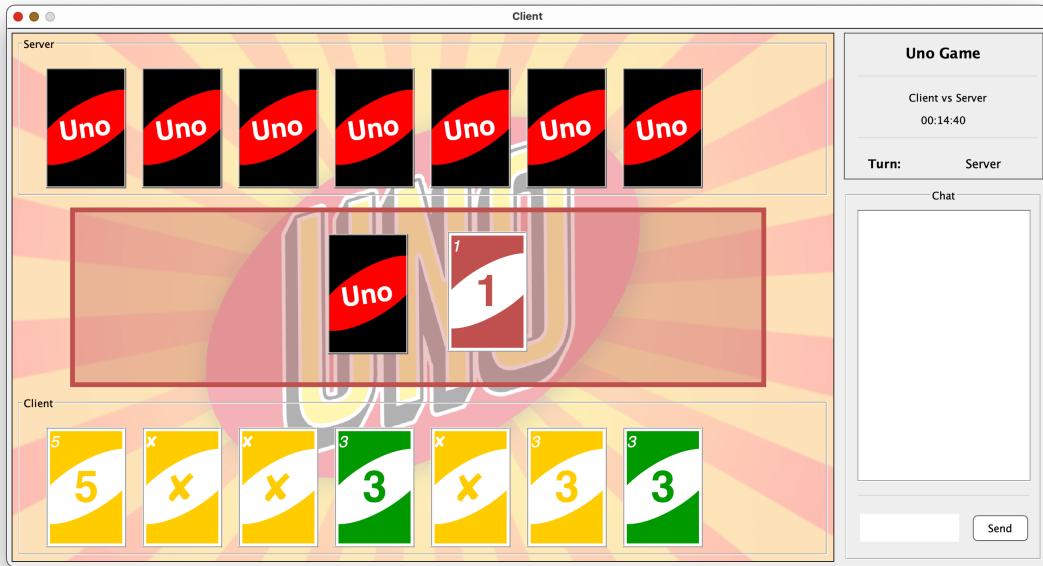


Figura 7: Finestra di gioco per il client

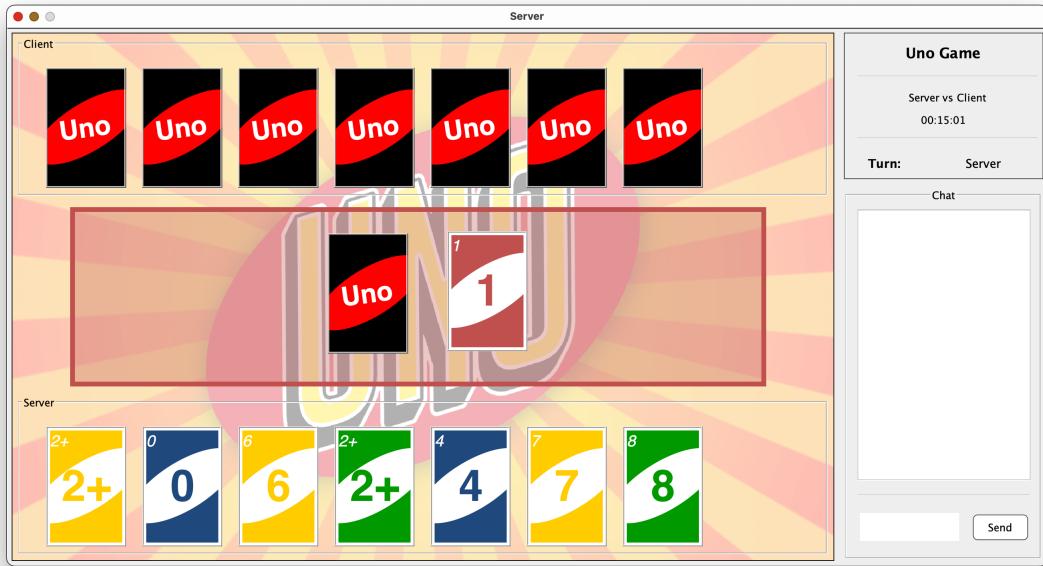


Figura 8: Finestra di gioco per il server

Il gioco prosegue, seguendo i turni come dalle regole, e quando termina viene mostrato un messaggio ad entrambi i giocatori. Ovviamente il messaggio sarà diverso a seconda dell’esito che ha avuto il giocatore nella partita.

1.6.3 Gioco in singplayer offline

L'ultima modalità che abbiamo implementato, è stata quella in singleplayer offline. Abbiamo creato questa modalità creando un'intelligenza artificiale che fosse in grado di scegliere al posto di un giocatore reale. Le regole del gioco sono le stesse riportate nella sezione 1. Le carte del giocatore avversario (robot) non sono visibili.

Esiste anche la possibilità di simulare una partita, facendo giocare entrambi i giocatori al computer. In questo modo, l'utente visualizzatore potrà vedere la partita, senza di fatto intervenire in nessun modo nella partita.

Nella sezione 7 verrà descritto il comportamento dell'intelligenza artificiale.

2 Pattern Model-View-Controller

Abbiamo utilizzato il pattern architettonale MVC, suddividendo il programma logicamente in view, model e controller.

- il **model** fornisce i metodi per accedere ai dati utili all'applicazione. Nel nostro programma il model rappresenta i vari metodi e attributi che servono per la gestione del gioco e delle varie mosse;
- la **view** visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti. Nel nostro caso, la view è il tavolo di gioco;
- il **controller** riceve i comandi dell'utente (attraverso la view) e li attua modificando lo stato degli altri due componenti. Nel nostro caso è il gestore della partita.

Inoltre, la comunicazione tra model, view e controller, è nel seguente modo:

1. Model e view non comunicano mai tra di loro;
2. Controller e view non comunicano sempre, ma solo attraverso dei messaggi o eventi;
3. Il model è invece sempre disponibile per il controller.

Il Controller dunque definisce un target su se stesso che potrà essere invocato dalla View al verificarsi di determinati eventi.

Nel progetto abbiamo rispettato il pattern, tuttavia, il framework Swing, che abbiamo utilizzato per la realizzazione grafica, non presenta una chiara separazione tra controller e view. Di fatto delle parti di view sono presenti nel controller, mentre la comunicazione avviene come effettivamente richiesto. La suddivisione tra controller e model invece è marcata e segue esattamente il pattern MVC. Ovviamente, **model e view non comunicano in nessun modo**.

2.1 Funzionamento generale

Ogni volta che l'user che sta utilizzando il gioco, compie un'azione (che ricordiamo, può essere soltanto pescare la carta oppure sceglierne una dal proprio mazzo per eventualmente posizionarla sul campo da gioco se valida), la view lancia un evento, ed, il controller, in ascolto per possibili eventi, recepisce che è avvenuta un'azione, e si occupa di compiere gli step successivi.

Esempio effettivo di funzionamento: nella view viene lanciato un evento: il giocatore ha scelto di pescare una carta. A questo punto il controller, in ascolto di possibili eventi, grazie all'evento lanciato, riconosce che è avvenuta questa azione, e si occupa di controllare (appunto *controller*) che l'azione sia valida. Interroga quindi il model, verificando che il giocatore abbia un numero di carte minore di 30 (limitazione spiegata nella 1), e nel caso il model restituisca un qualche valore che consente l'azione, allora l'azione viene correttamente effettuata, altrimenti si avvisa il giocatore che l'azione è *illegale*.

2.2 Classi rappresentative

Abbiamo suddiviso in maniera chiara model, view e controller.

Model

Il model è sostanzialmente costituito da una sola classe `GameModel`. Essa contiene tutte le funzioni logiche del gioco.

Controller

Abbiamo creato un controller principale, il quale si occupa di gestire i diversi tipi di giocatori che si possono avere, tuttavia abbiamo diversi tipi di giocatore

- `PlayerController` (classe astratta)
- `LocalPlayerController` (eredita da `PlayerController`)
- `AIPlayerController` (eredita da `PlayerController`)

E per quanto riguarda il gioco in multiplayer locale:

- `NetClient` fa da controller per il Client
- `NetServer` fa da controller per il Server

View

La view è composta da una serie di classi da noi create, che permettono di renderizzare a schermo i vari componenti grafici. In modo particolare abbiamo:

- Visualizzazione carta
 - `CardView`
 - `BackCardView`
 - `CardPlacedView`
- Visualizzazione tavolo da gioco
 - `TableView`
- Eventi (principali)
 - `CardDrawnEvent`
 - `CardSelectedEvent`

Tutti queste classi saranno descritte approfonditamente nelle sezioni successive.

2.3 Immagine rappresentativa pattern MVC implementato nel progetto

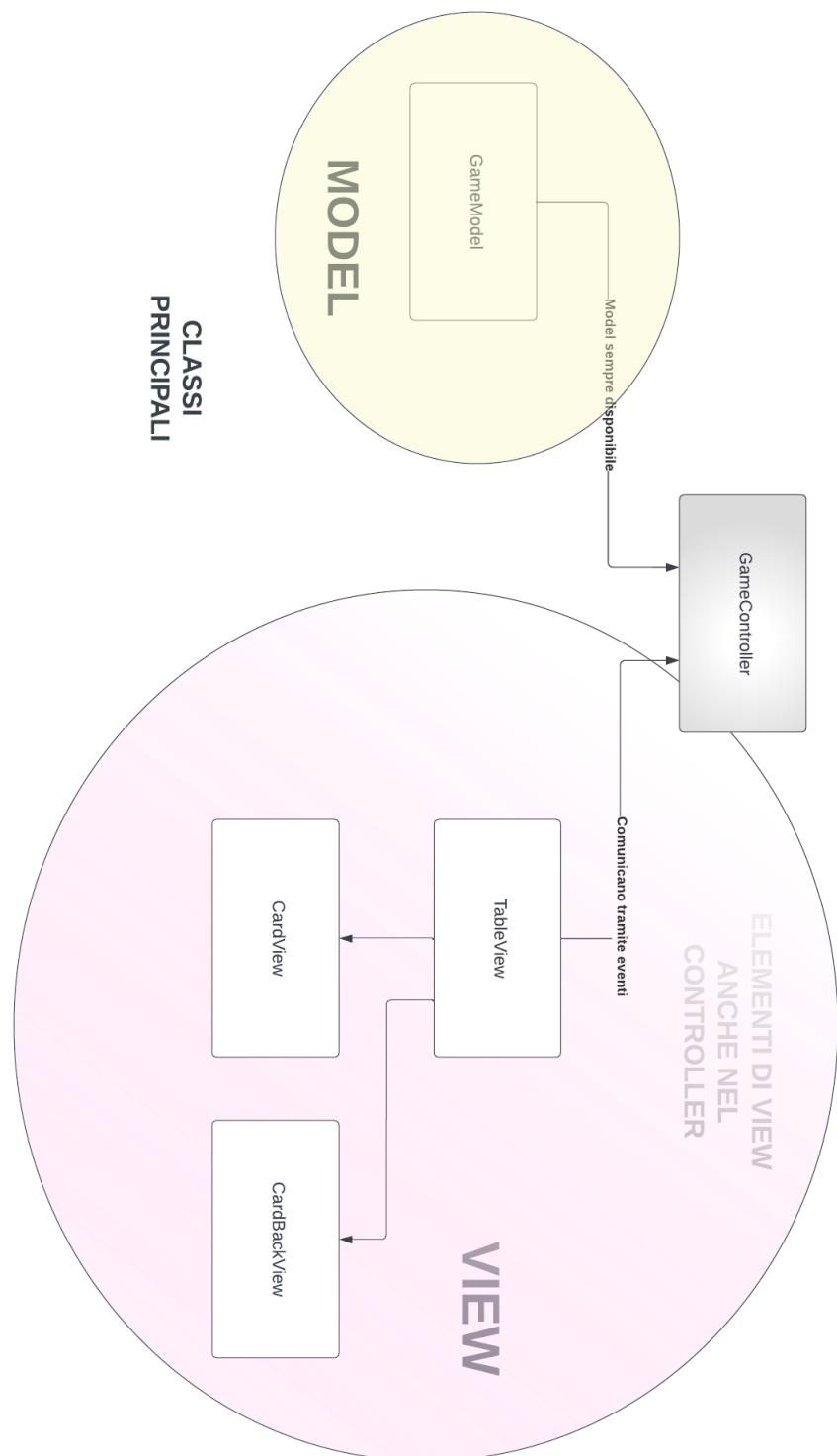


Figura 9: Il model è sempre disponibile al controller, mentre view e controller dialogano attraverso eventi o messaggi

3 View

La view del gioco è rappresentata dalle seguenti classi:

```
1 public class TableView
```

Rappresenta il tavolo da gioco, la chat di gioco e il tabellone dei turni;

```
1 public class CardView
```

Rappresenta la view di una singola carta;

```
1 public class BackCardView
```

Che rappresenta il retro di una singola carta.

3.1 CardView

La classe CardView si occupa di rappresentare graficamente un oggetto di tipo carta, che sia coerente con gli attributi della carta che si deve rappresentare. Segue un esempio di carta numerica, nella figura 11.



Figura 10: A sinistra delle carte numeriche, a sinistra delle carte speciali

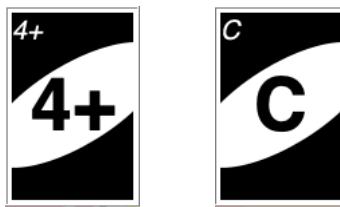


Figura 11: Le carte *wild-card*, che permettono di cambiare colore quando selezionate. In modo particolare la +4 permette di selezionare un nuovo colore e fa pescare al prossimo giocatore 4 carte da gioco.

Occupiamoci di definire il tipo di oggetto CardView. Innanzitutto, useremo come Container della carta un JPanel. Il JPanel sarà così suddiviso:

- In alto a sinistra, presenta il simbolo della carta in questione.
- Al centro, presenta un’ovale, ruotato di un certo grado rispetto alla carta.
- All’interno della carta è presente il simbolo rappresentante la carta
- Lo sfondo della carta è di colore uguale a quello della carta che va rappresentata.

3.1.1 Metodo fillBackground

```
1  private void fillBackground(Graphics2D g2, Color cardColor)
2  {
3      g2.setColor(Color.WHITE);
4      g2.fillRect(0, 0, cardWidth, cardHeight);
5
6      g2.setColor(cardColor);
7      g2.fillRect(margin, margin, cardWidth - 2 * margin, cardHeight - 2 * margin);
8  }
```

Il metodo `fillBackground` riempie lo sfondo della carta. Il funzionamento è molto semplice, inizialmente colora lo sfondo di bianco, e successivamente colora la carta del suo colore effettivo, lasciando ai margini -2 pixel di spazio, in modo da mostrare una sorta di bordo bianco (dalla prima volta che viene colorato il rettangolo, in quanto avevamo colorato lo sfondo di bianco).

3.1.2 Metodo drawWhiteOvalInCenter

```
1  private void drawWhiteOvalInCenter(Graphics2D g2)
2  {
3      var transformer = g2.getTransform();
4      g2.setColor(Color.white);
5      g2.rotate(45, (double) cardWidth * 3 / 4, (double) cardHeight * 3 / 4);
6      g2.fillOval(0, cardHeight / 4, cardWidth * 3 / 5, cardHeight);
7
8      g2.setTransform(transformer);
9  }
```

Il metodo disegna un ovale al centro della carta e lo ruota di 45°. In questo modo mima il disegno della carta del gioco originale. Per fare questo abbiamo utilizzato il metodo `fillOval` che colora un ovale delle dimensioni che abbiamo ripetuto opportune.

3.1.3 Metodo drawValueInCenter

```
1  private void drawValueInCenter(Graphics2D g2, Color cardColor)
2  {
3      var defaultFont = new Font(Util.DEFAULT_FONT, Font.BOLD, cardWidth / 2 + 5);
4      var fontMetrics = this.getFontMetrics(defaultFont);
5      int stringWidth = fontMetrics.stringWidth(value) / 2;
6      int fontHeight = defaultFont.getSize() / 3;
7
8      g2.setColor(cardColor);
```

```

9     g2.setFont(defaultFont);
10    g2.drawString(value, cardWidth / 2 - stringWidth, cardHeight / 2 + fontHeight);
11 }

```

Il metodo si occupa di prendere il simbolo della carta e di disegnarlo al centro della carta. Per fare questo utilizza la classe `Graphics2D`, la quale permette di usare il metodo `drawString`. Inoltre abbiamo utilizzato `setFont` per impostare il font che è stato creato in precedenza, e `setColor` per impostare un particolare colore del font.

3.1.4 Metodo `drawValueInCorner`

```

1  private void drawValueInCorner(Graphics2D g2)
2  {
3      var defaultFont = new Font(Util.DEFAULT_FONT, Font.ITALIC, cardWidth / 5);
4
5      g2.setColor(Color.white);
6      g2.setFont(defaultFont);
7      g2.drawString(value, margin, 5 * margin);
8  }

```

Il metodo si occupa di disegnare il valore della carta e metterlo in alto a sinistra della carta. Per fare questo, utilizza come anche negli altri metodi, la classe `Graphics2D`. In particolare utilizza il metodo `drawString` per effettivamente disegnare la stringa; `setFont` per impostare il font che è stato creato in precedenza, e `setColor` per impostare un particolare colore del font.

3.1.5 Metodo `showHoverEffect`

```

1  public void showHoverEffect()
2  {
3      if (active)
4      {
5          setBorder(focusedBorder);
6          Point p = getLocation();
7          p.y -= 20;
8          setLocation(p);
9      }
10 }

```

Quando l'utente interagisce con la carta, cioè quando passa sopra al risultato grafico della `CardView`, si vuole che essa sia posta in rilievo rispetto alle altre carte. La rilevazione di quando una carta è selezionata, non fa parte di questa classe, tuttavia è necessario avere un metodo che mostri dei cambiamenti quando la carta viene selezionata.

Questo deve avvenire soltanto quando la carta è contrassegnata come **attiva**, cioè quando effettivamente è selezionabile dall'utente. Per questo si introduce l'attributo nella **CardView** che permette di selezionare le carte si può muovere o meno.

Se la carta è selezionabile e viene posta in *hover*, allora viene impostato un bordo diverso, più spesso e scuro, in modo tale che risulti chiaro all'utente che sta per selezionare quella carta.

3.1.6 Metodo removeHoverEffect

```
1  public void removeHoverEffect()
2  {
3      if (active)
4      {
5          setBorder(defaultBorder);
6          Point p = getLocation();
7          p.y += 20;
8          setLocation(p);
9      }
10 }
```

Chiaramente, è necessario che l'effetto di *hover* venga rimosso una volta che è stato impostato a causa di una delle carte è posta in *hover*. Per questo è stato creato un metodo che rimuovesse gli effetti impostati dall'*over* e faccia ritornare la carta nella sua posizione di default.

3.1.7 Metodo paintComponent

A questo punto arriviamo al metodo fondamentale per la renderizzazione della carta, il metodo **paintComponent**.

Questo metodo è necessario per disegnare qualcosa su **JPanel**, oltre a disegnare il colore di sfondo. Questo metodo esiste già in una classe **JPanel**, per cui è necessario utilizzare la dichiarazione super per aggiungere qualcosa a questo metodo, che accetta oggetti **Graphics** come parametri.

Il metodo **super.paintComponent()**, che rappresenta il normale metodo **super.paintComponent()**, di **JPanel** che può gestire solo lo sfondo del pannello, deve essere richiamato nella prima riga.

```
1  @Override
2  protected void paintComponent(Graphics g)
3  {
4      super.paintComponent(g);
5
6      Graphics2D g2 = (Graphics2D) g;
```

```

7
8 // ANTIALIASING
9 g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
10   ↳ RenderingHints.VALUE_ANTIALIAS_ON);
11 g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
12   ↳ RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
13
14 var cardColor = Util.convertCardColor(card.getCardColor());
15
16 fillBackground(g2, cardColor);
17 drawWhiteOvalInCenter(g2);
18 drawValueInCenter(g2, cardColor);
19 drawValueInCorner(g2);
20 }
```

Vengono quindi richiamati tutti i metodi sopracitati, i quali disegnano *step-by-step* i singoli elementi grafici componenti le carte. Inoltre abbiamo impostato l'antialiasing come descritto nella sezione 3.1.8. Creando questo metodo, possiamo comodamente avere un solo metodo da invocare sull'oggetto CardView, in modo da inizializzare la grafica della carta.

3.1.8 L'antialiasing

Soffermiamoci su due righe di codice del metodo sopracitato, in modo particolare sulla riga di codice che sfrutta il metodo `setRenderingHint` della classe `java.awt.Graphics2D`:

```

1 g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
2   ↳ RenderingHints.VALUE_ANTIALIAS_ON);
3 g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
4   ↳ RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
```

L'antialiasing è stato fondamentale per rendere l'interfaccia grafica più piacevole alla vista. In modo particolare, dato che tutti gli elementi a schermo sono renderizzati dalla GPU, e non sono state impiegate fotografie (tranne quella di sfondo del gioco), abbiamo visto degli effetti di scalettatura, specialmente su alcuni schermi ad alta risoluzione³.

Questa riga di codice permette di utilizzare l'*antialiasing*. Dalla documentazione⁴:

```

1 public static final RenderingHints.Key KEY_ANTIALIASING
2
```

³L'antialiasing (delle volte abbreviato con AA) è una tecnica per ridurre l'effetto aliasing (in italiano, scalettatura, gradinatura o scalettamento) quando un segnale a bassa risoluzione viene mostrato ad alta risoluzione. L'antialiasing ammorbidisce le linee smussandone i bordi e migliorando l'immagine.

⁴<https://docs.oracle.com/javase/7/docs/api/java/awt/RenderingHints.html>

- 3 Antialiasing hint key. The ANTIALIASING hint controls whether or not the geometry
 ↳ rendering methods of a Graphics2D object will attempt to reduce aliasing
 ↳ artifacts along the edges of shapes.
- 4
- 5 A typical antialiasing algorithm works by blending the existing colors of the pixels
 ↳ along the boundary of a shape with the requested fill paint according to the
 ↳ estimated partial pixel coverage of the shape.

E inoltre, sempre dalla documentazione ⁵:

- 1 `public static final Object VALUE_ANTIALIAS_ON`
- 2
- 3 Antialiasing hint value -- rendering is done with antialiasing.

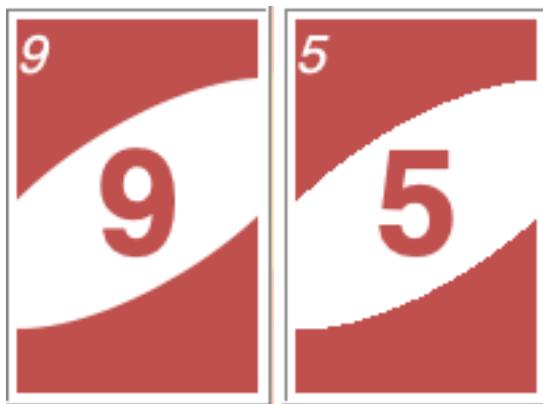


Figura 12: Una carta da gioco, numerica e di colore rosso. Si nota come la carta a destra abbia una scalettatura molto più evidente di quella a sinistra, che invece ha l'antialiasing applicato ad essa (l'effetto è poco visibile nel documento a causa delle compressioni delle foto e dello screenshot fatto. Su uno schermo ad alta risoluzione e con l'effettiva applicazione che abbiamo sviluppato, il risultato è più evidente).

3.2 CardBackView

Oltre alla visualizzazione della carta in sé, abbiamo ricreato il retro della carta in modo tale che potesse essere utilizzato in tutte le modalità di gioco a carte coperte. Le modifiche non sono state sostanziali rispetto alla grafica delle carte già descritte nella sezione precedente, in quanto le dimensioni della carta sono ovviamente le stesse, è sempre presente un ovale centrale ed è presente la scritta del logo *Uno*. Il codice rimane infatti pressoché invariato:

- 1 `/**`
 2 `*`

⁵<https://docs.oracle.com/javase/7/docs/api/java.awt/RenderingHints.html>

```

3   * @param g2
4   * @param cardColor
5   */
6 private void fillBackground(Graphics2D g2, Color cardColor)
7 {
8     g2.setColor(Color.BLACK);
9     g2.fillRect(0, 0, cardWidth, cardHeight);
10
11    g2.setColor(cardColor);
12    g2.fillRect(margin, margin, cardWidth - 2 * margin, cardHeight - 2 * margin);
13 }
14
15 /**
16 *
17 * @param g2
18 */
19 private void drawWhiteOvalInCenter(Graphics2D g2)
20 {
21     var transformer = g2.getTransform();
22     g2.setColor(Color.red);
23     g2.rotate(45, (double) cardWidth * 3 / 4, (double) cardHeight * 3 / 4);
24     g2.fillOval(0, cardHeight / 4, cardWidth * 3 / 5, cardHeight);
25
26     g2.setTransform(transformer);
27 }
28
29 private void drawValueInCenter(Graphics2D g2)
30 {
31     // var defaultFont = new Font(Util.DEFAULT_FONT, Font.BOLD, cardWidth / 2 + 5);
32     var defaultFont = new Font(Util.DEFAULT_FONT, Font.BOLD, cardWidth / 3);
33     var fontMetrics = this.getFontMetrics(defaultFont);
34     int stringWidth = fontMetrics.stringWidth("Uno") / 2;
35     int fontHeight = defaultFont.getSize() / 2;
36
37     g2.setColor(Color.white);
38
39     AffineTransform affineTransform = new AffineTransform();
40     affineTransform.rotate(Math.toRadians(-8), 0, 0);
41     Font rotatedFont = defaultFont.deriveFont(affineTransform);
42

```

```

43     g2.setFont(rotatedFont);
44     g2.drawString("Uno", cardWidth / 2 - stringWidth, cardHeight / 2 + fontHeight);
45 }

```

La parte effettivamente diversa del codice sta nel fatto che abbiamo dovuto ricreare una grafica simile all'originale, in modo tale che fosse simile al gioco originale, e per questo abbiamo creato uno sfondo colorato di nero, e riempito l'ovale centrale di rosso. Inoltre, abbiamo completato la grafica scrivendo, attraverso il metodo `drawString` la scritta *UNO*, in bianco.

Il testo è stato rotato di `Math.toRadians(-8)` radianti in modo tale che fosse inclinato per ricreare l'effetto tipico del gioco. Infine, abbiamo aggiunto una semplice label con scritto *card game*.



Figura 13: Rappresentazione della carta come appare effettivamente in gioco

3.3 TableView

La classe table view si occupa di mostrare a schermo la grafica dell'interfaccia del tavolo da gioco principale. La grafica si divide essenzialmente nei seguenti elementi, per cui per ciascuno abbiamo creato un pannello diverso:

- Tavolo da gioco (con annessa visualizzazione delle carte dei giocatori, e il tavolo centrale, con il mazzo di carte della carte usate e quelle da pescare);
- Chat di gioco, che è possibile utilizzare durante il gioco in multiplayer;
- Pannello delle informazioni, in modo tale che rimanga traccia del tempo di gioco e di chi è il turno in quel momento.

3.4 Componenti

Analizziamo la View del tavolo da gioco nelle sue componenti, partendo dalla versione finale che abbiamo realizzato.

3.4.1 Pannello informazioni giocatore

Abbiamo ideato una sorta di pannello che fosse in grado di descrivere il momento attuale del gioco, per fare questo abbiamo aggiunto al gioco una serie di elementi secondari, tra cui un timer che mostrasse a schermo il tempo passato dall'inizio del gioco.



Figura 14: Il pannello delle informazioni, dove vengono mostrati i nomi dei giocatori, il turno e il tempo corrente di gioco

Per realizzare questo in **Swing**, è stato sufficiente utilizzare delle **JLabel**, mentre è più interessante l'utilizzo del **Timer** di gioco, il quale è stato utilizzato utilizzando la classe **java.util.Timer**. Dalla documentazione⁶:

⁶<https://docs.oracle.com/javase/7/docs/api/java/util/Timer.html>

```

1 public class Timer
2 extends Object
3
4 A facility for threads to schedule tasks for future execution in a background
→ thread. Tasks may be scheduled for one-time execution, or for repeated execution
→ at regular intervals.
5
6 Corresponding to each Timer object is a single background thread that is used to
→ execute all of the timer's tasks, sequentially. Timer tasks should complete
→ quickly. If a timer task takes excessive time to complete, it "hogs" the timer's
→ task execution thread. This can, in turn, delay the execution of subsequent
→ tasks, which may "bunch up" and execute in rapid succession when (and if) the
→ offending task finally completes.
7
8 After the last live reference to a Timer object goes away and all outstanding tasks
→ have completed execution, the timer's task execution thread terminates
→ gracefully (and becomes subject to garbage collection). However, this can take
→ arbitrarily long to occur. By default, the task execution thread does not run as
→ a daemon thread, so it is capable of keeping an application from terminating. If
→ a caller wants to terminate a timer's task execution thread rapidly, the caller
→ should invoke the timer's cancel method.
9
10 If the timer's task execution thread terminates unexpectedly, for example, because
→ its stop method is invoked, any further attempt to schedule a task on the timer
→ will result in an IllegalStateException, as if the timer's cancel method had
→ been invoked.
11
12 This class is thread-safe: multiple threads can share a single Timer object without
→ the need for external synchronization.

```

Per ottenere quindi un cronometro funzionante, abbiamo usato un'etichetta `JLabel`, e abbiamo creato il `Timer` in modo che ogni 1000 millisecondi aggiornasse l'etichetta. Per questo abbiamo creato un metodo per l'inizializzazione del `Timer`:

```

1 public void initTimer()
2 {
3     Timer timer = new Timer(1000, new ActionListener()
4     {
5         int elapsedTime = 0;
6         int seconds = 0;
7         int minutes = 0;

```

```

8     int hours = 0;
9
10    String seconds_string = String.format("%02d", seconds);
11    String minutes_string = String.format("%02d", minutes);
12    String hours_string = String.format("%02d", hours);
13
14    public void actionPerformed(ActionEvent e)
15    {
16        elapsedTime = elapsedTime + 1000;
17        hours = (elapsedTime / 3600000);
18        minutes = (elapsedTime / 60000) % 60;
19        seconds = (elapsedTime / 1000) % 60;
20        seconds_string = String.format("%02d", seconds);
21        minutes_string = String.format("%02d", minutes);
22        hours_string = String.format("%02d", hours);
23        lblStopWatch.setText(hours_string + ":" + minutes_string + ":" +
24            seconds_string);
25    }
26
27    timer.setCoalesce(true);
28    timer.start();
29}

```

Il timer quindi aggiorna ogni 1000 millisecondi la label relativa al timer, con il tempo aggiornato. Per ottenere il tempo passato abbiamo creato una variabile `elapsedTime` a cui in ogni iterazione viene aggiunto 1000 (ms, cioè un secondo).

Per ottenere le ore basta dividere il tempo trascorso per 3600000, mentre per i minuti bisogna dividere per 60000 e fare il modulo con 60. Per i secondi basta dividere per 1000 e fare sempre il modulo con 60. Con il metodo `format` abbiamo formattato i numeri in modo che se fossero minori di 10, allora aggiungesse in automatico lo zero davanti al numero.

Essendo, dalla documentazione `java.util.Timer`, una classe *Thread-safe*, non abbiamo dovuto fare particolari operazioni di sincronizzazione.

3.4.2 Panello chat

La parte di gestione dei messaggi inviati e ricevuti non fa ovviamente parte della `View` in quanto essa si occupa solo della visualizzazione di tali messaggi.

La chat di gioco è composta da essenzialmente due parti:

- Pannello di contenimento degli altri elementi
- `JTextArea` area di testo per mostrare i messaggi mandati e ricevuti
- Box per l'inserimento del messaggio da mandare, con annesso bottone per l'invio



Figura 15: La chat di gioco, con alcuni messaggi inviati

La chat è ovviamente attiva soltanto nel caso in cui venga richiesta una partita in multiplayer, in tutte le altre modalità essa non è abilitata, in quanto non è necessario l'invio di messaggi in una partita locale.

```
1  public void addChatMessage(String message, String playerName)
2  {
3      LocalDateTime time = LocalDateTime.now();
4      DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("HH:mm");
5      String formattedDate = time.format(myFormatObj);
6
7      textAreaChat.append("\n" + formattedDate + " " + playerName + ": " + message);
8 }
```

Chiaramente bisogna aggiungere la data di arrivo del messaggio, in modo che rimanga una scaletta temporale dei messaggi inviati e ricevuti. Alla funzione bisogna passare il messaggio che è stato inviato (o ricevuto), e il nome del giocatore che manda (o riceve) il messaggio.

3.5 Pannello di gioco

Il pannello principale è quello della visualizzazione del gioco effettivo. Ovviamente abbiamo creato una sorta di tavolo nel quale sono posizionate le carte dei giocatori (indipendentemente dal fatto che siano coperte o scoperte, le carte dovranno essere mostrate negli appositi pannelli). Per questo motivo abbiamo suddiviso la grafica del pannello in:

- Pannelli per le carte dei giocatori;
- Pannello centrale per le carte da pescare e le carte posizionate, all'interno del quale ci sono:
 - Bottoni di uno da eventualmente premere quando il giocatore ha una sola carta (come descritto nelle regole di gioco);
 - Mazzo di carte da cui pescare;
 - Mazzo di carte già posizioante;
 - Pannello mostrante il colore delle carte che possono essere posizionate sul campo di gioco.

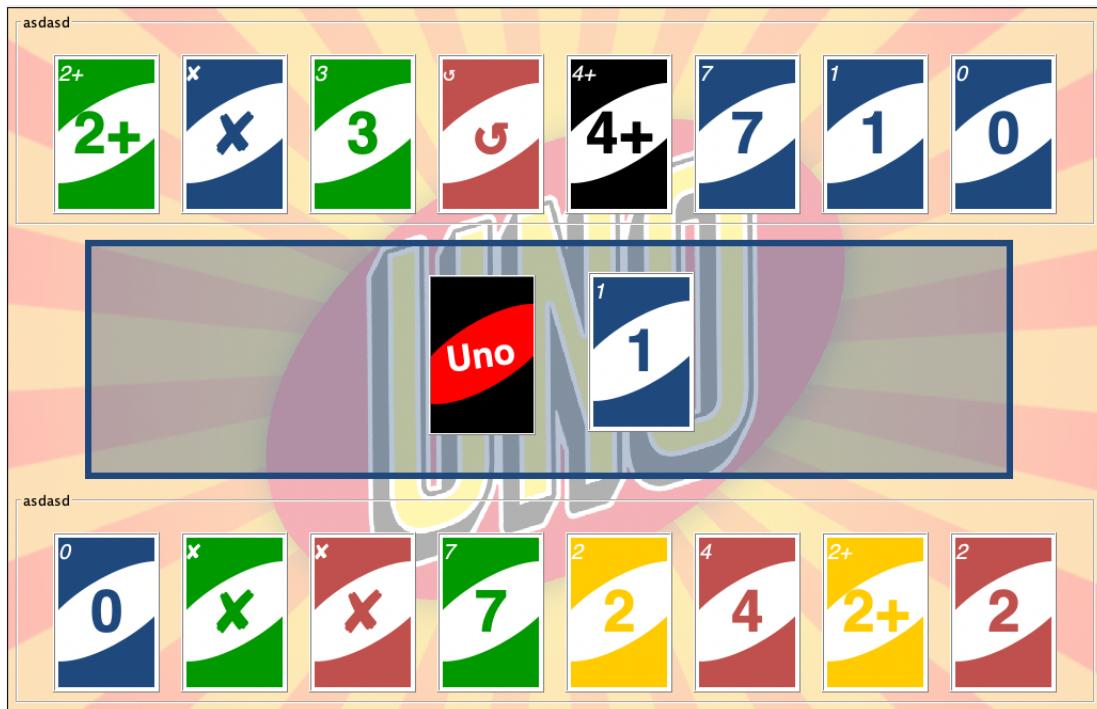


Figura 16: Tavolo principale di gioco

Per realizzare l'interfaccia di gioco, abbiamo sfruttato diversi pannelli. In modo particolare riconosciamo i seguenti elementi:

- Pannello avversario: composto da un `JPanel`, con all'interno un `JLayeredPane`.
- Pannello giocatore corrente: composto da un `JPanel`, con all'interno un `JLayeredPane`.

- Pannello centrale composto da un pannello JPanel con una percentuale di trasparenza con il colore e il bordi del colore della carta.

È interessante descrivere il funzionamento del **JLayeredPane**, un tipo di pannello i cui elementi possono essere sovrapposti tra di loro, e risulta essere particolarmente adatto per rappresentare le carte nella mano del giocatore. Dalla documentazione ⁷:

¹ This code uses the three-argument version of the add method. The third argument
 → specifies the Duke label position within its depth, which determines the
 → component's relationship with other components at the same depth. Positions are
 → specified with an int between -1 and (n - 1), where n is the number of
 → components at the depth. Unlike layer numbers, the smaller the position number,
 → the higher the component within its depth. Using -1 is the same as using n - 1;
 → it indicates the bottom-most position. Using 0 specifies that the component
 → should be in the top-most position within its depth. As the following figure
 → shows, with the exception of -1, a lower position number indicates a higher
 → position within a depth.

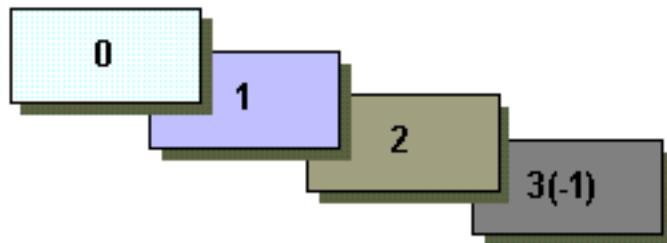


Figura 17: Immagine che descrive come il **JLayeredPane** dispone i pannelli

In termini semplici, un **JLayeredPane**, permette di aggiungere molteplici componenti a lui stesso e di e di poterne mettere uno in particolare in superficie. Noi siamo interessati a mettere in *superficie* il pannello (la carta) più a destra. La carta più a destra sarà quella più coperta invece.



Figura 18: Immagine del **JLayeredPane** effettivamente utilizzato

⁷<https://docs.oracle.com/javase/tutorial/uiswing/components/layeredpane.html>

4 Model

Il model contiene tutti i metodi contenenti la logica effettiva del gioco. La classe principale che si occupa di contenere il model è

```
1 public class GameModel
```

Per la gestione dei turni abbiamo introdotto la classe:

```
1 public class PlayerRoundIterator
```

Mentre, per quanto riguarda le carte di gioco, abbiamo creato una struttura gerarchica del seguente tipo:

```
1 public interface Card
```

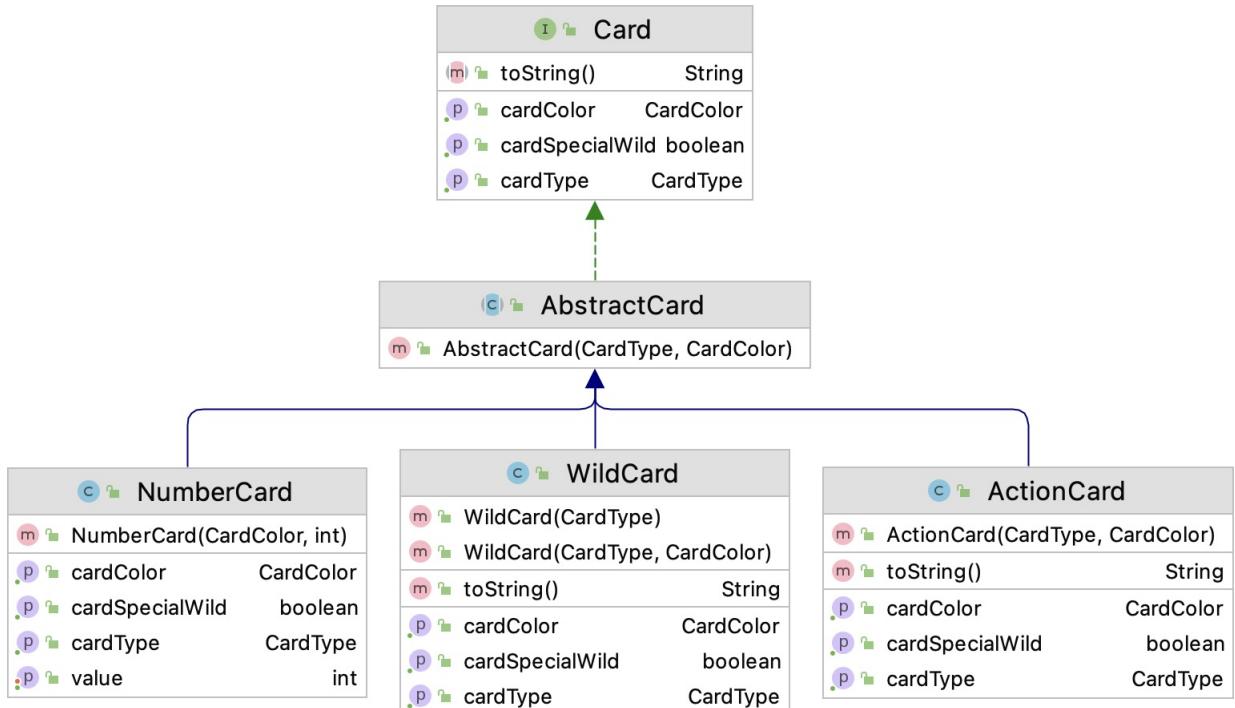


Figura 19: Logica UML delle carte

I tre tipi di carte, che ereditano dalla classe **AbstractCard**:

- **NumberCard**
- **WildCard**
- **ActionCard**

4.1 GameModel

Descriviamo a livello logico come abbiamo strutturato il `GameModel` del nostro gioco.

4.1.1 Attributi

A livello "logico", abbiamo pensato di implementare i seguenti attributi della classe:

- Il mazzo di carte, implementabile in Java da una lista. Abbiamo usato un `ArrayList`.
- Il mazzo delle carte usate, che rappresenta le carte che sono state utilizzate dai giocatori.
- La lista dei giocatori
- Gestore dei turni
- L'ultimo colore che è stato scelto

```
1 // Mazzi di carte usate
2 private CardDeck cardsDeck;
3 private UsedPile usedCards;
4
5 // ArrayList contenente i giocatori
6 private ArrayList<Player> players;
7 private int numberofPlayers;
8 private int maxNumberofPlayers;
9
10 // Gestore dei turni
11 private PlayerRoundIterator turnIterator;
12 private CardColor currentCardColor = null;
```

4.2 Metodi

4.2.1 Gestore mosse

Questo metodo si occupa di, dato come parametro una carta, di restituire gli effetti che essa compie. Abbiamo effettuato questo attraverso una `HashMap`.

Innanzitutto creiamo la hashmap di gestione delle mosse. Ciascuna `entry` della `HashMap` sarà composta nel seguente modo:

- **Key:** `ColorType`
- **Value:** `EvalMossa` (interfaccia)

```

1  public void loadMosse()
2  {
3      mapMosse = new ConcurrentHashMap<CardType, Mossa>();
4
5      mapMosse.put(CardType.NUMBER, () -> false);
6
7      mapMosse.put(CardType.WILD_COLOR, () -> true);
8
9      mapMosse.put(CardType.WILD_DRAW_FOUR, () -> {
10         addCardsToPlayer(4);
11         return true;
12     });
13
14     mapMosse.put(CardType.WILD_DRAW_TWO, () -> {
15         addCardsToPlayer(2);
16         return false;
17     });
18
19     mapMosse.put(CardType.REVERSE, () -> {
20         turnIterator.reverseDirection();
21         return false;
22     });
23
24     mapMosse.put(CardType.SKIP, () -> {
25         turnIterator.next();
26         return false;
27     });
28 }

```

Da cui quindi definiamo il metodo `evalMossa`, che permette, passata una carta ne determina gli effetti, e di riconoscere se è necessario o meno richiedere la scelta di un colore in base alla carta selezionata.

```

1  public boolean evalMossa(Card card)
2  {
3      players.get(turnIterator.getIndexCurrentPlayer()).removeCard(card);
4      usedCards.addCard(card);
5
6      boolean cardNeedsNewColorSelection = false;
7
8      if (mapMosse.containsKey(card.getCardType()))
9      {

```

```

10     if (card.getCardType() != CardType.REVERSE)
11         turnIterator.next();
12
13     cardNeedsNewColorSelection = mapMosse.get(card.getCardType()).evalMossa();
14 }
15
16 return cardNeedsNewColorSelection;
17 }
```

4.2.2 Controllo giocatore con una sola carta

Inoltre è molto utile ai fini del gioco riconoscere quando un giocatore ha una sola carta tra le carte della sua mano. Quindi abbiamo introdotto questo metodo, facendo *overloading*, in modo tale che fosse possibile sia un giocatore passando un indice numerico, oppure passando direttamente un oggetto di tipo giocatore.

```

1 /**
2 *
3 * @param player
4 * @return
5 */
6 public boolean hasPlayerOneCard(Player player)
7 {
8     boolean hasPlayerOneCard = false;
9
10    if (player.getHandCards().getNumberOfCards() == 1)
11    {
12        hasPlayerOneCard = true;
13    }
14
15    return hasPlayerOneCard;
16 }
17
18 /**
19 *
20 * @param player
21 * @return
22 */
23 public boolean hasPlayerOneCard(int playerIndex)
24 {
25     boolean hasPlayerOneCard = false;
```

```

26
27     if (players.get(playerIndex).getHandCards().getNumberOfCards() == 1)
28     {
29         hasPlayerOneCard = true;
30     }
31
32     return hasPlayerOneCard;
33 }
```

4.2.3 Controllo se il gioco è terminato

A questo punto dobbiamo creare un metodo che sia in grado di controllare se nel gioco si è arrivati al punto di fine. Per questo punto abbiamo introdotto un metodo che permettesse di controllare se fossero presenti le condizioni o meno di terminazione.

```

1  public boolean isGameOver()
2  {
3      boolean isGameOver = false;
4
5      for (Player player : players)
6      {
7          if (player.getHandCards().getNumberOfCards() == 0)
8          {
9              return true;
10         }
11     }
12
13     return isGameOver;
14 }
```

4.2.4 Controllo chi è il vincitore

Successivamente, nel caso in cui sia stato riconosciuto che la partita è terminata, abbiamo creato un metodo che per comodità, fosse in grado di ritornare quale fosse il giocatore vincitore.

```

1  public Player getWinnerPlayer()
2  {
3      Player playerWinner = null;
4
5      // Checks if there's actually a winning player
6      for (Player player : players)
7      {
```

```

8     if (player.getHandCards().getNumberOfCards() == 0)
9     {
10         playerWinner = player;
11     }
12 }
13
14 return playerWinner;
15 }
```

4.2.5 Gestore dei turni

A questo punto ci siamo occupati di gestire i turni dei giocatori. I turni, infatti, possono essere soggetti a cambiamenti durante la partita, infatti, se ad esempio un giocatore usa una carta *stop*, il giocatore successivo non sarà quello logicamente successivo al giocatore che ha usato la carta *stop*, bensì il giocatore ancora successivo.

Per fare questo abbiamo creato una classe `PlayerRoundIterator`, che gestisse proprio i turni di gioco. Essa è caratterizzata nel seguente modo:

```

1 private final ArrayList<Player> players;
2 private int current = 0;
3 private int previous = 0;
4 private Direction direction = Direction.CLOCKWISE;
```

4.2.6 Metodo `getNextIndex`

Il seguente metodo ritorna il prossimo index di gioco, in base al numero di giocatori che sono stati inizializzati, e in base al fatto che l'ordine di gioco sia `clockwise` o `counter-clockwise`.

```

1 public int getNextIndex()
2 {
3     previous = current;
4     int increment = direction == Direction.CLOCKWISE ? 1 : -1;
5     return (players.size() + current + increment) % players.size();
6 }
```

La classe `enum` `Direction` è composta nel seguente modo:

```

1 public enum Direction
2 {
3     CLOCKWISE, COUNTER_CLOCK_WISE;
4 }
```

Ovviamente sono presenti altri metodi di più semplice comprensione che sono descritti nei commenti JavaDoc.

4.3 Controller

Ora descriviamo il **controller** della nostra applicazione. Il *controller* risulta essere piuttosto complesso, in quanto esistono diverse modalità di gioco, che hanno delle meccaniche di funzionamento molto diverse tra di loro. Per questo, abbiamo creato una gerarchia di questo tipo:

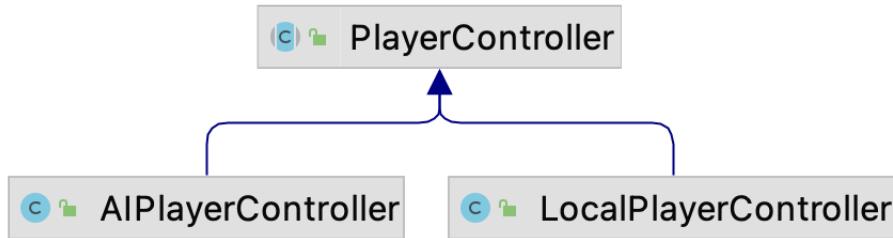


Figura 20: Gerarchia dei controller in UML

Una parte molto importante del controller è quella di gestire gli aggiornamenti della grafica. Per fare questo, abbiamo creato un metodo che fosse in grado di dare alla view tutti gli elementi aggiornati.

```
1  protected void updateView()
2  {
3      SwingUtilities.invokeLater(() -> {
4
5          // SETS TURN AND LOADS CARDS
6          view.setTurn(model.getCurrentPlayer().getNamePlayer());
7          view.loadCards(model.getPlayers().get(0).getHandCards(), 0);
8          view.loadCards(model.getPlayers().get(1).getHandCards(), 1);
9          view.changeDroppedCardView(model.getLastCardUsed(),
10             model.getCurrentCardColor());
11
12         // DISABLE BUTTONS
13         view.setSayUnoButtonVisible(false, model.getPreviousPlayerIndex());
14         view.setSayUnoButtonVisible(false, model.getCurrentPlayerIndex());
```

```

14
15 // GETS ALL CARDS VIEWS FROM GAMEVIEW
16 ArrayList<CardView> panelPlayerOneCards = view.getAllCards(0,
17     model.getPlayers().get(0).getHandCards().getNumberOfCards());
18 ArrayList<CardView> panelPlayerTwoCards = view.getAllCards(1,
19     model.getPlayers().get(1).getHandCards().getNumberOfCards());
20
21 // ADD MOUSE LISTENERS TO CARDS
22 mouseListener = new CardSelectedEvent(notifyObj);
23 panelPlayerOneCards.forEach(e -> e.addMouseListener(mouseListener));
24 panelPlayerTwoCards.forEach(e -> e.addMouseListener(mouseListener));
25
26 // ENABLES / DISABLE VIEW FOR PLAYERS
27 view.enableViewPlayer(model.getCurrentPlayerIndex(), true);
28 view.enableViewPlayer(model.getNextPlayerIndex(), false);
29
30 // CHECKS IF UNO BUTTON SHOULD BE ENABLED
31 view.setSayUnoButtonVisible(model.hasPlayerOneCard(model.getCurrentPlayer()),
32     model.getCurrentPlayerIndex());
33
34 // REPAINTS VIEW
35 view.repaint();
36 }
37 }
```

4.4 Rungame

Il metodo che si occupa di gestire il gioco è il seguente. Ovviamente il controller chiede al model se il gioco è finito, e continuerà a iterare il gioco finché il model non determina che il gioco è terminato.

```

1 @Override
2 protected void runGame()
3 {
4     while (!model.isGameOver())
5     {
6         synchronized (notifyObj)
7         {
8             try
9             {
10                 notifyObj.wait();
11             }
12         }
13     }
14 }
```

```

12         checkPlayerUno();
13
14     if (mouseListenerDrawnCard.isCardDrawn()
15         && model.getCurrentPlayer().getHandCards().getNumberOfCards() < 30)
16     {
17         if (model.getCurrentPlayer().getHandCards().getNumberOfCards() <= 30)
18         {
19             playerDrawCard();
20         }
21         else
22         {
23             JOptionPane.showMessageDialog(null, "Non puoi pescare ulteriori
24             carte!");
25         }
26     if (mouseListener.getSelected() != null)
27     {
28         playerSelectedCard();
29     }
30 }
31 catch (InterruptedException e)
32 {
33     e.printStackTrace();
34 }
35 }
36
37 updateView();
38
39 mouseListener.setCardSelectedNull();
40 mouseListenerDrawnCard.setCardDrawn(false);
41 }
42
43 gameOver();
44 }

```

5 Thread e Executor Service

Data la necessità di dover compiere molteplici operazioni nello stesso istante (vedi ad esempio continuare a *scannerizzare* l'eventuale arrivo di messaggi dal `client` o dal server, oppure il dover gestire le meccaniche di gioco), è stato necessario utilizzare i Thread. Dalla documentazione l'interfaccia `Runnable`⁸:

```
1 public interface Runnable  
2  
3 The Runnable interface should be implemented by any class whose instances are  
→ intended to be executed by a thread. The class must define a method of no  
→ arguments called run.
```

5.1 Thread

In Java, la classe ad hoc che implementa una CPU virtuale è la `java.lang.Thread`. È importante però, prima di vedere il codice Java all'opera, fissare bene i seguenti due punti:

1. Due o più thread possono condividere, indipendentemente dai dati, il codice che essi eseguono. Questo avviene quando tali thread eseguono il loro codice da istanze della stessa classe.
2. Due o più thread possono condividere, indipendentemente dal codice, i dati su cui eseguono delle operazioni. Questo avviene quando tali thread condividono l'accesso ad un oggetto comune.

Per questo motivo l'utilizzo di `Thread` è una questione piuttosto delicata, che spesso richiede l'utilizzo di varie sincronizzazioni.

`public Thread (Runnable target)`

5.2 Executor Service

A partire da Java 8 le API per la concorrenza hanno avuto interessanti modifiche con l'introduzione degli `Executor` come strato di più alto livello nella gestione dei `Thread`. Gli `Executor` sostituiscono la modalità di realizzazione diretta dei `Thread` consentendo l'esecuzione di task asincroni e pool di `Thread`. Dalla documentazione Java:

```
1 public interface ExecutorService  
2 extends Executor  
3  
4 An Executor that provides methods to manage termination and methods that can produce  
→ a Future for tracking progress of one or more asynchronous tasks.
```

⁸<https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html>

```

5
6 An ExecutorService can be shut down, which will cause it to reject new tasks. Two
→ different methods are provided for shutting down an ExecutorService. The
→ shutdown() method will allow previously submitted tasks to execute before
→ terminating, while the shutdownNow() method prevents waiting tasks from starting
→ and attempts to stop currently executing tasks. Upon termination, an executor
→ has no tasks actively executing, no tasks awaiting execution, and no new tasks
→ can be submitted. An unused ExecutorService should be shut down to allow
→ reclamation of its resources.

```

Ogni **Thread** all'interno del pool è riutilizzabile, un **Executor** infatti non termina autonomamente la sua esecuzione ma rimane in attesa dell'esecuzione di nuovi task. Per terminare un **Executor** dobbiamo esplicitamente invocare su di esso uno dei metodi di **shutdown** offerti dalla classe.

All'intero del nostro programma abbiamo utilizzato diversi tipi di pool di thread (in modo particolare i primi due):

- **newCachedThreadPool()** - Per un pool di **Thread** che può crescere dinamicamente riutilizzando i **Thread** creati precedentemente.
- **newFixedThreadPool()** - Per un pool di **Thread** a dimensione fissa riutilizzabili dall'**Executor**.
- **newScheduledThreadPool()** - Per un pool di **Thread** che eseguono task dopo un certo intervallo di tempo o periodicamente.

5.3 Thread utilizzati

Ovviamente nel nostro programma abbiamo avuto bisogno di diversi **Thread**, infatti abbiamo diversi task da compiere in contemporanea. Basti pensare a:

- Gestione del gioco
- Connessione tra client e server
- Gestione dei messaggi in ricezione ed invio della chat
- Gestione della grafica e dei suoi componenti

In tutti questi casi è stato necessario implementare dei **Thread**.

```

1 ExecutorService executor =
→ Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
2
3 executor.execute(this::listenForNewMessagesToSend);
4 executor.execute(this::listenToServer);
5 executor.execute(this::runGameLogic);

```

Tramite la prima riga di codice creiamo un nuovo pool fisso di Thread, utilizzando come numero di Thread il numero di processori della CPU del nostro computer.

5.4 Sincronizzazione eventi di gioco

Dato che possono avvenire diversi eventi di gioco, e spesso è necessario che i diversi Thread siano sincronizzati tra di loro, è necessario utilizzare diversi metodi di sincronizzazione dei Thread. Per questo è stato utilizzare i metodi `wait` e `notify`.

```
1  private final Object syncCardSelected = new Object();
2  private final Object syncObjectModel = new Object();
3  private final Object syncObjectChat = new Object();
```

5.5 Implementazione molteplici client

Come riportato nella sezione 6.2, abbiamo dovuto implementare una classe per gestire i diversi Thread per quanto riguarda il gioco.

6 Architettura Client/Server

Nella modalità gioco in multiplayer LAN, abbiamo necessitato di un architettura client/server che ci permetesse di far comunicare due o più applicazioni collocate su due diversi computer. Per fare questo, abbiamo creato due classi diverse, **NetClient** (client) e **NetServer** (server), entrambe le quali hanno come obiettivo la comunicazione tra client e server.

Nella logica di **client / server**, è possibile avere più **client** connessi ad un solo server. Per un **client** è necessario sapere l'indirizzo IP e la porta del server a cui si deve connettere, e nel nostro caso, è necessario che il server si trovi sulla stessa rete locale del **client**. Chiaramente sarebbe possibile avere un multiplayer che spazi a reti diverse dalla propria subnet locale, tuttavia questo avrebbe apportato delle problematiche non inerenti a questo corso.

Iniziamo con il vedere l'implementazione del **client**.

6.1 Client

Un **client** cerca di connettersi a un server ad un certo indirizzo IP a una certa porta. Ovviamente, più **client** possono coesistere nello stesso momento, connessi allo stesso server. Ovviamente il **client** non si preoccupa di questo, sarà il server a gestire la gestione contemporanea di più **client**.

6.1.1 Attributi

```
1  private final String IP_ADDRESS;
2  private final int port;
3  private boolean isConnected = false;
4
5  private Socket clientSocket;
6  private ObjectInputStream objInputStream;
7  private ObjectOutputStream objOutputStream;
8  private Object objReceivedGame = null;
```

Risulta ovviamente fondamentale, avere degli attributi che rappresentino IP e porta del server a cui il **client** *cerca* di connettersi. Inoltre, aggiungiamo un flag **boolean** che rappresenti se il **client** si sia effettivamente connesso o meno al server.

Per la connessione effettiva del client al server è fondamentale l'utilizzo della classe **Socket**⁹.

```
1  public class Socket
2  extends Object
3  implements Closeable
```

⁹<https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html>

```

1 This class implements client sockets (also called just "sockets"). A socket is an
→ endpoint for communication between two machines.
2
3 The actual work of the socket is performed by an instance of the SocketImpl class.
→ An application, by changing the socket factory that creates the socket
→ implementation, can configure itself to create sockets appropriate to the local
→ firewall.

```

6.2 Server

Il server si occupa di gestire la connessione con i client. Nel nostro caso abbiamo implementato la funzionalità con un solo client, tuttavia, per implementare più client che giochino contemporaneamente basterebbe aggiungere alcune funzionalità con i `Thread`, come descritto nella sezione 8

6.2.1 Attributi

```

1 // MODEL AND VIEW
2 private GameModel model;
3 private final TableView view;
4
5 // NETWORK ATTRIBUTES
6 private Socket client;
7 private ServerSocket serverSocket;
8 private final int PORT;
9 private boolean isConnected = false;
10
11 // OUTPUT / INPUT STREAMS
12 private Object objReceivedGame = null;
13 private String playerNameServer = null;
14 private String playerNameClient = null;
15 private CardSelectedEvent mouseListener;
16 private final CardDrawnEvent mouseListenerDrawnCard;
17
18 // MULTIPLE CLIENTS
19 private final int playerIndex = 0;
20 private int totalHostsConnected = 0;
21 private final int numberOfPlayers;
22 private ArrayList<ClientManager> clientManager = new ArrayList<ClientManager>();
23 private final Object syncObjectConnections = new Object();
24
25 // SYNC

```

```

26  private final Object syncCardSelected = new Object();
27  private final Object syncObjectModel = new Object();
28  private final Object syncObjectChat = new Object();

```

6.2.2 Attributi

```

1   // MODEL AND VIEW
2   private GameModel model;
3   private final TableView view;
4
5   // NETWORK ATTRIBUTES
6   private Socket client;
7   private ServerSocket serverSocket;
8   private final int PORT;
9   private boolean isConnected = false;
10
11  // OUTPUT / INPUT STREAMS
12  private Object objReceivedGame = null;
13  private String playerNameServer = null;
14  private String playerNameClient = null;
15  private CardSelectedEvent mouseListener;
16  private final CardDrawnEvent mouseListenerDrawnCard;
17
18  // MULTIPLE CLIENTS
19  private final int playerIndex = 0;
20  private int totalHostsConnected = 0;
21  private final int numberofPlayers;
22  private ArrayList<ClientManager> clientManager = new ArrayList<ClientManager>();
23  private final Object syncObjectConnections = new Object();
24
25  // SYNC
26  private final Object syncCardSelected = new Object();
27  private final Object syncObjectModel = new Object();
28  private final Object syncObjectChat = new Object();

```

6.2.3 Gestore client

Abbiamo creato una classe per gestire molteplici client allo stesso tempo.

```

1 package unibs.pajc.uno.controller.net;
2
3 import java.io.EOFException;

```

```

4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.io.ObjectOutputStream;
7 import java.net.Socket;
8
9 import javax.swing.JOptionPane;
10
11 import unibs.pajc.uno.model.GameModel;
12 import unibs.pajc.uno.view.TableView;
13
14 public class ClientManager extends Thread
15 {
16     private TableView view;
17
18     private Socket client;
19     private String playerNameClient;
20     private int clientIndex;
21
22     private ObjectInputStream objInputStream;
23     private ObjectOutputStream objOutputStream;
24
25     private final Object syncObjectModel;
26     private final Object syncObjectChat;
27
28     private Object objReceivedGame = null;
29
30     private boolean isConnected = false;
31
32     public ClientManager(TableView view, Socket clientSocket, int clientIndex, Object
33         → syncObjectModel,
34         Object syncObjectChat)
35     {
36         this.client = clientSocket;
37         this.clientIndex = clientIndex;
38
39         this.syncObjectModel = syncObjectModel;
40         this.syncObjectChat = syncObjectChat;
41     }
42     @Override

```

```

43 public void run()
44 {
45     try
46     {
47         objOutputStream = new ObjectOutputStream(client.getOutputStream());
48         objInputStream = new ObjectInputStream(client.getInputStream());
49         isConnected = true;
50         System.out.println("[SERVER] - CONNECTED TO CLIENT: " + isConnected);
51     }
52     catch (IOException e)
53     {
54         isConnected = false;
55     }
56
57     if (isConnected)
58     {
59         try
60         {
61             // TRIES TO READ CLIENT NAME
62             playerNameClient = (String) objInputStream.readObject();
63
64             // WAITS FOR A VALID CLIENT NAME
65             while (playerNameClient.equals(""))
66             {
67                 playerNameClient = (String) objInputStream.readObject();
68             }
69         }
70         catch (IOException e)
71         {
72             System.out.println("[SERVER] - No player init message received");
73         }
74         catch (NullPointerException e)
75         {
76             System.out.println("[SERVER] - No player init message received");
77         }
78         catch (ClassNotFoundException e)
79         {
80             e.printStackTrace();
81         }
82     }
}

```

```

83     }
84
85     public void listenToClient()
86     {
87         Object objReceived = null;
88
89         try
90         {
91             objReceived = objInputStream.readObject();
92
93             if (objReceived != null && objReceived instanceof String && ((String)
94                 ↪ objReceived).length() > 0)
95             {
96                 System.out.println("[SERVER] - Message received from client: " +
97                 ↪ objReceived);
98
99                 view.addChatMessage((String) objReceived, playerNameClient);
100            }
101            if (objReceived != null && objReceived instanceof GameModel)
102            {
103                synchronized (syncObjectModel)
104                {
105                    syncObjectModel.notify();
106                }
107
108                objReceivedGame = objReceived;
109                System.out.println("[SERVER] - Game model received");
110            }
111        }
112        catch (EOFException e)
113        {
114            JOptionPane.showMessageDialog(view, "Network error!");
115            System.exit(0);
116        }
117        catch (IOException e)
118        {
119            System.out.println("Errors in listening to the client");
120            System.exit(0);
121        }
122        catch (ClassNotFoundException e)

```

```

121    {
122        System.out.print("Class not found");
123        System.exit(0);
124    }
125 }
126
127 public void sendToClient(Object objToSend)
128 {
129     try
130     {
131         objOutputStream.reset();
132         objOutputStream.writeObject(objToSend);
133
134         if (objToSend instanceof String)
135         {
136             System.out.println("Message sent: " + objToSend);
137         }
138
139         objOutputStream.flush();
140     }
141     catch (IOException e)
142     {
143         System.out.println("Error while sending - Couldn't send object to client");
144         e.printStackTrace();
145     }
146 }
147 }
```

7 Intelligenza artificiale

Per avere un avversario che fosse in grado di giocare senza influsso umano, abbiamo dovuto creare un'intelligenza artificiale che fosse in grado di scegliere la carta al posto del giocatore. Per fare questo, abbiamo creato un classe `AI`.

Abbiamo abusato del termine `AI`, in quanto chiaramente il numero di operazioni che è in grado di svolgere sono molto limitate e di fatto compie solo operazioni basiche, senza prendere in considerazione nessun algoritmo particolare, che tuttavia sarebbe potuto essere implementato, ma è stato ritenuto superfluo ai fini del corso. Per `AI` intendiamo quindi un sistema che sia in grado di prendere decisioni simulando un giocatore.

Tuttavia è sufficiente per avere una partita soddisfacente¹⁰.

7.1 Metodo `determineNextMossa`

Questo metodo è in grado di determinare la prossima mossa (migliore) che può prendere il giocatore in base alla carta della sua mano e all'ultima carta posizionata sul tavolo da gioco.

Innnanzituttto è necessario filtrare le carte che non sono ritenute valide. Per questo è necessario interpellare il model, chiamando il metodo `isPlacedCardValid`, che passato un oggetto di tipo carta, ritorna se essa è valida in base alle condizioni di gioco di quel momento (carta posizionata sul tavolo, turno, ecc).

```
1 // Stampa a schermo solo le carte valide
2 playerCardList.stream()
3         .filter(e -> model.isPlacedCardValid(e))
4         .forEach(e -> System.out.print(e.getCardColor() + " , " +
5             e.getCardType()));
```

In questo modo siamo in grado di ottenere dalla mano del giocatore, solo le carte che effettivamente sono valide, e operare solo su queste ultime.

A questo punto è necessario indirizzare il computer verso la scelta migliore da effettuare. Ad esempio, nel caso tra le carte della mano del giocatore ci fosse una carta cambio colore, è opportuno cambiare il colore corrente del gioco in modo da favorire il giocatore.

Per fare questo, abbiamo implementato il metodo `determineMostPresentColor`, il quale ritorna il colore più presente tra le carte in mano ad un giocatore.

¹⁰Per soddisfacente si intende che il giocatore sembra compiere mosse "umane", cioè effettivamente sensate e non prese in modo *casuale*.

```

1  private CardColor determineMostPresentColor(ArrayList<Card> cardsList)
2  {
3      HashMap<CardColor, Integer> hashColorMap = new HashMap<CardColor, Integer>();
4      CardColor maxColor = null;
5      int maxOccurrences = 0;
6
7      for (Card card : cardsList)
8      {
9          if (hashColorMap.containsKey(card.getCardColor()))
10         {
11             hashColorMap.put(card.getCardColor(), hashColorMap.get(card.getCardColor())
12                 + 1);
13         }
14         else
15         {
16             hashColorMap.put(card.getCardColor(), 0);
17         }
18     }
19
20     if (!hashColorMap.isEmpty())
21     {
22         for (Entry<CardColor, Integer> entry : hashColorMap.entrySet())
23         {
24             if (entry.getValue() >= maxOccurrences)
25             {
26                 maxColor = entry.getKey();
27             }
28         }
29     }
30
31     return maxColor != null ? maxColor : CardColor.RED;
}

```

Abbiamo utilizzato un `HashMap` nel seguente modo:

- **Key:** Il colore della carta
- **Value:** Il numero di occorrenze del colore (key)

Chiaramente siamo interessati a ritornare il valore della carta che ha il valore più diffuso nel nostro mazzo di carte, in modo tale da poterlo scegliere nel caso di un cambio colore.

Nel caso in cui un giocatore non abbia carte disponibili valide, allora verrà presa una carta dal mazzo.

- Cerco se il giocatore ha delle carte valide
 - Il giocatore ha carte valide
 - Il giocatore ha carte speciali? Se sì, allora la uso
 - Ha una carta che obblighi l'altro giocatore a pescare? Se sì, allora la uso
 - Ultimo caso, la scelta è irrilevante e scelgo una carta tra quelle disponibili in modo casuale
 - Il giocatore non ha carte valide, pesco una carta dal mazzo
 - Passo il turno al giocatore successivo

8 Conclusioni finali

Il progetto è stato impegnativo, in quanto ci ha richiesto di rivedere tutti gli argomenti del corso, e spesso ha richiesto molti approfondimenti, che ci hanno portato a lavorare a questo progetto per svariati mesi.

Problemi riscontrati

Una volta affrontati tutti gli argomenti del corso (in modo particolare i `Thread`), il progetto è stato completato senza particolari problemi. Gli argomenti trattati dal corso sono infatti stati sufficienti al completamento del progetto. Abbiamo cercato di porre cura anche nei dettagli, curandoci di sistemare le eccezioni, di sistemare bug e problemi di ogni sorta, anche se raramente ancora accade che avvengano dei bug, che spesso però risultano innocui, ma che nella maggior parte dei casi sono imputabili a errori di rete.

Miglioramenti

Eventuali miglioramenti futuri riguarderebbero certamente l'implementazione di più di due giocatori, che risulterebbe comunque piuttosto semplice (ma *laboriosa*) in quanto il model già supporta un numero elevato di giocatori (di fatto, senza limiti), e quindi si tratterebbe soltanto di un lavoro meccanico di sistemazione della GUI e la gestione di più `Thread` per i diversi `client` che si avrebbero.

Tuttavia, seppur impegnativo, crediamo sia stato il modo migliore per imparare gli argomenti del corso in quanto abbiamo affiancato alla teoria molta pratica e come gruppo riteniamo i risultati più che soddisfacenti per le competenze apprese.

Repository github

Il progetto è reperibile pubblicamente su github:

<https://github.com/xStevatt/uno-card-game>

Con ogni probabilità miglioreremo il progetto in futuro, implementando alcune funzionalità che per motivi di tempo non siamo riusciti ad implementare.