Relazione Progetto

Matteo Noro
Giovanni Cocco

1 Scopo del progetto

Il seguente progetto di Programmazione ad oggetti si propone di sviluppare un piccolo gioco di carte di nome *UGNO*, dove l'utente può giocare contro un numero arbitrario (scelto all'avvio di una nuova partita) di avversari comandati dal computer.

Il gioco che quindi si va a creare prende forma e spunto dalle logiche del famoso gioco di carte UNO, dal quale ne prende ispirazione principalmente per struttura e regole, nonostante alcune sostanziali differenze da quest'ultime per facilitarne l'accessibilità.

Le tipologie di carta sono due: le *carte numero* e le *carte effetto;* quest'ultime sono in particolare la *Salta Turno,* la *Pesca* e la *Random.*

All'inizio della partita, l'utente giocatore può scegliere sia il numero di avversari contro cui giocare sia il numero di carte iniziali.
Lo scopo è, come nella versione originale del gioco, di finire tutte le carte nella propria mano.
Quando la partita inizia, ci sarà già una carta casuale al centro del tavolo.
Il giocatore ad ogni turno dovrà tentare di buttare più carte possibili, purché, ovviamente, siano sempre compatibili con l'ultima presente nella pila centrale.
Se nessuna carta viene gettata, alla mano ne sarà aggiunta una nuova casuale, pescata casualmente dal mazzo.

1.1 Elenco carte

Si elencano di seguito tutti i tipi di carta presenti nel gioco, quali possono avere un numero (nel caso nelle carte numero) e un colore. I colori presenti sono i tradizionali Rosso, Blu, Verde e Giallo; eccezion fatta per la iconica carta *Pesca Quattro* che presenta il colore nero.
Si evidenziano quindi, per ogni tipo, la via di identificazione, la caratterizzazione, la compatibilità e l'eventuale effetto.

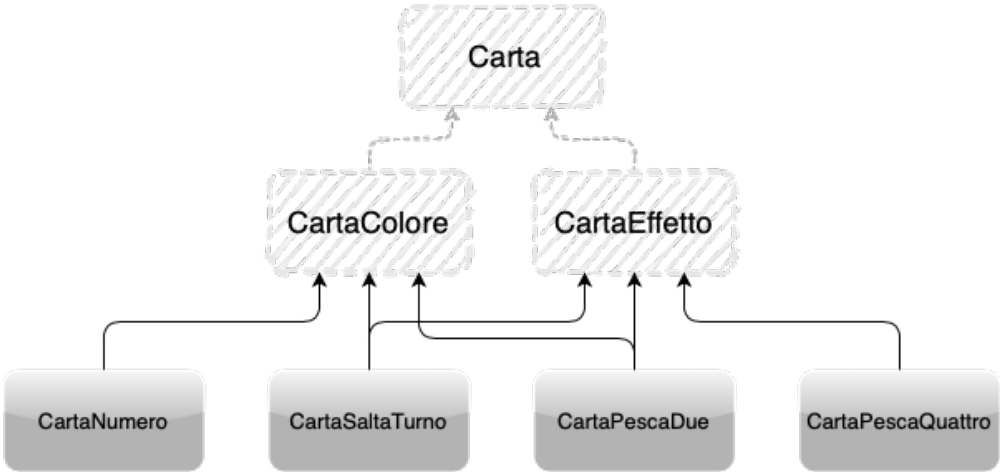
- **Carte numero**
  a. Identificate tramite un numero compreso tra *1 e 9*
  b. Caratterizzate da un numero compreso tra 1 e 9 e un colore
  c. Sono compatibili solamente con altre carte che presentino lo stesso numero e/o lo stesso colore (oppure il nero).
     Questa risulta quindi la condizione affinché possano essere "gettate" dal giocatore

- **Carte Salta Turno**
  a. Identificate tramite la lettera *S*
  b. Caratterizzate da un colore
  c. Sono compatibili con un'altra carta *Salta Turno* e/o con una che abbia lo stesso colore (oppure il nero)
  d. Applicano il seguente effetto: il giocatore successivo al giocante salta il turno. L'effetto è cumulabile, quindi se giocata più volte dallo stesso giocatore se ne propaga l'azione.

- **Carte Pesca Due**
  a. Identificate tramite la lettera *D*
  b. Caratterizzate da un colore
  c. Sono compatibili con un'altra carta *Pesca Due* e/o con una che abbia lo stesso colore (oppure il nero)
  d. Applicano il seguente effetto: il giocatore successivo al giocante aggiungerà alla sua mano due carte.

- **Carta Pesca Quattro**
  a. Identificate tramite la lettera *Q*
  b. Caratterizzate, diversamente dalle carte *Pesca Due*, dal colore nero.
  c. Sono compatibili con qualsiasi carta.
  d. Applicano il seguente effetto: il giocatore successivo al giocante aggiungerà alla sua mano quattro carte.

2. Descrizione Delle Gerarchie



Dallo schema sovrastante si può evincere la scala gerarchica delle carte.
Si noti come le classi astratte, a differenze di quelle concrete, siano contraddistinte da un bordo tratteggiato nel riquadro raffigurativo e un effetto “schizzo”.
Una freccia da una classe verso un’altra significa che la prima deriva dalla seconda; se la freccia è tratteggiata (sempre con effetto “schizzo”) significa che una classe deriva virtualmente dall’altra.

2.1 Carta

La classe in cima allo schema di cui prima è la classe ***Carta****:* questa è la base astratta virtuale di tutta la gerarchia.
Essa contiene:
- Un distruttore virtuale ***virtual ~Carta();*** per supportare il polimorfismo
- Un metodo virtuale puro ***virtual Carta* clone() const = 0;*** per la gestione profonda della memoria tramite metodo di clonazione
- Un metodo virtuale puro ***virtual bool valida(const Carta& c) const = 0;*** per il controllo di compatibilità fra carte

La classe ***CartaColore*** è una classe astratta, quale aggiunge la caratterizzazione per colore delle carte.
Essa contiene:
- Un campo dati proprio ***colore_*** , per implementare il concetto di colore nelle carte
- Un costruttore di default ***CartaColore();*** che genera una carta random
- Un costruttore fisso ***CartaColore(int colore);***
- Una funzione getter pubblica ***int GetColore() const;***
- Una eccezione ***colore_fuori_range;*** che viene invocata nelle situazione in cui il colore passato al costruttore non sia valido

La classe ***CartaEffetto*** è una classe astratta, quale aggiunge la caratteristica di applicare un effetto alle carte relative.
Essa contiene:
- Un metodo virtuale puro ***virtual void applicaEffetto(Partita* partita) const = 0;*** che dato un puntatore a partita, applica l’effetto della carta gettata.

La classe ***CartaNumero*** è una classe concreta ed eredita da *CartaColore.*
Essa contiene
- Un campo dati proprio ***numero_*** , per implementare la identificazione delle carte associate
- Un costruttore di default ***CartaNumero()*** che genera una carta random con numero e colore
- Un costruttore fisso ***CartaNumero(int numero, int colore);*** che genera una carta dati un numero e un colore
- Una funzione getter pubblica ***virtual int getNumero() const;***
- Tutti i metodi richiesti per essere istanziabile (cioè ***bool valida(const Carta& c) const;*** e ***CartaNumero* clone() const;***), facendone l'overriding
- Una eccezione ***numero_fuori_range;*** che viene invocata nella situazione in cui il numero passato al costruttore non sia valido

La classe ***CartaSaltaTurno*** è una classe concreta ed eredita da *CartaColore* e *CartaEffetto*.
Essa contiene:
- Un costruttore di default ***CartaSaltaTurno();*** che genera una carta di tipo CartaSaltaTurno con un colore random
- Un costruttore fisso ***CartaSaltaTurno(int colore);*** che genera una carta di tipo CartaSaltaTurno dato un colore
- Tutti i metodi richiesti per essere istanziabile (cioè ***bool valida(const Carta& c) const; CartaSaltaTurno* clone() const;*** e ***void applicaEffetto(Partita* partita) const***), facendone l'overriding

La classe ***CartaPescaDue*** è una classe concreta ed eredita da *CartaColore* e *CartaEffetto*.
Essa contiene:
- Un costruttore di default ***CartaPescaDue();*** che genera una carta di tipo CartaPescaDue con un colore random
- Un costruttore fisso ***CartaPescaDue(int colore);*** che genera una carta di tipo CartaPescaDue dato un colore
- Tutti i metodi richiesti per essere istanziabile (cioè ***bool valida(const Carta& c) const; CartaPescaDue* clone() const;*** e ***void applicaEffetto(Partita* partita) const***), facendone l'overriding

La classe ***CartaPescaQuattro*** è una classe concreta ed eredita da *CartaEffetto*.
Essa contiene:
- Un costruttore di default ***CartaPescaQuattro();*** che genera una carta di tipo CartaPescaQuattro nell'unico modo in cui può essere generata
- Tutti i metodi richiesti per essere istanziabile (cioè ***bool valida(const Carta& c) const; CartaPescaQuattro* clone() const;*** e ***void applicaEffetto(Partita* partita) const***), facendone l'overriding

2.2 Giocatore

La classe ***Giocatore*** è una base concreta polimorfa. Da questa ne deriva la classe ***GiocatoreCPU*** che espande la classe base tramite costruttore e una funzione ***bool stepMossa(DeepPtr<Carta>& tavolo, Partita* partita);*** per avanzare di una mossa: ritorna false quando ha finito le mosse nel proprio turno.
La classe Giocatore contiene i seguenti metodi:

```
  // Costruttore a 1 parametro col numero di carte iniziali
     Giocatore(int numCarte);
   // Distruttore virtuale per poliformismo
     virtual ~Giocatore();
   // Getter publici
     virtual const Carta& getCarta(int id) const;
     virtual int getNumeroCarteInMano() const;
     virtual bool haVinto() const;
   // Gioca la carta
     virtual void giocaCarta(int id, Partita* partita);
```

```
      virtual void giocaCarta(const Lista<DeepPtr<Carta>>::iterator& it,
      Partita* partita);
    // Finisce il turno pescando se non ha gettato carte in questo turno
      virtual void fineTurno();
    // Pesca n carte, default una
      virtual void pesca(int numCarte = 1);
    // Eccezioni
      class id_carta_out_of_range {};
    // Setter per caricare una partita
      virtual void aggiungiCartaAllaMano(const DeepPtr<Carta>& carta);
```

Inoltre, la classe contiene due campi dati privati:

```
      Lista<DeepPtr<Carta>> mano_;
      bool haGettato_;
```

2.2 Partita

La classe ***Partita*** è il centro del gioco in un certo senso, ed è stata costruita per predisporla al polimorfismo e, dunque, a future espansioni del programma.
La classe contiene i seguenti metodi e classi per eccezioni:

```
    //Costruttore di default
      Partita();
    // Costruttore con numero di giocatori totali e carte in mano a ciascuno
      Partita(int numGiocatori, int numCarteIniziali);
    // Distruttore
      virtual ~Partita();
    // Giocata la carta di indice idCarta nella mano del giocatore umano,
ritorna false se la carta non è giocabile
    // con quella correntemente sul tavolo
      virtual bool playerGiocaCarta(int idCarta);
    // Finisce il turno del giocatore umano, pesca automaticamente una carta se
in questo turno non ha buttato alcuna carta
      virtual void playerFineTurno();
    // Avanza di una mossa i giocatori controllati dal computer
      virtual void eseguiStepGiocatoreCPU();
    // Getter publici
      virtual const Giocatore& getGiocatore(int id) const;
      virtual const Carta& getCartaSulTavolo() const;
      virtual int getNumeroGiocatori() const;
      virtual int getTurno() const;
```

```
    // Funzioni per ausiliari per carte effetto
      virtual void applicaCartaSulTavolo();
      virtual void setTurno(int turno);
      virtual void pescaGiocatore(int id, int numCarte) const;
      virtual void aumentaTurniDaPassare();
      virtual void avanzaTurni(int turni);
    // Funzione statica che genera una carta, al fine di pescare e per non
permettere di contare le carte
    // si genera random una carta al posto di avere un mazzo
      static DeepPtr<Carta> generaCarta();
    // Helper per caricare una partita e giocare le carte
      virtual void setCartaSulTavolo(const DeepPtr<Carta>& carta);
      virtual void aggiungiCartaAllaMano(const DeepPtr<Carta>& carta, int
      idGiocatore) const;
      virtual void aggiungiGiocatore();

    // Eccezioni
      class turno_giocatore_non_controllato_dalla_cpu {};
      class non_e_il_turno_del_giocatore {};
```

Inoltre, la classe contiene quattro campi dati privati:

```
      std::vector<Giocatore*> giocatori_;
      DeepPtr<Carta> cartaSulTavolo_;
      int turno_;
      int turniDaPassare_;
```

std::vector risulta più efficace della Lista dato che permette l'accesso casuale e in quanto il numero dei giocatori rimane invariato durante la partita.

2.2 View

Le classi della View in Qt sono derivate tutte da *QWidget* con l'eccezione di *CartaWidget* che eredita da *QPushButton* per permettere di usare il segnale *pressed()* per selezionare che carta giocare.

3 Descrizione del codice polimorfo

L'uso di codice polimorfo e conseguente sfruttamento di RTTI (ovvero determinare il tipo dell'oggetto durante l'esecuzione invece che solo al momento della compilazione) avviene su più istanze.
Per esempio, si evidenziano di seguito delle casistiche di uso di codice polimorfo nel programma:

Il metodo virtuale ***virtual bool valida(const Carta& c) const*** permette di verificare la compatibilità fra carte, in particolare, cioè, di controllare che una carta sia giocabile sopra un'altra.
Usando il dynamic binding, viene chiamato il metodo corretto in base al tipo dinamico del riferimento o puntatore a *Carta*, il suo corpo sfrutta RTTI e in particolare il dynamic cast per
controllare se una carta possiede un colore (ovvero è sottotipo di *CartaColore*) oppure un numero prima di invocare i getter corrispondenti per effettuare il confronto.

La struttura dati che gestisce la carte in mano a ogni giocatore è una Lista templatizzata contenente *DeepPtr* per la gestione in memoria profonda di puntatori di tipo *Carta**.

Per la gestione di memoria profonda si fa uso dei metodi con opportuno overriding ***virtual Carta* clone() const*** con covarianza sul tipo di ritorno.
Il campo *cartaSulTavolo* della classe *Partita* sfrutta *DeepPtr* templatizzati istanziati al tipo *Carta**.

Nel metodo ***void applicaCartaSulTavolo()*** della classe *Partita* si usa RTTI tramite dynamic cast per controllare se il tipo dinamico della carta è un sottotipo di *CartaEffetto:* in questo caso quindi si chiede di invocare il metodo ***void applicaEffetto (Partita* partita)*** per eseguire l'azione opportuna chiamando il metodo tramite dynamic binding.

Nel metodo della classe *Partita* ***void eseguiStepGiocatoreCPU()*** si usa dynamic cast per controllare che il giocatore a cui spetta il turno sia di tipo *GiocatoreCPU* o un suo eventuale sottotipo prima di chiamare il metodo ***stepMossa().***

Nella classe Controller, al fine di fornire le corrette informazioni necessarie alle classi della View, si utilizzano molto spesso metodi di controllo sul tipo dinamico di un parametro, tramite sia typeid che dynamic cast.
I metodi che utilizzano tali controlli sono:
- ***char cartaInManoChar(int idGiocatore, int idCarta) const***
- ***int cartaInManoColore(int idGiocatore, int idCarta) const***
- ***char cartaSulTavoloChar() const***
- ***int cartaSulTavoloColore() const***

Inoltre, tutti i distruttori delle classi polimorfe sono resi virtuali per evitare memory leak.

4. Formato file

Il formato per le partite salvate è XML.
Gli elementi del file sono i seguenti:

- Un elemento partita che funge da radice contenente un elemento tavolo e elementi giocatore in numero variabile
- Un elemento seed contente il seed attuale per il random
- Un elemento tavolo contente un elemento tipo e un elemento colore
- Un elemento giocatore per ogni giocatore nella partita contente un elemento carta per ogni carta nella sua mano
- Un elemento tipo per ogni tavolo o carta contenente un char che indica la carta, da 1 a 9 per le carte numero e le lettere S, D, Q per le carte effetto
- Un elemento colore per ogni tavolo o carta contenente un char che indica il colore, da 0 a 3 per rosso, giallo, blu e verde. Le carte nere sono indicate dal char 'N'

5. Manuale Utente

Quando si avvia il programma si può decidere di iniziare una nuova partita *UGNO,* impostando il numero di giocatori avversari comandati da computer e il numero di carte iniziali.
Nel caso si voglia riprendere una partita, interrotta in una sessione precedente, si può caricare il file corrispondente dal Menù File.

Sempre dal Menù File è possibile anche salvare, quindi, una partita che si vuole interrompere e iniziarne una nuova.

V'è inoltre un Menù Aiuto dedito, appunto, al supporto dell'utente, con un manuale delle regole di gioco.

6. Istruzioni per l'avvio del programma

Per compilare il programma si richiede l'uso del file `progetto.pro`, fornito nella cartella con tutti i file necessari.

Al fine della **compilazione** quindi, si invocano da terminale i comandi:

```
qmake; make
```

Previa corretta installazione di QT nella macchina. In caso non sia presente nel sistema si può installare dal sito proprietario, anche utilizzando un software Open Source, o tramite comando:

```
sudo apt-get install qt5-default.
```

Al fine dell'**esecuzione** del progetto, una volta compilato, si invoca da terminale il comando:

```
./progetto
```

7. Dettagli sul Lavoro (Su macchina MacOS Big Sur; compilatore clang 12.0.0 e Qt 5.15.2)

Lo sviluppo del progetto *Ugno* ha richiesto un totale di circa 50 ore di lavoro individuale.
Nel dettaglio, sono risultate necessarie le seguenti tempistiche ad ogni fine:

- *Analisi preliminare del problema*: 5 ore
- *Progettazione modello*: 3 ore
- *Progettazione GUI*: 4 ore
- *Apprendimento libreria Qt*: 2 ore (più ore extra tramite tutorati)
- *Codifica modello*: 15 ore
- *Codifica GUI*: 16 ore
- *Debugging*: 3 ore
- *Testing*: 3 ore

Sono stati visionati anche i tutorati al fine di migliorare e consolidare l'approfondimento circa la libreria QT e il software Qt Creator.

Le classi sono state realizzate come segue:

Da: **Giovanni Cocco** (Matricola 1223856).

- Lista
- DeepPtr
- Carta
- CartaEffetto
- CartaPescaDue
- CartaPescaQuattro
- GiocatoreCPU
- Controller
- CartaWidget
- MainMenuWidget
- PartitaWidget

Da: **Matteo Noro** (Matricola 1229145)

- Partita
- CartaColore
- CartaNumero
- CartaSaltaTurno
- NuovaPartitaWidget
- GiocatoreCPUWidget
- Giocatore
- CreditiWidget
- RegoleWidget