Relazione progetto

Giovanni Cocco
Matricola: *1223856*

Matteo Noro
Matricola: *1229145*

Indice

1 Scopo del progetto

Il progetto si prefigge lo scopo di creare un gioco di carte per il tempo libero che permetta a un giocatore di giocare contro un numero arbitrario di avversari controllati dal computer.

Il gioco prende forte ispirazione dal famoso gioco *UNO* tuttavia presenta delle differenze che lo rendono più semplice e accessibile.

Il numero di giocatori e carte iniziali è configurabile dall'utente alla creazione della partita. Inoltre è possibile salvare e caricare le partite in formato XML.

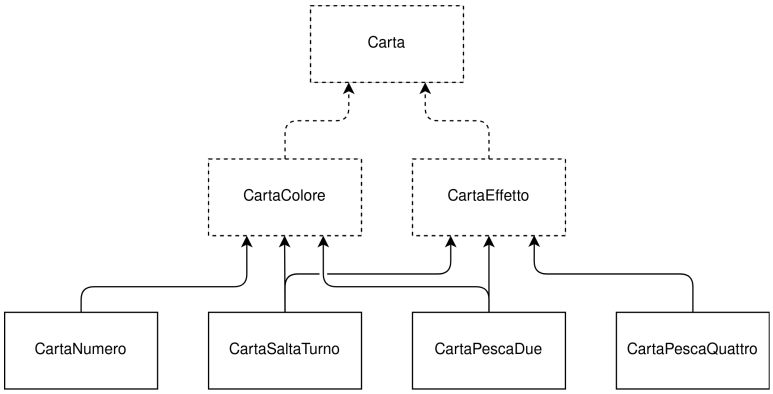
Ad ogni turno il giocatore può gettare sul tavolo quante carte vuole purché la carta sia compatibile con l'ultima gettata (all'inizio della partita è già presente una carta random sul tavolo).
Se un giocatore non getta carte in un turno alla fine di esso pesca automaticamente una carta.
Se un giocatore finisce tutte le carte nella sua mano ha vinto.

Le carte si dividono in 2 tipi principali: le carte numero e quelle effetto (salta turno, pesca due e pesca quattro) che applicano l'effetto quando vengono gettate.

1.1 Elenco carte

- **Carte numero**
  - simbolo nelle schermata di gioco: *numero tra 1 e 9*
  - possiedono un numero (1-9) e un colore(rosso, giallo, blu o verde)
  - sono compatibili con ogni carta abbia lo stesso colore oppure lo stesso numero
- **Salta turno**
  - simbolo nelle schermata di gioco: *S*
  - possiedono un colore(rosso, giallo, blu o verde)
  - sono compatibili con ogni carta salta turno oppure ogni carta abbia lo stesso colore
  - effetto: *il giocatore successivo salta il turno, se giocata più volte nello stesso turno l'effetto si propaga ai giocatori successivi*
- **Pesca due**
  - simbolo nelle schermata di gioco: *D*
  - possiedono un colore(rosso, giallo, blu o verde)
  - sono compatibili con ogni carta pesca due oppure ogni carta abbia lo stesso colore
  - effetto: *il giocatore successivo pesca 2 carte*
- **Pesca quattro**
  - simbolo nelle schermata di gioco: *D*.
  - possiedono il colore nero che vale come jolly
  - sono compatibili con ogni carta ed ogni carta è compatibile con esse
  - effetto: *il giocatore successivo pesca 4 carte*

2 Descrizione delle gerarchie



2.1 Carta

La classe **Carta** è la base astratta virtuale della gerarchia, ha un distruttore virtuale per supportare il polimorfismo e fornisce i metodi virtuali puri *Carta* clone() const* per il supporto alla gestione profonda della memoria tramite clonazione polimorfa e *bool valida(const Carta& c) const* per controllare se la carta è giocabile sopra un'altra carta data.

La classe **CartaColore** è una classe astratta che aggiunge il concetto di colore alla base con il campo dati proprio *colore_*, un getter pubblico e un'eccezione *colore_fuori_range* che viene invocata nel caso il colore passato al costruttore a un parametro non sia valido. Il costruttore di default inizializza la carta con un colore random.

La classe **CartaEffetto** è una classe astratta con un metodo virtuale puro *void applicaEffetto(Partita* partita) const* che dato un puntatore a partita applica l'effetto della carta.

La classe **CartaNumero** eredita da CartaColore e aggiunge il campo proprio *numero_*; implementa tutti i metodi richiesti per essere istanziabile e un'eccezione *numero_fuori_range* con la medesima logica del colore. Possiede anch'essa 2 costruttori, uno che permette di specificare il numero e il colore ed uno che li genera entrambi random.

La classe **CartaSaltaTurno** eredita da CartaColore e CartaEffetto; implementa tutti i metodi richiesti per essere istanziabile.

La classe **CartaPescaDue** eredita da CartaColore e CartaEffetto; implementa tutti i metodi richiesti per essere istanziabile.

La classe **CartaPescaQuattro** eredita da CartaColore; implementa tutti i metodi richiesti per essere istanziabile.

2.2 Giocatore

La classe **Giocatore** è una base concreta polimorfa che possiede la classe derivata **GiocatoreCPU** che espande la classe con il supporto all'esecuzione delle mosse gestite dal computer.

2.3 Partita

La classe **Partita** è predisposta al polimorfismo per permettere l'espansione futura l'applicativo al supporto di altri tipi di gioco o varianti dello stesso.

2.4 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

3.1 Distruttori polimorfi

I **distruttori** di tutte le classi polimorfe sono resi virtuali per evitare memory leak.

3.2 Metodo polimorfo valida

Il metodo virtuale **bool valida(const Carta& c) const** pertmette di controllare la possibilità di giocare una carta 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.

3.3 Struttura dati e gestione memoria profonda

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 della base della gerarchia ovvero di tipo **Carta***.

Per la gestione di memoria profonda fa uso dei metodi **Carta* clone() const** con covarianza sul tipo di ritorno.

Il campo *cartaSulTavolo_* della classe *Partita* sfrutta *DeepPtr* templatizzati istanziati al tipo **Carta***.

3.4 Metodo polimorfo applicaEffetto

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** e in caso invocare il metodo **void applicaEffetto(Partita* partita)** per eseguire l'azione opportuna chiamando il metodo tramite dynamic binding.

3.5 Controlli di tipo

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()* in caso non sia così viene sollevata un'eccezione.

La classe **Controller** fa ampio uso di controlli sul tipo dinamico sia **typeid** che **dynamic_cast** per fornire le informazioni sul tipo di carta e sul suo colore alle classi della *View* nei metodi:

- **char cartaInManoChar(int idGiocatore, int idCarta) const**
- **int cartaInManoColore(int idGiocatore, int idCarta) const**
- **char cartaSulTavoloChar() const**
- **int cartaSulTavoloColore() const**

4 Formato file

Il formato per le partite salvate è XML con una struttura ad albero.

4.1 Dettaglio elementi XML

- 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

5.1 Menù Principale

All'avvio viene presentato il menù principale da cui è possibile iniziare una nuova partita impostando il numero di giocatore e di carte iniziali oppure caricare una partita precedentemente salvata.

5.2 Menù File

Attraverso il menù File è possibile iniziare una nuova partita, salvare la partita corrente o caricare una partita precedentemente salvata.

5.3 Menù Aiuto

Attraverso il menù Aiuto è possibile accedere a un manuale dettagliato delle regole di gioco.

6 Istruzioni di compilazione

6.1 Compilazione

Per compilare il progetto è necessario usare il file *progetto.pro* fornito invocando i comandi:

qmake; **make**

Se Qt non risultasse installato nella VM è necessario installarlo con il comando: **sudo apt-get install qt5-default**.

6.2 Esecuzione

Per eseguire il progetto compilato invocare il comando **./progetto**.

7 Ore di lavoro richieste

Il progetto ha richiesto un totale di 48 ore di lavoro individuale.

7.1 Dettaglio ore

- Analisi preliminare del problema: 5 ore
- Progettazione modello: 2 ore
- Progettazione GUI: 3 ore
- Apprendimento libreria Qt: 2 ore
- Codifica modello: 15 ore
- Codifica GUI: 15 ore
- Debugging: 3 ore
- Testing: 3 ore

8 Suddivisione del lavoro progettuale

Classi realizzate dai membri del gruppo:

8.1 Giovanni Cocco

Matricola: *1223856*

- Lista
- DeepPtr
- Carta
- CartaEffetto
- CartaPescaDue

- CartaPescaQuattro
- GiocatoreCPU
- Controller
- CartaWidget
- MainMenuWidget
- PartitaWidget

8.2 Matteo Noro

Matricola: *1229145*

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

9 Ambiente di sviluppo

- Sistema operativo: *Manjaro Nibia 20.2*
- Compilatore C++: *g++ (GCC) 10.2.0*
- Qt: *version 5.15.2*

9.1 Dettagli extra

- Desktop enviroment: *XFCE 4.14*
- Qt style: *kvantum*
- Qt palette: *airy*
- Qt font: *Cantarell 10*
- QMake: *version 3.1*
- Make: *GNU Make 4.3*