Lazy Card Game

LIMONGELLI MARCO ANDREA

Matricola: 1225415

Progetto del corso Programmazione a Oggetti (A.A. 2020-2021)

1 Prefazione

1.1 Scopo del progetto

Lo scopo del progetto è quello di realizzare un'applicazione nel linguaggio C++ abbinandolo al framework QT per l'implementazione dell'interfaccia grafica.

1.2 Strumenti utilizzati

- Sistema operativo di sviluppo: Windows 10 Home
- IDE: QT Creator versione 4.13.3
- Linguaggio: C++ 11
- Compilatore: GNU g++ 7.3

1.3 Fasi del progetto e la loro durata

- 8 ore per l'analisi totale del problema e per la realizzazione del diagramma delle classi
- 2 ore per la progettazione della GUI
- 1 ora per la progettazione e scrittura dell'algoritmo della strategia dei giocatori della CPU
- 8 ore per l'apprendimento della libreria QT
- 16 ore per la codifica del modello
- 14 ore per la codifica della GUI
- 6 ore per il debugging ed il testing

Il totale del tempo impiegato è quindi di 55 ore. La strategia del giocatore controllato dalla CPU è stata realizzata come ultima cosa, infatti è abbastanza basilare e non troppo complessa. Mi sarebbe piaciuto renderla ancora più intelligente ma avevo già sfondato il tetto massimo di ore, quindi l'ho mantenuta semplice. Le ore in eccesso rispetto alle 50 previste sono dovute al debug e anche alla progettazione durante la quale ho avuto più volte cambiamenti di idee (sia sulle regole del gioco, sia sul come implementare le funzionalità) che mi hanno portato a modificare il diagramma.

1.4 Metodologia di lavoro e pattern MVC

La specifica del progetto richiedeva espressamente la *massima separazione tra parte logica e grafica (GUI) del codice*. Seguendo questo principio, ho deciso di utilizzare il pattern MVC (*Model-View-Controller*). Questo pattern consiste appunto nel separare la logica di presentazione dei dati (*View*) dalla logica di business (*Model*). Il *Controller* è l'elemento che lega queste due parti. Prima di tutto, per essere certo di non essere dipendente dalla *View* nel *Model*, ho deciso di sviluppare la logica di business del programma. Questo mi ha consentito di poter avere un programma funzionante e utilizzabile al 100% da linea di comando. La realizzazione della GUI è avvenuta in un secondo momento solamente per presentare i dati e ricevere gli input dell'utente.

Per quanto riguarda la gestione delle versioni dei sorgenti, ho utilizzato un repository GIT.

2 Lazy Card Game

2.1 Regole del gioco

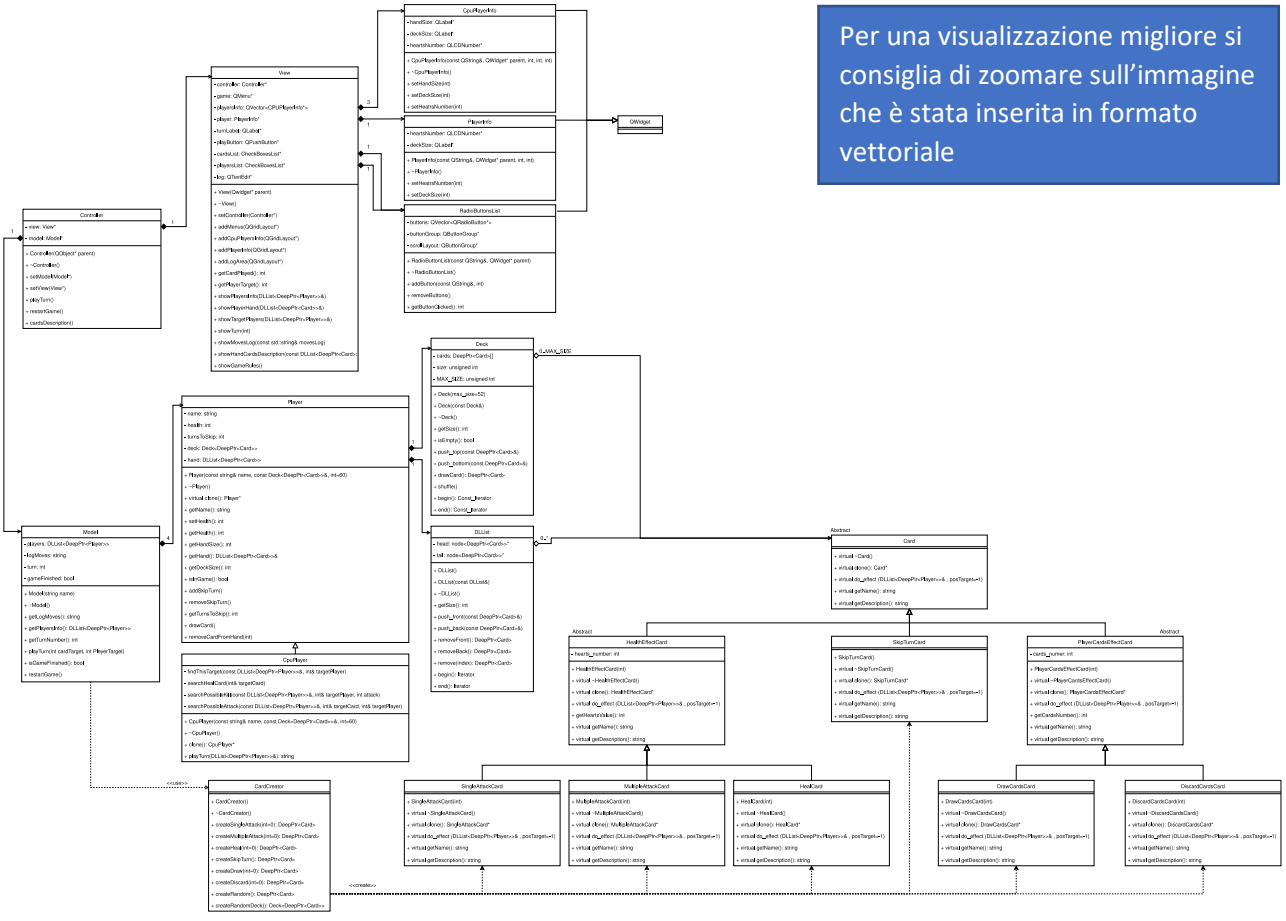
Lazy Card Game è un gioco di carte di mia invenzione. Ogni giocatore ha un mazzo di 52 carte (che contiene le stesse carte per ciascun giocatore ma mescolate in maniera differente) e una vita inizialmente impostata a 30. Il primo turno ogni giocatore pesca 5 carte. Le carte, che possono essere utilizzate una sola per turno, fungono da modificatore dei giocatori. Sono disponibili i seguenti tipi di carte:

- ***Pesca x***: Il giocatore selezionato come bersaglio pesca un numero di carte pari a x dal proprio mazzo.
- ***Scarta x***: Al giocatore selezionato come bersaglio vengono scartate le prime x carte dal proprio mazzo.
- ***Salta turno***: Il giocatore selezionato come bersaglio salta un turno. Se è stata già giocata una carta *salta turno* su uno stesso giocatore, quest' ultimo dovrà saltare tanti turni quante sono le carte *salta turno* giocate su di esso.
- ***Cura x***: Al giocatore selezionato vengono aggiunti x punti vita.
- ***Attacco multiplo x:*** Vengono sottratti x punti vita a *tutti* i giocatori (incluso chi gioca la carta).
- ***Attacco singolo x:*** Al giocatore selezionato come bersaglio vengono rimossi x punti vita.

Alla fine del proprio turno il giocatore pesca una carta dal mazzo. Se un giocatore non ha più carte nel proprio mazzo, per ogni carta che tenterà di pescare gli verranno sottratti 3 punti vita.

Lo scopo del gioco è azzerare la vita di tutti gli altri giocatori.

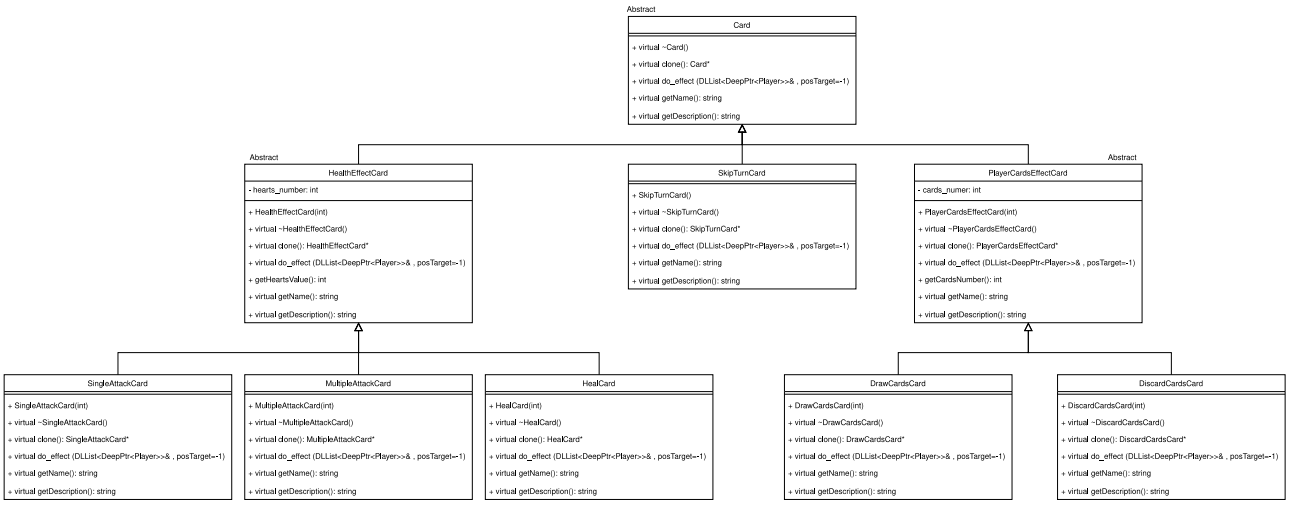
3 Diagramma delle classi



Per una visualizzazione migliore si consiglia di zoomare sull'immagine che è stata inserita in formato vettoriale

3.1 Gerarchia del modello

3.1.1 Carte



Al vertice della gerarchia troviamo la classe astratta *Card*. La classe *Card*, oltre al *distruttore* polimorfo e al metodo polimorfo *clone* (richiesto dalla classe *DeepPtr*), presenta tre metodi polimorfi puri:

- ***do_effect(DLList<DeepPtr<Player>>& players, int posTarget)***
  Questo metodo è super-polimorfo in quanto ciascun diverso tipo di carta lo andrà a definire in modo totalmente diverso da ciascun altro tipo di carta. Come parametro viene passata la lista dei giocatori per riferimento e un intero che indica l'indice del giocatore sul quale giocare la carta.

- ***getName(): std::string***
  Questo metodo restituisce semplicemente il nome della carta in base al suo tipo.

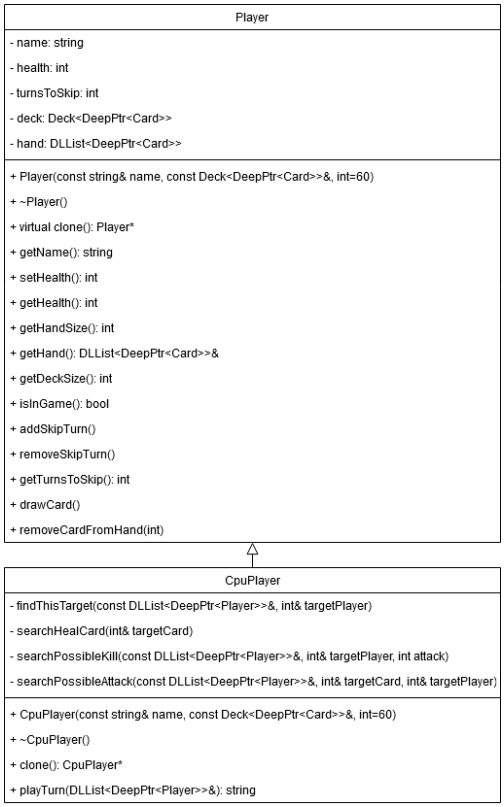
- ***getDescription(): std::string***
  Questo metodo restituisce semplicemente l'effetto provocato da tale carta e ovviamente varia in base al suo tipo.

All'interno della gerarchia si possono notare altre due classi astratte, derivate direttamente dalla base *Card*. Queste due classi sono *HealthEffectCard* e *PlayerCardsEffectCard*. Ho deciso di realizzare queste due classi per raggruppare sotto di esse altre carte con effetti della stessa tipologia che hanno caratteristiche comuni.
In *HealthEffectCard* è presente un intero che indica di quanti cuori modificare la vita di un giocatore. Infatti le sue sottoclassi sono tutte relative alla salute dei giocatori, ovvero *SingleAttackCard*, *MultipleAttackCard* e *HealCard*.
In *PlayerCardsEffectCard* è presente un intero che indica il numero di carte che i suoi sottotipi devono scartare (*DiscardCardsCard)* o pescare (*DrawCardsCard)*.
Per come è strutturata la gerarchia ed il programma, è facile inserire nuove carte nella gerarchia. Per esempio, pensando ad una possibile estensione, potremmo fare alcune aggiunte:

- ***EveryoneDrawsCard***: Carta per far pescare *tutti*. Sottotipo di *PlayerCardsEffectCard*
- ***DiscardFromHand***: Carta per far scartare un certo numero di carte dalla mano. Sottotipo di *PlayerCardsEffectCard*
- ***AttackAndHealCard***: Carta per sferrare un attacco singolo e curare il giocatore che sferra l'attacco di un certo quantitativo di vita. Sottotipo di *SingleAttackCard*

Per facilitare la creazione delle carte, ho aggiunto una classe *CardCreator* che è praticamente l'implementazione del *FactoryPattern*. Infatti questa classe viene utilizzata soprattutto per i metodi *createRandom()* che create una nuova carta di tipo casuale e per *createRandomDeck()* che restituisce in output un mazzo di carte casuali.

3.1.2 Giocatori

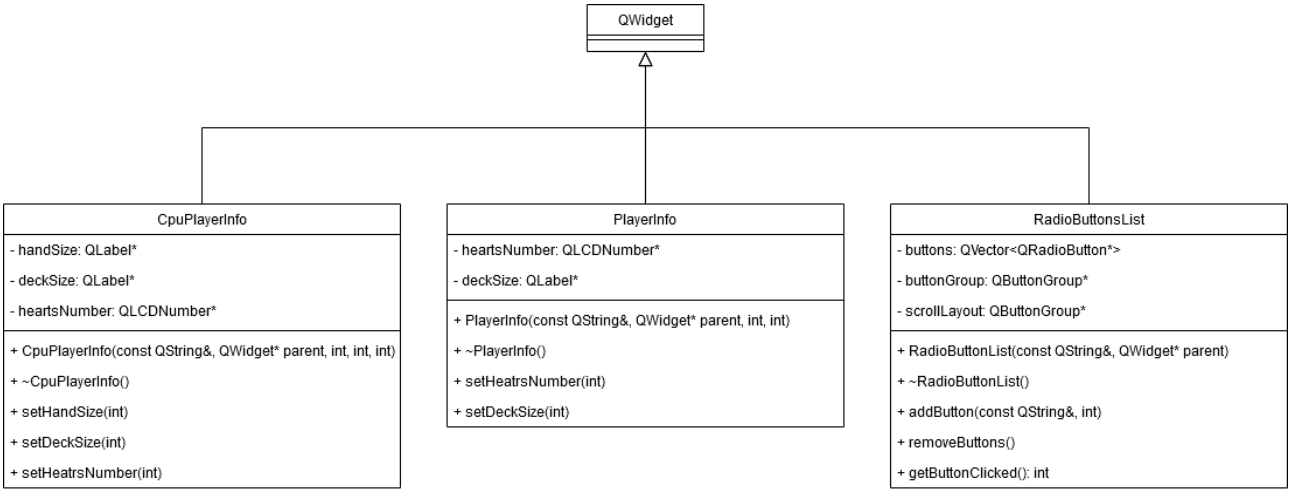


Anche per quanto riguarda i giocatori è presente una minima gerarchia. La classe base *Player* contiene tutti i metodi *get* e *set* per la gestione di tutte le sue proprietà.
La classe derivata *CpuPlayer* è un'estensione della classe *Player* in quanto aggiunge il metodo *playTurn(DLList<DeepPtr<Player>>&)* che è la funzione di mossa automatica. La strategia dei giocatori controllati dalla *CPU* è molto semplice ed è formata dai seguenti passi:

1) Prima di tutto controlla se ha turni da saltare. In caso affermativo, salta il turno e quindi non prosegue nella strategia.
2) Controlla se si deve curare: nel caso in cui la sua vita si minore di 10 cerca se ha una *carta cura* nella mano e la gioca su sé stesso.
3) Se non si deve curare, cerca se con qualche tipo di *carta attacco* (*SingleAttackCard* o *MultipleAttackCard*) può azzerare la vita di un altro giocatore.
4) Se non può curarsi e non può uccidere un altro giocatore, gioca casualmente una carta con l'accortezza di non scegliersi come bersaglio nel caso di *attacchi singoli* e di *scarta carte*. Nel caso in cui scelga casualmente una *carta cura*, il bersaglio sarà sé stesso.

Ovviamente va giocata una carta per turno, quindi si va al passo successivo solo se non si è potuta giocare nessuna carta nel precedente.

3.2 Gerarchia View



La GUI, che verrà descritta più dettagliatamente nel paragrafo 5, è composta da 4 tipi di widget, di cui 3 sono stati definiti da me. Pertanto è presente nel diagramma delle classi anche questa gerarchia. Costruire Widget personalizzati mi ha permesso di gestire in modo più ordinato la presentazione dei dati e la gestione dei vari *radio button*. Tutti i widget creati da me derivano dalla classe base *QWidget*:

1) ***CpuPlayerInfo***:
   Widget che serve per visualizzare le informazioni relative ai giocatori della CPU. Mostra la vita, il numero di carte presenti nel mazzo e il numero di carte presenti nella mano del giocatore. Fornisce tutti i metodi necessari a settare tali informazioni.

2) ***PlayerInfo***:

Widget che serve a visualizzare le informazioni relative al giocatore controllato dall'utente dell'applicazione. Mostra il numero di carte ancora presenti nel mazzo e la vita del giocatore. Fornisce tutti i metodi necessari a settare tali informazioni.

3) *RadioButtonsList*:

Widget che viene usato per mostrare una lista di *radio button*. Viene utilizzato sia per far visualizzare la lista dei giocatori fra i quali scegliere il bersaglio, sia per visualizzare la lista delle carte attualmente in mano fra le quali scegliere la carta da giocare. La classe prevede un metodo *addButton(QString&, int)* che permette di inserire un nuovo *radio button* nella lista, fornendo come parametri una *Qstring* da associargli e un *intero* che corrisponde all' *ID* del bottone. Questa classe ha inoltre un metodo *getButtonClicked()* che ritorna l'*ID* del bottone selezionato. Gli *ID* sono assegnati in modo tale che corrispondano alla posizione delle carte e dei giocatori all'interno dei loro contenitori: per esempio se in posizione 0 della mano del giocatore c'è un *Attacco Singolo 5*, l'*ID* del bottone associato a tale carta sarà il valore 0.

4 I contenitori

I contenitori erano un elemento centrale della specifica e dovevano rispettare due semplici vincoli:

1) Devono essere template di classe, ovvero essere in grado di immagazzinare oggetti di tipo parametrico T
2) Devono fornire gli opportuni iteratori per visitare tutti gli elementi contenuti al loro interno

Per la realizzazione della mia applicazione mi sono reso conto di aver bisogno di due tipi di contenitori diversi: una classe *Deck* e una classe *DLList.*

4.1 Deck

Deck
- cards: DeepPtr<Card>[] - size: unsigned int - MAX_SIZE: unsigned int
+ Deck(max_size=52) + Deck(const Deck&) + ~Deck() + getSize(): int + isEmpty(): bool + push_top(const DeepPtr<Card>&) + push_bottom(const DeepPtr<Card>&) + drawCard(): DeepPtr<Card> + shuffle() + begin(): Const_Iterator + end(): Const_Iterator

La classe *Deck* viene utilizzata come contenitore di carte ed è infatti il contenitore designato per implementare il mazzo. Per la realizzazione di questo contenitore ho pensato alle funzionalità fondamentali che devono essere presenti in un mazzo:

- Funzione per mescolare il mazzo
- Funzione per pescare la carta in cima al mazzo

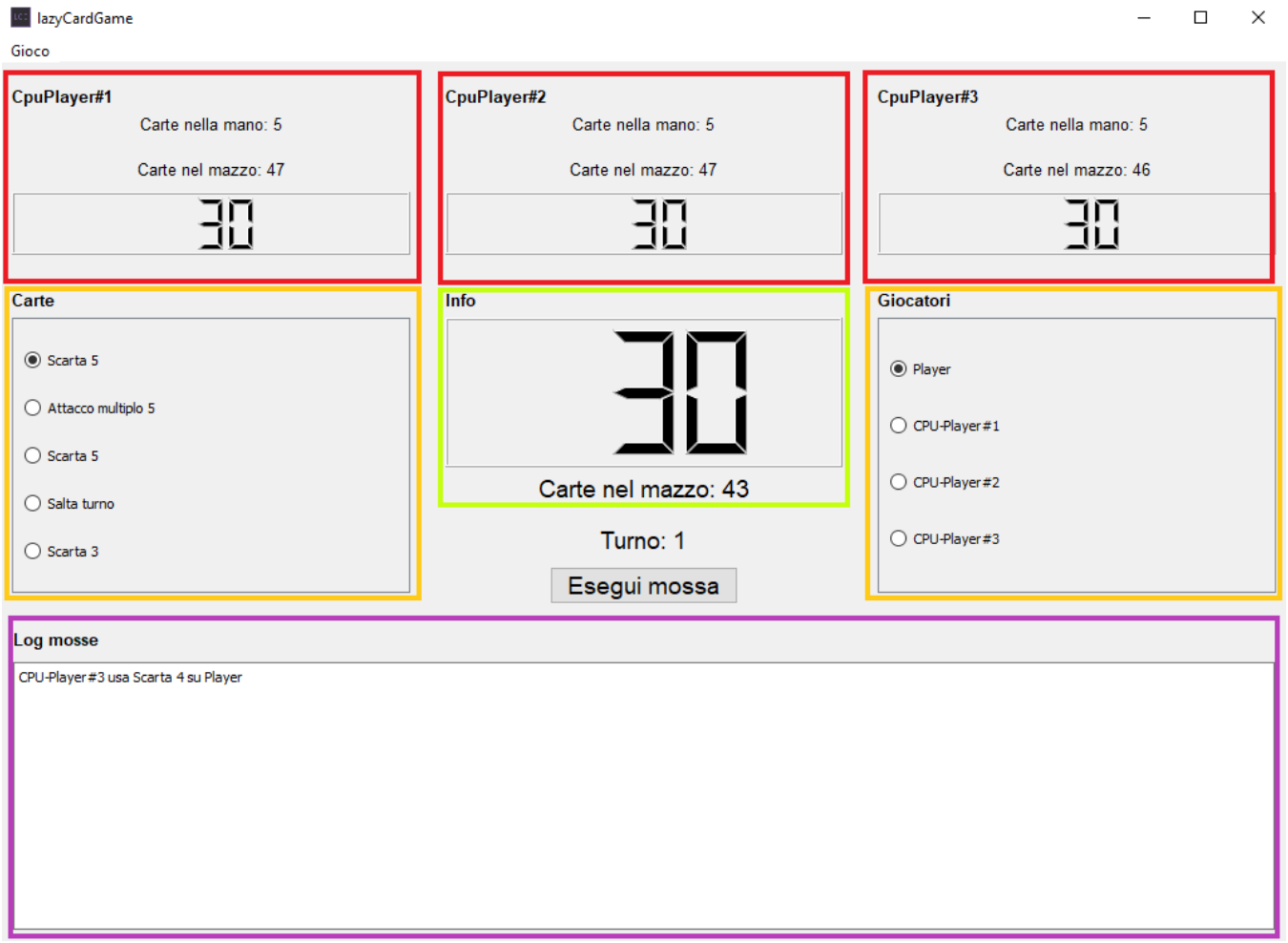
Utilizzare una lista mi avrebbe dato sicuramente dei vantaggi nel momento di pescare una carta dalla cima e anche nell'inserimento di una carta, però la funzione per mescolare sarebbe stata molto più complessa di un semplice swap di elementi di un vettore. Perciò, visto che una delle funzionalità principali è quella di mescolare il mazzo, la scelta implementativa è sfociata nell'utilizzo di un array. Nel costruttore è previsto un parametro che indica il massimo numero di carte che possono essere contenute nel mazzo. Per evitare di fare continuamente swap tra gli elementi dell'array, quando viene chiamata la funzione *drawCard()* viene dato in output l'elemento in posizione *size-1* dell'array, dove *size* indica il numero totale di carte attualmente contenute nel mazzo.

4.2 DLList

DLList
- head: node<DeepPtr<Card>>*
- tail: node<DeepPtr<Card>>*
+ DLList()
+ DLList(const DLList&)
+ ~DLList()
+ getSize(): int
+ push_front(const DeepPtr<Card>&)
+ push_back(const DeepPtr<Card>&)
+ removeFront(): DeepPtr<Card>
+ removeBack(): DeepPtr<Card>
+ remove(index): DeepPtr<Card>
+ begin(): Iterator
+ end(): Iterator

La classe *DLList* implementa una lista doppiamente concatenata, ovvero ogni nodo è collegato con il precedente e con il successivo. Questo tipo di contenitore è stato utilizzato per immagazzinare i giocatori e per l'implementazione delle mani dei giocatori. Inizialmente ho provato a ragionare sull'utilizzo dei *Deck* per realizzare le mani dei giocatori ma sarebbe risultato completamente inefficiente in quanto un giocatore rimuove le carte dalla propria mano in una qualsiasi posizione in maniera imprevedibile. Per questo motivo la rimozione di un elemento casuale in un array avrebbe comportato ad una continua ricompattazione di quest' ultimo. Una lista invece è molto adatta a risolvere problemi come questo in quanto è necessario modificare solo due *link* per scollegare un nodo dalla lista ed eliminarlo.

5. GUI e manuale



La GUI è composta da 4 widget principali, disposti in un *GridLayout*, che nell'immagine soprastante sono incorniciati con ognuno un colore diverso:

- ***CpuPlayerInfo***:
  Widget segnati in rosso. Servono a fornire le informazioni sugli altri giocatori. Ogni *Widget* ha una *QLabel* che indica il giocatore a cui fanno riferimento tali dati.

- ***RadioButtonList***:
  Widget segnati in arancione. Servono ad ottenere l'input sulla carta da giocare e su quale giocatore usarla. L'utente dovrà selezionare una carta dalla lista delle carte ed un giocatore dalla lista dei giocatori.

- ***PlayerInfo***:
  Widget segnato in verde. Fornisce le informazioni sulla vita ed il numero di carte ancora presenti nel mazzo dell'utente.

- ***Log***:
  Widget che serve a mostrare all'utente tutte le mosse che sono state effettuate durante la partita. Guardando cosa c'è scritto nel log il giocatore riesce a comprendere i cambiamenti sulle informazioni di tutti i giocatori. Le prime righe del Log contengono le ultime mosse che sono state effettuate prima dell'inizio del turno del giocatore.

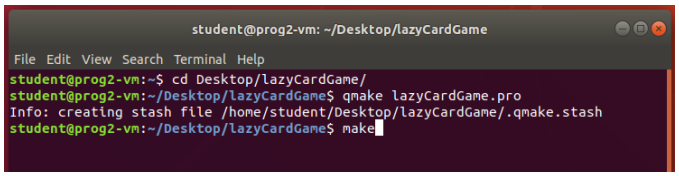
Oltre a questi *widget* si può notare che la GUI ha il bottone *Esegui mossa*, il quale dovrà essere premuto dall'utente dopo aver selezionato giocatore e carta da giocare per eseguire la propria mossa. Inoltre è presente un *menu gioco* che contiene tre bottoni:

- ***Nuova partita***
  Permette di cominciare una nuova partita, sia che la partita attuale sia finita, sia se semplicemente si vuole cominciarne una nuova abbandonando la vecchia.

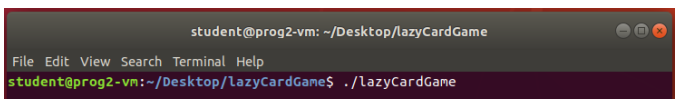
- ***Effetto carte***
  Apre un *QDialog* che mostra all'utente l'effetto delle carte che ha in mano.

- ***Regole***
  Apre un *QDialog* che mostra all'utente le regole del gioco.

6 Materiale consegnato e compilazione



Per la compilazione è necessario il file *.pro* che viene fornito nel file *.zip* consegnato, in quanto la generazione automatica del file *.pro* tramite il comando *qmake -project*, non permette una corretta compilazione utilizzando il comando *make*.



Sulla sinistra sono mostrati i comandi utilizzati per la compilazione ed esecuzione sulla macchina di test (macchina virtuale Linux).

Vengono inoltre consegnati:

- Tutti i file *.cpp*
- Tutti i file *.h*
- File *LCG.ico* per l'icona dell'applicazione
- Relazione in formato *.pdf*