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: 22 giugno 2022

Indice

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 gratifica, del framework **Swing**, che verrà descritta approfonditamente successivamente nella sezione **??**.

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 giocatori 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. L'intero mazzo di carte è presentato nella figura 1.

- 19 carte di colore Blu, numerate dall'1 al 9 (2 serie) più uno 0
- 19 carte di colore Rosso, numerate dallì1 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)

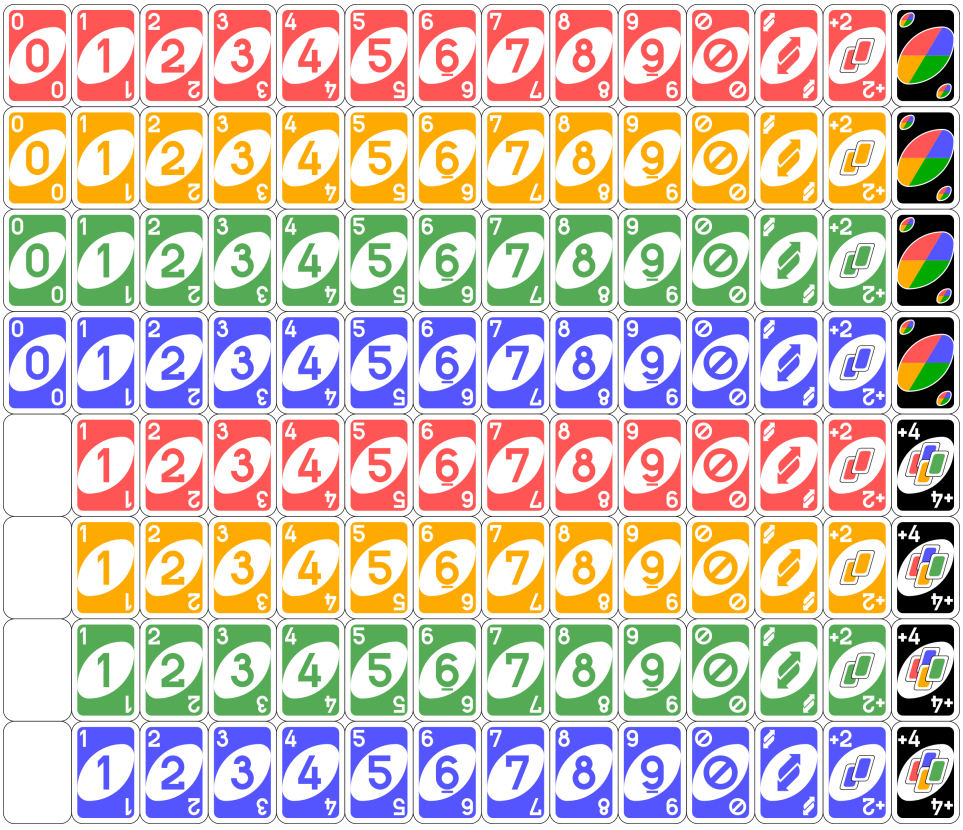

Figura 1: L'intero set di carte di uno

1.3 Successione di gioco

regole 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

- Una carta ***Pesca quattro*** obbliga l'altro giocatore a pescare quattro carte
- 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, inizia un giocatore. L'estrazione del giocatore che inizia per prima 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. 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* 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 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 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 2: Schermata principale del gioco da cui selezionare la modalità di gioco

1.5 Gioco 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.

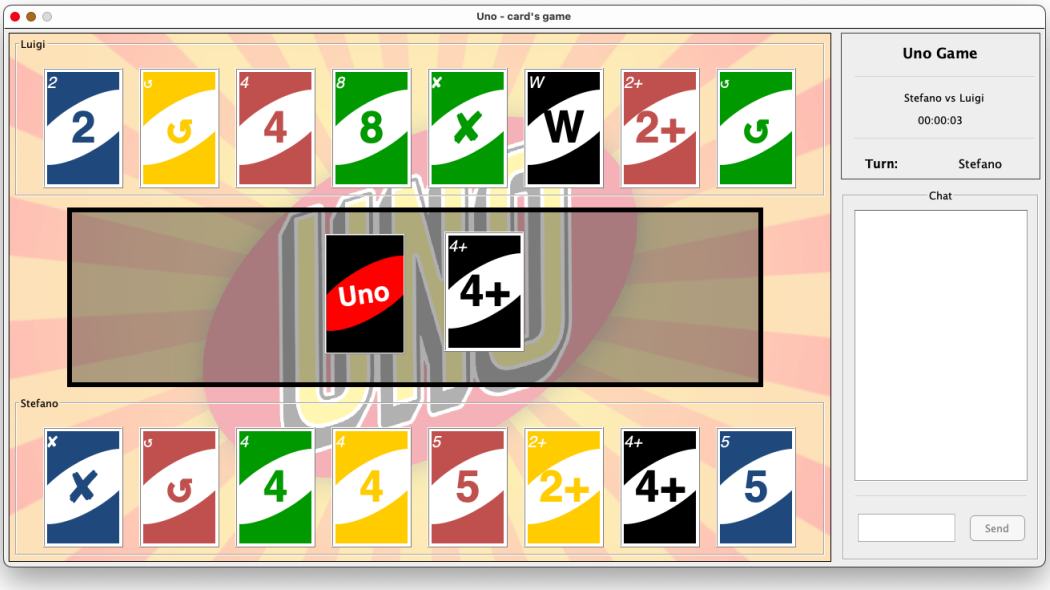

Figura 3: Il gioco in modalità offline: entrambi i giocatori hanno le carte scoperte

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

1.5.1 Gioco online

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 **??**.

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

Finestra di inizio gioco:

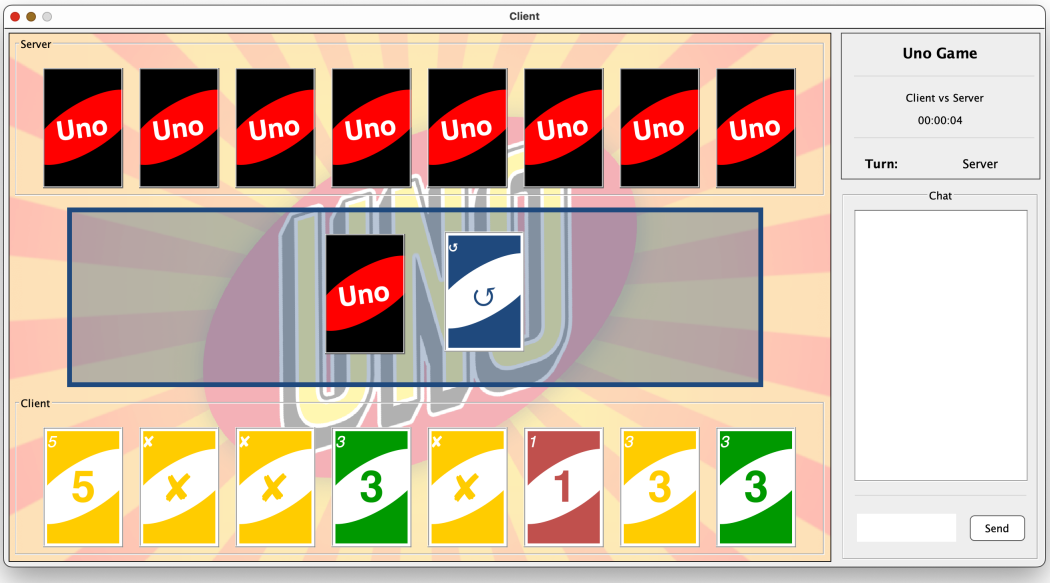

Figura 4: Finestra di gioco per il client

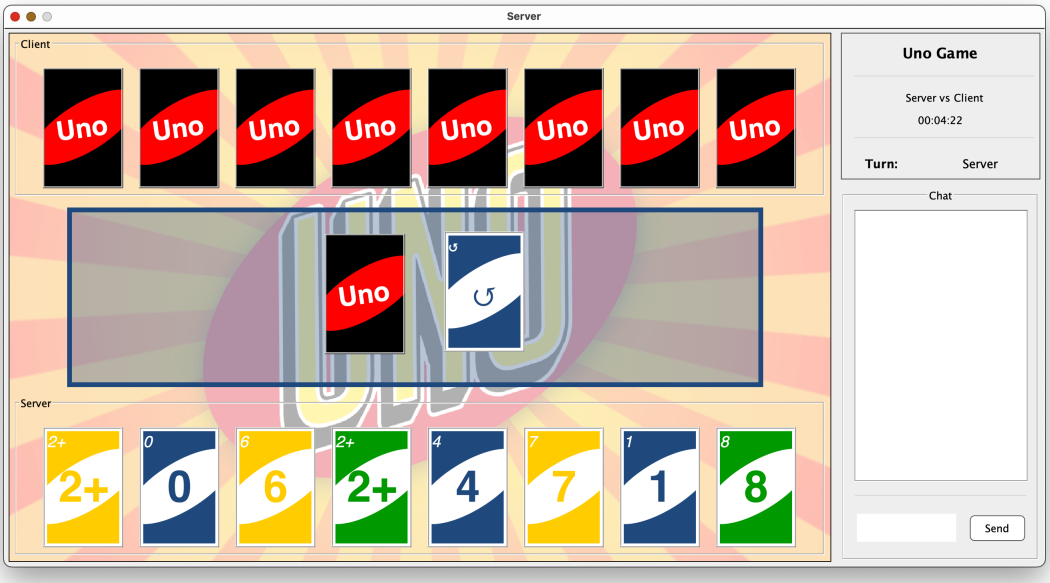

Figura 5: Finestra di gioco per il server

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

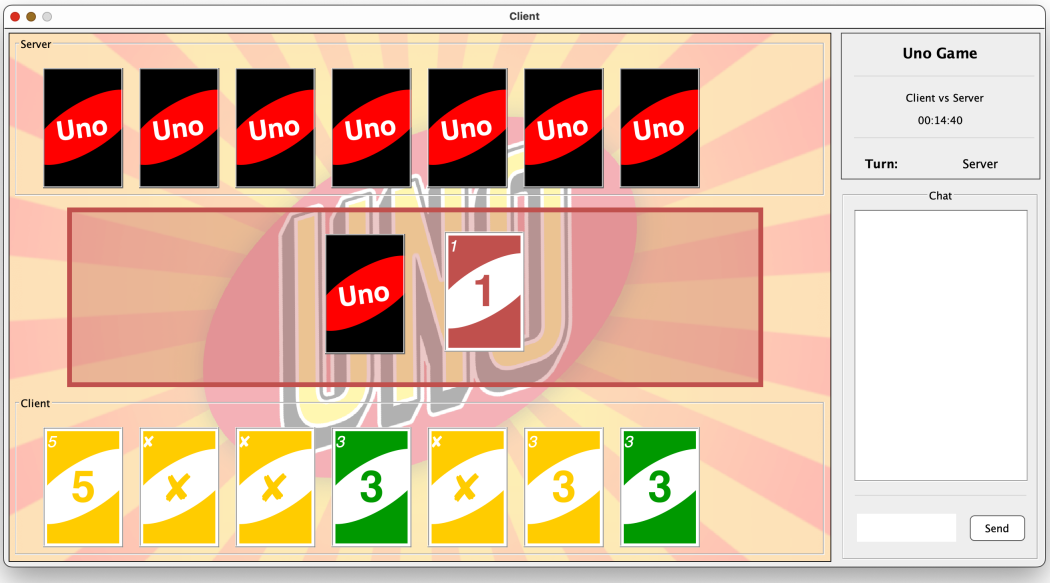

Figura 6: Finestra di gioco per il client

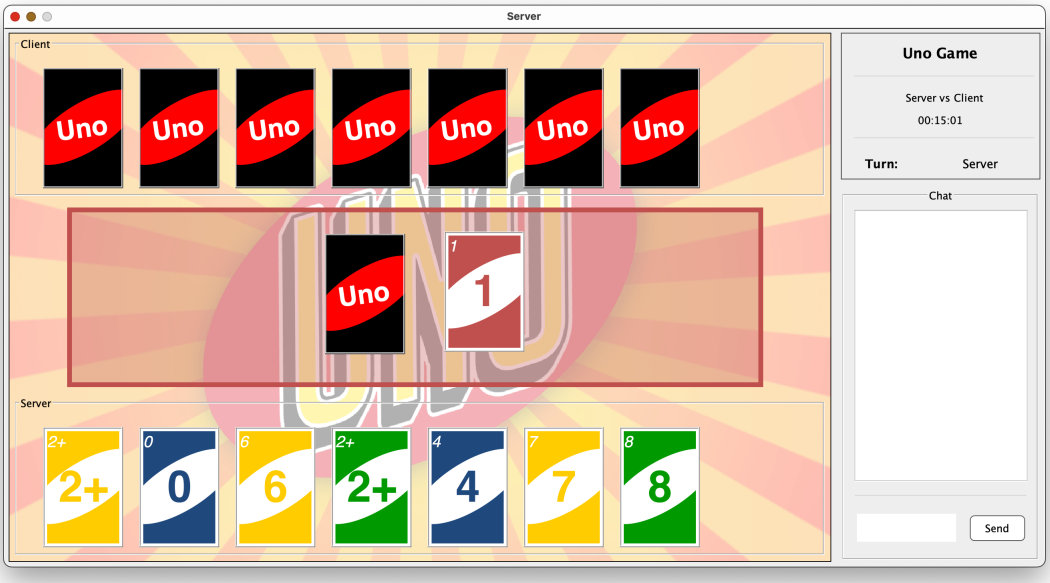

Figura 7: Finestra di gioco per il server

2 Pattern Model-View-Controller

Abbiamo utilizzato il pattern architetturale 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*.

3 View

La view del gioco è rappresentata dalle seguenti classi:

```
public class TableView
```

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

```
public class CardView
```

Rappresenta la view di una singola carta;

```
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 8.

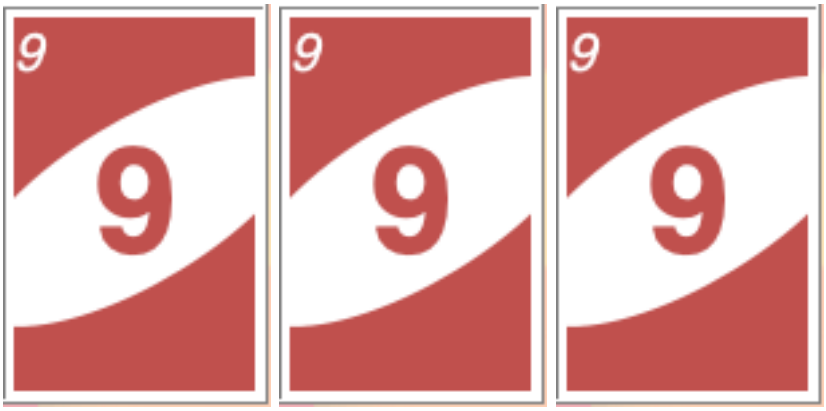

Figura 8: Una carta da gioco, numerica e di colore rosso

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`

```
    private void fillBackground(Graphics2D g2, Color cardColor)
  {
    g2.setColor(Color.WHITE);
    g2.fillRect(0, 0, cardWidth, cardHeight);

    g2.setColor(cardColor);
    g2.fillRect(margin, margin, cardWidth - 2 * margin, cardHeight - 2 * margin);
  }
```

Il metodo mostra 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).

3.1.2 Metodo `drawWhiteOvalInCenter`

```
  private void drawWhiteOvalInCenter(Graphics2D g2)
  {
    var transformer = g2.getTransform();
    g2.setColor(Color.white);
    g2.rotate(45, (double) cardWidth * 3 / 4, (double) cardHeight * 3 / 4);
    g2.fillOval(0, cardHeight / 4, cardWidth * 3 / 5, cardHeight);

    g2.setTransform(transformer);
  }
```

Il metodo disegna un ovale e lo ruota di 45°.

3.1.3 Metodo `drawWhiteOvalInCenter`

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

    g2.setColor(cardColor);
    g2.setFont(defaultFont);
    g2.drawString(value, cardWidth / 2 - stringWidth, cardHeight / 2 + fontHeight);
  }
```

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 utizzato `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'user 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. 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 carta 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,
      ↪  RenderingHints.VALUE_ANTIALIAS_ON);
10    g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
      ↪  RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
11
```

```
12      var cardColor = Util.convertCardColor(card.getCardColor());
13
14      fillBackground(g2, cardColor);
15      drawWhiteOvalInCenter(g2);
16      drawValueInCenter(g2, cardColor);
17      drawValueInCorner(g2);
18    }
```

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

```
public class GameModel
```

4.1 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

```
    // Mazzi di carte usate
  private CardDeck cardsDeck;
  private UsedPile usedCards;

    // ArrayList contenente i giocatori
  private ArrayList<Player> players;
  private int numberOfPlayers;
  private int maxNumberOfPlayers;

    // Gestore dei turni
  private PlayerRoundIterator turnIterator;
  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 uno switch.

```
  public boolean evalMossa(Card card)
  {
    players.get(turnIterator.getIndexCurrentPlayer()).removeCard(card);
    usedCards.addCard(card);
```

```
5
6       boolean newColorNeedsSelection = false;
7
8       switch (card.getCardType())
9       {
10      case NUMBER:
11
12        turnIterator.next();
13        // NOTHING NEEDS TO BE DONE WHEN CARD PLACED IS A NUMBER
14        newColorNeedsSelection = false;
15
16        break;
17
18      case WILD_COLOR:
19
20        turnIterator.next();
21        // NEW COLOR NEEDS TO BE SELECTED
22        newColorNeedsSelection = true;
23
24        break;
25
26      case WILD_DRAW_FOUR:
27
28        turnIterator.next();
29        // ADD CARDS TO NEXT PLAYER
30        for (int i = 0; i < 4; i++)
31        {
32          turnIterator.getCurrentPlayer().addCard(getCardFromDeck());
33        }
34
35        // NEW COLOR NEEDS TO BE SELECTED
36        newColorNeedsSelection = true;
37
38        break;
39      case WILD_DRAW_TWO:
40
41        turnIterator.next();
42        // ADD CARDS TO NEXT PLAYER
43        for (int i = 0; i < 2; i++)
44        {
```

```
45          turnIterator.getCurrentPlayer().addCard(getCardFromDeck());
46        }
47
48        // NEW COLOR NEEDS TO BE SELECTED
49        newColorNeedsSelection = false;
50
51        break;
52
53      case SKIP:
54
55        // SKIPS ONE PLAYER
56        turnIterator.next();
57        turnIterator.next();
58        newColorNeedsSelection = false;
59
60        break;
61
62      case REVERSE:
63
64        // REVERSES ORDER
65        turnIterator.reverseDirection();
66        newColorNeedsSelection = false;
67
68        break;
69      }
70
71      return newColorNeedsSelection;
72    }
```

4.2.2 Controllo giocatore con una sola carta

```
1   public boolean hasPlayerOneCard(Player player)
2   {
3     boolean hasPlayerOneCard = false;
4
5     if (player.getHandCards().getNumberOfCards() == 1)
6     {
7       hasPlayerOneCard = true;
8     }
9
```

```
10      return hasPlayerOneCard;
11    }
```

4.3 Controller