Programmazione ad Oggetti
- PROGETTO -

AnimalGame

di Bizzaro Francesco

Scopo del progetto

Si vuole realizzare una versione desktop di un nuovo gioco di carte chiamato "*Animal*". In particolare si dovrà creare un'interfaccia intuitiva attraverso la quale un giocatore potrà sfidare un'intelligenza artificiale, da fornire in una prima versione basilare, ma perfezionabile in futuro. Le regole del gioco *Animal* sono specificate di seguito. Dato che il gioco si basa su delle carte, è richiesto un metodo per ricavare da file i dati delle stesse.

La struttura base del gioco *Animal* prende spunto dalla catena alimentare presente in natura fra le varie specie animali. In particolare la maggior parte delle carte rappresenta proprio degli animali, e ognuna di esse sarà in grado di "mangiare" altre carte, facendo dei punti. Al termine del gioco, vincerà il giocatore con il numero maggiore di punti.
Nel gioco sono presenti 4 tipologie di carte: le carte "animale", "animale con effetto", "magia" e "arma". Tutte le carte hanno un nome, un numero identificativo e un tipo. Il tipo serve a categorizzarle e verrà specificato meglio in seguito.
Le carte animale[1] sono caratterizzate da una resistenza, un attacco e dei nemici. Resistenza e attacco sono due valori numerici, mentre i nemici sono un insieme di tipi. I tipi in tutto sono 11: "magia" per le carte magia, "arma" per le carte arma, e "carnivoro", "uccello", "insetto", "pesce", "pianta", "erbivoro", "rettile", "onnivoro" o "insettivoro" per le carte animale e animale con effetto. Questi valori servono a stabilire se una carta animale ne può mangiare un'altra: per poterlo fare, la prima deve avere un tipo tra quelli presenti fra i nemici della seconda, e un attacco maggiore o uguale alla resistenza di questa. Se queste condizioni non sono entrambe soddisfatte, la prima carta non potrà mangiare la seconda. Quando un giocatore tirerà una carta sul campo di gioco, questa dovrà mangiare forzatamente tutte le carte che potrà.
Le carte animale con effetto sono un sottoinsieme delle carte animale e pertanto ne posseggono tutte le caratteristiche. In aggiunta hanno un "effetto", ossia un particolare comportamento che, una volta che la carta è stata tirata, può far variare il normale andamento del gioco. È richiesta l'implementazione solo di alcuni effetti basilari, ma si chiede che possano essere espansi facilmente. Gli effetti da realizzare sono: "Elimina tutte le carte dell'avversario in campo" e "Tutte le carte di tipo pesce in campo diventano tuoi punti".
Le carte magia sono carte che hanno solo un effetto. Non possono mangiare altre carte e non possono essere mangiate. Si chiede di realizzare almeno le seguenti: "I giocatori si scambiano le carte che hanno in mano", "Spia le carte in mano del tuo avversario" e "Copia una carta presente sul campo, che ti verrà in mano".
Infine le carte arma sono carte che hanno solo un attacco e possono mangiare tutte le carte (animale) con una resistenza minore o uguale ad esso. Quando mangiano una carta tuttavia non portano punti, ma si limitano a toglierla dal campo. Queste carte non possono essere mangiate da altre.
Il gioco è diviso in giri, composti da 4 turni. Ad ogni turno un giocatore potrà tirare una carta. Appena tira la carta, questa mangia le carte in campo che può mangiare e/o si comporta secondo il suo effetto. A questo punto si avanza di un turno e tira l'altro giocatore. I giocatori si alternano fino a tirare 2 volte ciascuno e a questo punto finisce il giro. Alla fine di ogni giro – e solo a questo

1 Allegata immagine nella pagina seguente.

punto – le carte che mangiano qualche altra carta vengono tolte dal campo, e per ciascuna di esse viene assegnato un punto al giocatore che l'aveva tirata, per ogni carta mangiata dalla stessa. Vengono tolte dal campo anche le carte magia e arma presenti, ma senza che i giocatori ricevano punti. Si aspetta la fine del giro a togliere le carte per permettere che una carta mangiante possa essere a sua volta mangiata e/o subire effetti.
Finito un giro si prosegue con il successivo, dove a tirare per primo (il primo turno) sarà il giocatore che al giro precedente aveva fatto più punti, o chi aveva iniziato prima in caso di parità. Il gioco termina con la fine del 12° giro. A questo punto il vincitore sarà il giocatore con più punti.

Rappresentazione di una **carta animale**. L'icona in alto a destra è una possibile rappresentazione per il tipo “rettile”. Resistenza e attacco della carta sono riportati sotto la figura. Infine i nemici sono mostrati subito sotto, come elenco di tipi (in questo caso i nemici hanno tipo uccello, rettile e insettivoro).

Prima di iniziare la partita ad ogni giocatore viene assegnato un mazzo di 40 carte, creato casualmente con i dati delle carte forniti tramite file. Durante il gioco ognuno potrà tenere in mano fino a 5 carte. Si pesca solamente all'inizio di ogni giro, in modo da averne esattamente 5, o nel caso venissero esaurite.

Ambiente di sviluppo
L'applicazione è stata sviluppata sul sistema operativo Linux Mint 17 (Xfce 64bit versione kernel 3.13.0-24-generic GNU/Linux). Il compilatore usato è g++ nella versione 4.8.2. La libreria Qt installata è nella versione 5.2.1 (64-bit).

Progettazione
Si è cercato di aderire al pattern Model-View-Controller. Per la parte di modellazione sono state individuate 3 gerarchie di classi: una avente come base la classe astratta *Carta*, una la classe astratta *Effetto* e la terza la classe *Giocatore*. Queste classi dovranno incapsulare le informazioni e i metodi necessari per la gestione dei concetti del gioco che il loro stesso nome suggerisce. Anche una classe *Campo* sarà utile per mantenere le informazioni sulle carte giocate.
Il controller sarà la classe *Partita*, la quale avrà come campi dati dei riferimenti ai giocatori e al campo, e si occuperà di far procedere il gioco turno dopo turno. Quando avverranno degli avvenimenti durante la partita, questa classe emetterà opportuni segnali che andranno ad aggiornare la grafica.
La parte grafica sarà mantenuta il più possibile separata dalle classi precedenti, e prenderà le informazioni necessarie interrogando *Partita* e *Carta* (dato che ogni carta dovrà essere resa graficamente secondo le proprie regole, si è scelto di inserire nell'interfaccia di *Carta* un metodo virtuale che ne ritorni la rappresentazione).

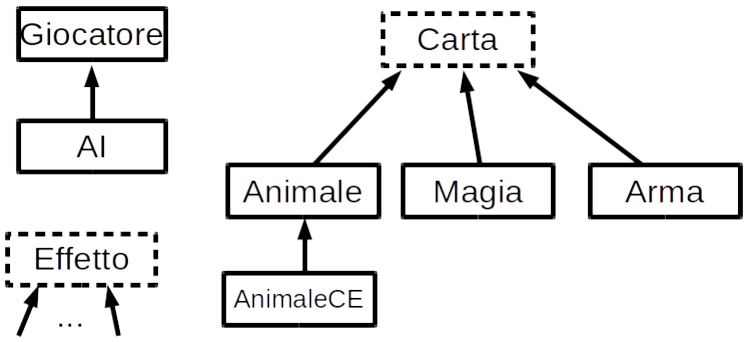
Descrizione delle gerarchie
Dall'analisi dei requisiti si distinguono 4 tipologie differenti di carte (animale, animale con effetto, magia e arma). Queste però hanno anche degli aspetti in comune. Si è pertanto deciso di creare una gerarchia di classi, per fornire una stessa interfaccia alle future implementazioni specifiche. Alla base di questa gerarchia vi è una *Carta* astratta, che non potrà essere concretizzata ad oggetto, ma che sarà utile a creare riferimenti e puntatori polimorfi.
Questa classe racchiude nella parte privata le informazioni che tutte le carte comunque devono avere, ossia un intero positivo *id*, il *nome* della carta (usato QString) e il *tipo* della carta. Per

rappresentare il concetto di *tipo* è stato scelto di usare un'enumerazione che associa ad ogni *tipo* un numero primo. Questo dettaglio implementativo è pensato per facilitare la verifica che una carta animale possa mangiarne un'altra, e verrà discusso in seguito. Tornando alla classe *Carta*, essa fornisce i seguenti metodi pubblici:

- Un costruttore a tre parametri con valori di default, utile a costruire sotto-oggetti dato che la classe è astratta. Il costruttore di copia è lasciato in versione standard.
- Il distruttore è quello standard ma è marcato virtuale in quanto, se chiamato su di un puntatore polimorfo che punta ad un'istanza di una classe discendente, deve chiamare il distruttore del tipo dinamico dell'oggetto puntato, per non lasciare garbage.
- Un operatore di uguaglianza (e disuguaglianza) NON virtuale che paragona (a meno di ridefinizioni) due oggetti discendenti da Carta sulla base del solo attributo id (gli altri campi potrebbero variare in relazione ad effetti, mentre l'id identifica una carta).
- Vari metodi di get per accedere al nome, id e tipo
- Un metodo virtuale puro bool mangia(const Carta&) che dovrà essere implementato nelle classi derivate e restituire true se e solo se l'oggetto su cui viene invocato può mangiare l'oggetto compatibile con Carta che viene passato per parametro.
- Un metodo virtuale puro void onTira(Partita*,Giocatore*) che dovrà essere implementato nelle classi derivate in modo da richiamare eventuali procedure che devono svolgersi nel momento in cui la carta viene tirata (pensato per invocare gli effetti delle carte magia e animale con effetto).
- Un metodo virtuale puro QWidget* getPicture() che ogni classe derivata dovrà implementare per restituire la rappresentazione grafica della carta (tipica della classe).
- Un metodo virtuale puro Carta* clona() che, implementato nelle classi discendenti, dovrà restituire un puntatore polimorfo di tipo Carta*, ad un oggetto che è la copia profonda del parametro di invocazione.

Discendente diretta di *Carta* è la classe *Animale*, che ha come campi dati (privati) i valori di attacco, resistenza e nemici. Per le regole del gioco spiegate nei requisiti, una carta animale per mangiare una carta deve avere il tipo presente fra i nemici di quest'ultima (e l'attacco maggiore o uguale alla resistenza). Si è scelto di rappresentare i tipi come numeri primi, e i nemici – che devono essere un insieme di tipi – come prodotto di tali numeri. In questo modo per verificare che un tipo sia presente fra i nemici di una carta, basterà vedere il resto della divisione intera nemici su tipo. In questo modo si evita l'uso di contenitori a basso livello. La classe Animale eredita da Carta l'operatore di uguaglianza e i metodi per ricavare nome, id e tipo. Implementa invece i seguenti metodi:

- bool haNemico(Tipo), che attraverso la divisione discussa in precedenza restituisce true se e solo se l'oggetto di invocazione ha fra i nemici il tipo passato per parametro.
- void aggiungiNemico(Tipo) che va invece ad aggiungere un nemico all'oggetto istanziato.
- Il metodo virtuale mangia(const Carta&) dichiarato in Carta viene qui implementato per restituire true se e solo se l'oggetto di invocazione può mangiare la carta polimorfa passata per parametro: se la carta parametro è un animale, ha tra i nemici il tipo di questa e ha la resistenza <= dell'attacco di questa. In questo metodo viene usato un dynamic_cast per verificare che la carta polimorfa sia un Animale (o animale con effetto).
- Il metodo onTira viene implementato per rendere la classe istanziabile, ma di fatto è vuoto.
- Il metodo virtuale getPicture() crea un oggetto PicCarta (sottotipo di QWidget) e ne ritorna un puntatore. Questo rispetta la segnatura di Carta::getPicture() in quanto modifica solo il tipo di ritorno, che è qui specializzato. PicCarta è una classe derivata da QWidget che rappresenta una “immagine cliccabile di carta” ed emette un segnale quando è cliccata. Questo segnale sarà poi usato per richiamare il metodo di *Partita* delegato a tirare in campo una carta.



Rappresentazione delle gerarchie discusse nel capitolo

Ad un livello inferiore della gerarchia è stata creata la classe *AnimaleCE*, sottotipo diretto di *Animale* che rappresenta le carte animale con effetto. Ha un solo attributo proprio che è un puntatore ad *Effetto*. *Effetto* è un'altra classe astratta che fa da interfaccia per le implementazioni degli effetti concreti (gli effetti richiesti nei requisiti sono stati incapsulati in classi inserite in una seconda gerarchia, come figlie di *Effetto*). *AnimaleCE* eredita tutti i metodi di *Animale*, ma ne ridefinisce alcuni. I metodi principali sono:

- Un distruttore virtuale che dealloca dalla memoria il puntatore ad *Effetto.*
- Una ridefinizione del costruttore di copia e dell'assegnazione per fare copia profonda dell'effetto.
- Un metodo per assegnare un effetto all'oggetto.
- La ridefinizione del metodo virtuale onTira(Partita*,Giocatore*) per chiamare il metodo proprio dell'effetto che causa l'alterazione del normale svolgimento del gioco, secondo quanto previsto dal particolare oggetto sottotipo di *Effetto*. Questo è reso possibile grazie al metodo virtuale puro Effetto::onTira(Partita*,Giocatore*) che deve essere implementato sulla particolare istanza di *Effetto*.
- La ridefinizione del metodo getPicture() inserisce nell'immagine la descrizione dell'effetto.

In seguito saranno forniti maggiori dettagli sulla classe *Effetto*. Ora prosegue la trattazione della gerarchia delle carte. La classe *Magia* rappresenta le carte che secondo i requisiti devono possedere un effetto, non possono mangiare e non sono mangiate da altre carte. Questa classe, come *AnimaleCE*, ha un putatore ad *Effetto*, ma è figlia diretta di *Carta*. I principali metodi sono:

- Distruttore che dealloca l'oggetto puntato dal campo dati *Effetto**.
- Ridefinizioni del costruttore di copia e dell'operatore di assegnazione per fare copia profonda dell'effetto.
- Implementazione del metodo virtuale bool mangia(const Carta&) che ritorna sempre false.
- Implementazione del metodo virtuale onTira(Partita*,Giocatore*), che come per *AnimaleCE* va a chiamare il metodo onTira sull'oggetto puntato dal puntatore ad *Effetto* posseduto.
- Implementazione del metodo virtuale getPicture che restituisce anche in questo caso una *PicCarta* cliccabile, con un design grafico caratteristico della classe.

Per concludere la descrizione della gerarchia delle carte resta la classe Arma, che deve rappresentare carte con un attacco, in grado di mangiare qualsiasi carta animale avente una resistenza <= di esso (indipendentemente dai nemici), ma che non portano punti al giocatore. Tralasciando per il momento la gestione dei punti, questa classe ridefinisce il metodo virtuale booleano mangia(const Carta&) con la propria regola peculiare. Entrando nei particolari viene eseguito un dynamic_cast sul parametro *Carta*, per verificare se si tratti o meno di un *Animale* (o sottotipo), ed in caso affermativo viene valutata la diseguaglianza. Il metodo virtuale onTira viene implementato vuoto come per Animale, in quanto non sono previsti comportamenti particolari al momento del tiro. GetPicture() ritornerà anche in questo caso un puntatore a PicCarta per poter rilevare i click sulla carta.

La classe *Effetto*, come si è detto, fa da interfaccia per le implementazioni "concrete" degli effetti descritti nei requisiti. È una classe astratta da cui queste dovranno ereditare per essere portate all'esecuzione attraverso binding dinamico su puntatori polimorfi. *Effetto* ha un metodo per ricavare una descrizione testuale e un distruttore virtuale, ma sopratutto un metodo virtuale puro onTira(Partita*,Giocatore*) che è il cuore di questa struttura. I 5 effetti richiesti nei requisiti (2 per animali con effetto e 3 per magie) sono stati resi all'interno delle implementazioni di onTira su classi discendenti di *Effetto* create ad hoc. Se in futuro si volesse creare un nuovo effetto basterà creare una nuova classe che discenda dalla classe *Effetto* e dia un'implementazione di onTira (è possibile eventualmente memorizzare anche uno stato interno e/o creare metodi aggiuntivi).

Dalla fase di progettazione è emersa l'utilità di una terza – più piccola – gerarchia di classi, per fornire una struttura al concetto di *Giocatore*, che può essere sia l'umano che interagisce con la GUI, sia l'intelligenza artificiale. L'intelligenza artificiale, rappresentata dalla classe *Ai*, sarà un sottotipo di *Giocatore*. Ogni *Giocatore* ha come attributi privati un nome, un contatore dei punti e due contenitori di *Carte* polimorfe: uno per il mazzo e uno per le carte che il giocatore ha in mano (che per i requisiti devono essere 5). Il tipo dei contenitori usati è Collettore<Carta*>, che è di fatto una lista[2] con doppio link, che garantisce buone prestazioni in caso di frequenti inserimenti e cancellazioni che mantengono l'ordine degli oggetti precedentemente memorizzati.
Giocatore offre nella parte pubblica:

- Un distruttore virtuale che rende la classe polimorfa
- Metodi per inserire, togliere o estrarre carte dal mazzo e dalla mano, e per conoscere il numero di carte che vi sono.
- Un metodo per pescare le carte, che trasferisce carte dal mazzo alla mano, seguendo la regola richiesta di mantenere in mano 5 carte.
- Un metodo per ottenere in lettura una lista delle carte in mano
- Metodi per conoscere e aumentare/diminuire il numero di punti del giocatore
- Un metodo virtuale agisci(Partita*) che verrà invocato da *Partita* all'inizio di ogni turno in cui dovrà tirare il giocatore in questione. In questa versione il metodo verifica che il giocatore abbia delle carte, e lo fa pescare in caso negativo. Questo metodo verrà ridefinito in *Ai*, per farlo tirare automaticamente.
- Una slot tira(unsigned int) che è in grado di ricevere il segnale con l'id di una carta, lanciato da una *PicCarta* quando viene cliccata. Se l'id ricevuto corrisponde all'id di una *Carta* che è presente in mano, allora viene invocato il metodo di *Partita* per mettere in campo la carta.

La classe *Ai* deve essere in grado di scegliere da sola la *Carta* da tirare con un proprio algoritmo interno, anziché attendere attraverso un segnale che sia l'utente a farlo. Per questo motivo viene ridefinito il metodo virtuale agisci, che in questo caso si occupa anche di scegliere e tirare direttamente in campo una carta (attraverso il puntatore a *Partita* passato per parametro). Per la scelta della carta si avvale del metodo costante protetto trovaCartaDaTirare(Partita*), che in questa semplice versione di gioco seleziona la carta che in quel momento riesce a mangiare di più. Per fornire una intelligenza artificiale migliore, basterà in futuro ridefinire questo metodo o estendere la classe (che è polimorfa) e fare un override. Per questo motivo il metodo è protetto e non privato.

Descrizione del controller *Partita* e di come usa le classi polimorfe

Partita è la classe principale dell'applicazione, che si occupa di mandare avanti il gioco nelle sue fasi. Essendo logicamente scorretto che vi siano più istanze di *Partita* in un medesimo istante, si è cercato di renderla una sorta di singleton. Nello specifico sono stati resi privati i costruttori (anche di copia), e sono invece stati forniti dei metodi statici per creare, ottenere o cancellare l'unica istanza, mantenuta attraverso un puntatore statico privato.

2 È un template di lista con doppio link. Viene usato anche nella classe *Campo*, istanziato ad un tipo diverso.

Prima di entrare nello specifico di questa classe, si descriverà brevemente la classe *Campo*, i cui servizi verranno usati in *Partita*. Questa classe rappresenta il campo da gioco, su cui si accumulano le carte tirate, prima di essere eventualmente tolte alla fine di un giro o per altri motivi. È a tutti gli effetti una classe contenitore. Inoltre, proprio come ha senso che vi sia una sola istanza di *Partita*, anche per il *Campo* sarebbe errato se vi fossero molteplici oggetti. Anche *Campo* è stato reso un singleton mediante campi e metodi statici e rendendo privati i costruttori. Gli oggetti che *Campo* deve contenere sono delle *Carte* che sono state tirate. Serve tenere traccia anche del Giocatore che le ha tirate ed eventualmente del numero di carte che al momento del tiro sono state mangiate (per poi calcolare il numero di punti guadagnati a fine giro). Pertanto è stato definito un nuovo tipo *CartaGiocata* all'interno della classe *Campo*, che memorizza un puntatore polimorfo a *Carta*, uno a *Giocatore* e un intero per il numero di carte mangiate.
L'attributo principale della classe *Campo* è il contenitore di questi oggetti *CartaGiocata*. È stata usata nuovamente un'istanza del template *Collettore*[3], questa volta a *CartaGiocata*. Si è scelta una lista anziché un vector per non alterare l'ordine delle carte a seguito di una eliminazione: la scelta è discutibile, ma in questo modo si ritiene di aver garantito una maggiore usabilità verso gli utenti. Infatti prendendo la decisione opposta le carte in campo si sarebbero potenzialmente mescolate nell'ordine durante il succedersi dei vari giri del gioco.
Campo offre metodi di inserimento, estrazione e rimozione di *CartaGiocata*, e di scorrimento attraverso iteratori.
Tornando a *Partita*, essa mantiene nella sua parte privata due puntatori a *Giocatore* (uno per l'utente e uno per l'intelligenza artificiale), un puntatore a *Campo* e vari interi per gestire i giri, i turni e l'attribuzione della facoltà di tirare fra i giocatori. I metodi principali che la sua interfaccia offre sono i seguenti:

- Vari metodi per reperire informazioni sullo stato del gioco, ad esempio getGiro e getTurno restituiscono i numeri di giro e turno, toccaPC e terminata sono metodi booleani che verificano se deve tirare Ai e se la partita è giunta al termine, getCampo e getMano permettono di scorrere attraverso iteratori sulle carte in campo e nella mano del giocatore umano; getAi e getGiocatore forniscono puntatori ai giocatori della partita.
- Il metodo avanzaTurno, che si occupa di far avanzare la partita di un turno. Se il turno al momento dell'invocazione era l'ultimo del giro, calcola i punti guadagnati nel giro e li assegna ai giocatori, dopodiché assegna un ordine di tiro per il giro, pulisce il campo dalle carte che non servono più, incrementa il giro e fa pescare i giocatori. Se invece il turno non era l'ultimo del giro si limita ad incrementarlo e scambiare il giocatore attivo. In entrambi i casi, dopo aver fatto le operazioni precedenti, controlla se la partita sia finita e, in caso negativo, invoca sul giocatore cui tocca tirare il metodo polimorfo agisci. Questo metodo invocato sul puntatore a tipo dinamico *Giocatore* non fa nulla se non controllare che il numero di carte in mano sia 5, mentre invocato sul puntatore a tipo dinamico *Ai* va a richiamare il metodo di *Ai* che sceglie una carta e la tira. Questa chiamata polimorfa di metodo automatizza il tiro dell'intelligenza artificiale.
- Il metodo mettiInCampo(Carta*,Giocatore*), che deve gestire la fase di tiro: verificare che un giocatore possa tirare una carta, applicare gli eventuali effetti di questa, far mangiare il più possibile questa carta ed infine metterla in campo e passare il controllo ad avanzaTurno. Subito dopo il controllo che il giocatore possa effettivamente tirare in questo momento (inserito per sicurezza dato che il tiro di un “giocatore utente” avviene tramite segnali), viene invocato il metodo virtuale onTira sulla *Carta* passata per parametro. Questa è una chiamata polimorfa, che potrebbe non fare nulla nel caso il tipo dinamico della *Carta* sia *Animale* o *Arma*, ma che va invece ad effettuare una seconda chiamata polimorfa su un *Effetto* nel caso il tipo dinamico sia *Magia* o *AnimaleCE*. Questa seconda chiamata di metodo richiama il codice incapsulato nel tipo dinamico del puntatore a *Effetto* posseduto da

3 La descrizione di questo contenitore è oggetto del capitolo successivo.

queste carte. Con questo meccanismo basta una sola chiamata polimorfa per richiamare l'esecuzione degli effetti delle *Carte*, qualsiasi sia il tipo dinamico di queste e dell'E*ffetto*. In seguito mettiInCampo scorre le carte in *Campo*, e invoca per ognuna di esse il metodo polimorfo mangia – sempre sulla carta passata per parametro – per far mangiare alla carta il più possibile prima di metterla in *Campo*. Il metodo mangia è stato implementato su ogni sottotipo di *Carta*, ed è pertanto in grado di determinare indipendentemente dal tipo dinamico se una *Carta* ne possa mangiare una seconda, con la giusta regola. Anche qui una sola chiamata polimorfa risparmia l'analisi di un certo numero di combinazioni *Carta-Carta* e rende nel contempo il codice aperto a future modifiche/aggiunte. Dopo aver calcolato cosa il parametro *Carta* sia in grado di mangiare (e aver tolto dal *Campo* le carte mangiate), questo viene messo in *Campo* e viene infine invocato il metodo avanzaTurno per far proseguire il gioco.

- Il metodo assegnaCarte, che genera (pseudo)casualmente i mazzi dei *Giocatori* e li fa pescare una prima volta. Per prima cosa questo metodo apre un file xml dove sono contenuti i dati delle carte, rispettando in questo modo il requisito di avere i dati salvati su file esterno (per facilitare l'inserimento di nuove carte). A questo punto i dati vengono convertiti e trattati in modo da ottenere una lista di oggetti polimorfi *Carta*, del corretto tipo dinamico. Per convertire una *Carta* da xml a binary si fa uso anche di due funzioni globali: traduciTipo (dichiarata in carta.h) e traduciEffetto (dichiarata in effetto.h). La prima funzione trasforma la rappresentazione testuale di un tipo nel corrispondente numero primo, mentre la seconda a partire dal nome di un effetto e la sua descrizione, costruisce un oggetto avente tipo statico *Effetto*, e dinamico quello particolare per l'effetto (sottotipo creato ad hoc). Purtroppo quest'ultima funzione non permette la massima estensibilità del codice, in quanto per ogni nuovo sottotipo di *Effetto* che viene creato, si dovrebbe aggiungere uno statement. Per migliorare tale funzione sarebbero utili funzionalità di reflection. Ad ogni modo, una volta creato l'insieme delle *Carte* disponibili, vengono creati i mazzi facendo copie profonde di queste attraverso il metodo polimorfo clona.

Per concludere si fa presente che, in alcuni dei metodi precedenti, *Partita* emette dei segnali coi quali trasmette informazioni alla parte di gestione della grafica e/o ne richiede l'aggiornamento.

Breve descrizione del contenitore templatizzato *Collettore*

Collettore, come già annunciato, viene utilizzato per contenere le carte nel mazzo e in mano dei giocatori e per gestire il *Campo*. È una lista con doppio link, per cercare di rendere efficienti inserimenti e rimozioni e mantenere l'ordine delle carte. L'accesso agli elementi sarà per scorrimento e non casuale, e ciò è in accordo con l'uso che di questo contenitore viene fatto. Per rendere lo scorrimento agevole viene fornito un iteratore.

Collettore gestisce la memoria in maniera condivisa: non vengono mai effettuate copie profonde degli elementi contenuti. Tuttavia è di fatto impossibile creare una copia del contenitore, al quale ci si potrà solamente riferire (o puntare), dato che il costruttore di copia e l'assegnazione sono stati resi privati. Questa decisione è stata guidata dall'uso che si vuol fare di questo contenitore: né il *Campo*, né i mazzi, né le carte in mano dei giocatori dovranno essere copiati in un secondo contenitore, ma solamente acceduti attraverso iteratori e riferimenti. Pertanto non vi potranno essere problemi causati dalla condivisione della memoria.

Collettore offre metodi di inserimento in testa e prima di un iteratore, di rimozione in corrispondenza di un iteratore o di un particolare oggetto contenuto, di estrazione di un elemento, di ricerca di un elemento e di clear.

Gestione della grafica

L'interfaccia grafica è costituita per la maggior parte dalla classe *MainWindow*[4], sottotipo di QMainWindow. Unico metodo pubblico di questa classe (oltre al costruttore) è newGame, che è in grado di catturare un segnale lanciato dal menù di gioco (per segnalare l'intenzione di iniziare una nuova partita), e agire di conseguenza. Nel caso vi fosse già una partita aperta, questo metodo la chiude cancellando dalla memoria tutti i campi dati. Successivamente ne avvia una invocando il metodo statico per costruire una *Partita*, e su questa istanza chiama il metodo assegnaCarte. A questo punto sono stati creati gli oggetti *Giocatore*, e sono completi di mazzo ecc. Il gioco può cominciare e viene invocato il metodo privato mostraPartita, che stampa a video le carte in mano al "giocatore umano" e le carte in campo. Per ogni carta del giocatore viene creata una connessione che permette, in caso di click del mouse, di andare a chiamare – attraverso un segnale – il metodo di *Giocatore* per tirare la carta.

Ogni volta che nella *Partita* succedono certi eventi (ad esempio il tiro di una carta, l'incremento del turno..) vengono emessi dei segnali che informano su cosa sia avvenuto. Tutti questi segnali sono catturati da slot private di *MainWindow*, che portano ad aggiornare la finestra. È presente anche un piccolo box dove vengono inseriti messaggi informativi su quanto sta accadendo nel gioco.

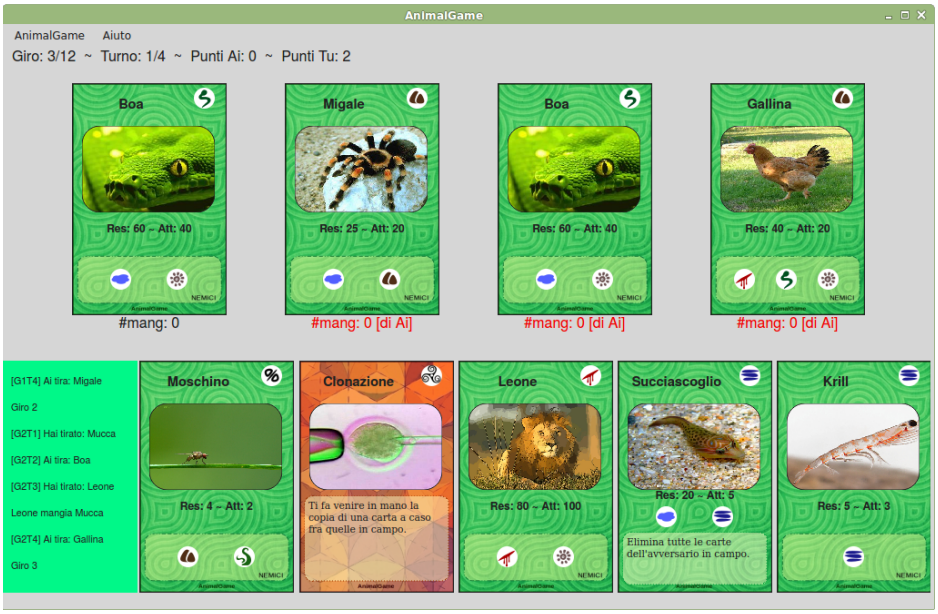
Tutte le immagini delle carte sono ottenute tramite la chiamata polimorfa al metodo getPicture di *Carta*, che ne ritorna la rappresentazione caratteristica in base al tipo dinamico.

Per aumentare il grado di usabilità verso l'utente, è stato inserito in *Partita* un metodo che crea una finestra di dialogo informativa mostrante la carta tirata da *Ai*. Questa finestra si chiude da sola dopo circa 2 secondi.

Manuale utente

Quando un utente esegue l'applicazione si troverà una finestra vuota con un menù. Tramite il menù potrà aprire una finestra di dialogo che riassume le regole del gioco (esplicitate dai requisiti del progetto), o iniziare una partita. Per iniziare una partita è richiesto un nickname da assegnare al *Giocatore* (se si omette ne verrà assegnato uno di default). Durante il gioco basterà cliccare con il mouse sulla carta che si intende tirare durante il proprio turno. L'intelligenza artificiale farà da sola le proprie mosse.

Per la compilazione basterà eseguire da terminale il comando qmake e successivamente make. Accertarsi che la cartella delle Risorse sia presente nella stessa cartella del file eseguibile.



Una schermata del gioco a partita iniziata

4 Vi sono in realtà anche alcune finestre di dialogo che vengono create durante l'esecuzione.