Università degli Studi di Bergamo

SCUOLA DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

Progetto C++: The Game

Studente
Davide Salvetti
Matricola 1057596

Anno Accademico 2021–2022

1 Introduzione

Il programma sviluppato consiste in un gioco a turni in cui ci sono due giocatori che si sfidano. Il gioco è composto da una mappa di grandezza variabile (8x8, 12x12 o 16x16) dove sono posizionati dei personaggi. Ogni giocatore possiede un certo numero di personaggi e l'obbiettivo è quello di distruggere tutti quelli dell'avversario. Inoltre, è possibile creare nuovi personaggi utilizzando i castelli: ogni giocatore, infatti, possiede un castello, e con il passare dei turni i castelli guadagnano stelle. Si possono poi utilizzare le stelle per creare personaggi da schierare nella battaglia.
Il progetto è stato sviluppato utilizzando il framework **Qt** e le librerie grafiche **Qt Quick**.

2 Funzionamento

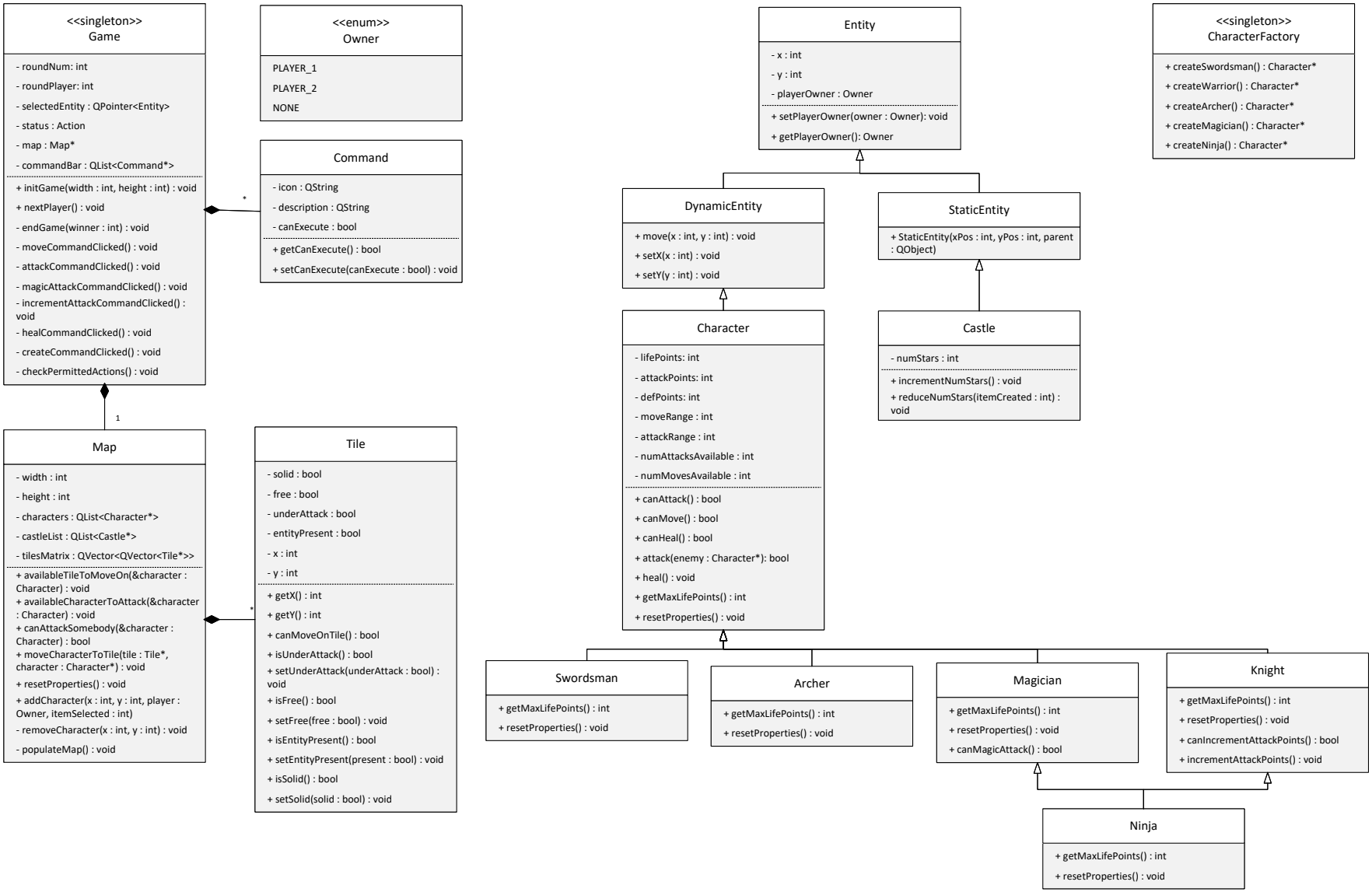
Ad ogni turno, il giocatore può eseguire diverse mosse. Per ogni suo personaggio può:

- muovere il personaggio;
- attaccare un nemico;
- recuperare punti vita;
- se il personaggio è un mago, eseguire un incantesimo;
- se il personaggio è un cavaliere, aumentare i suoi punti di attacco.

Inoltre, può creare una nuova unità nel castello.

3 Diagramma delle classi

Di seguito viene riportato il Diagramma delle classi. Nelle classi mostrate sono stati inseriti solo i metodi ed i membri più significativi.


<<singleton>>
Game
- roundNum: int
- roundPlayer: int
- selectedEntity : QPointer<Entity>
- status : Action
- map : Map*
- commandBar : QList<Command*>
+ initGame(width : int, height : int) : void
+ nextPlayer() : void
- endGame(winner : int) : void
- moveCommandClicked() : void
- attackCommandClicked() : void
- magicAttackCommandClicked() : void
- incrementAttackCommandClicked() : void
- healCommandClicked() : void
- createCommandClicked() : void
- checkPermittedActions() : void
<<enum>>
Owner
PLAYER_1
PLAYER_2
NONE
Command
- icon : QString
- description : QString
- canExecute : bool
+ getCanExecute() : bool
+ setCanExecute(canExecute : bool) : void
*
1
Map
- width : int
- height : int
- characters : QList<Character*>
- castleList : QList<Castle*>
- tilesMatrix : QVector<QVector<Tile*>>
+ availableTileToMoveOn(&character : Character) : void
+ availableCharacterToAttack(&character : Character) : void
+ canAttackSomebody(&character : Character) : bool
+ moveCharacterToTile(tile : Tile*, character : Character*) : void
+ resetProperties() : void
+ addCharacter(x : int, y : int, player : Owner, itemSelected : int)
- removeCharacter(x : int, y : int) : void
- populateMap() : void
Tile
- solid : bool
- free : bool
- underAttack : bool
- entityPresent : bool
- x : int
- y : int
+ getX() : int
+ getY() : int
+ canMoveOnTile() : bool
+ isUnderAttack() : bool
+ setUnderAttack(underAttack : bool) : void
+ isFree() : bool
+ setFree(free : bool) : void
+ isEntityPresent() : bool
+ setEntityPresent(present : bool) : void
+ isSolid() : bool
+ setSolid(solid : bool) : void
*
Entity
- x : int
- y : int
- playerOwner : Owner
+ setPlayerOwner(owner : Owner): void
+ getPlayerOwner(): Owner
<<singleton>>
CharacterFactory
+ createSwordsman() : Character*
+ createWarrior() : Character*
+ createArcher() : Character*
+ createMagician() : Character*
+ createNinja() : Character*
DynamicEntity
+ move(x : int, y : int) : void
+ setX(x : int) : void
+ setY(y : int) : void
StaticEntity
+ StaticEntity(xPos : int, yPos : int, parent : QObject)
Character
- lifePoints: int
- attackPoints: int
- defPoints: int
- moveRange : int
- attackRange : int
- numAttacksAvailable : int
- numMovesAvailable : int
+ canAttack() : bool
+ canMove() : bool
+ canHeal() : bool
+ attack(enemy : Character*): bool
+ heal() : void
+ getMaxLifePoints() : int
+ resetProperties() : void
Castle
- numStars : int
+ incrementNumStars() : void
+ reduceNumStars(itemCreated : int) : void
Swordsman
+ getMaxLifePoints() : int
+ resetProperties() : void
Archer
+ getMaxLifePoints() : int
+ resetProperties() : void
Magician
+ getMaxLifePoints() : int
+ resetProperties() : void
+ canMagicAttack() : bool
Knight
+ getMaxLifePoints() : int
+ resetProperties() : void
+ canIncrementAttackPoints() : bool
+ incrementAttackPoints() : void
Ninja
+ getMaxLifePoints() : int
+ resetProperties() : void

4 Classe Game

La classe `Game` è quella che gestisce la partita: inizializza la mappa, gestisce i turni e gestisce le mosse dei giocatori.

4.1 Singleton

Dal momento che è necessaria solo un istanza di questa classe in qualunque istante, si è deciso di implementarla come *Singleton*. Per fare ciò è stato definito il costruttore privato e un metodo pubblico statico che ritorna un'istanza di `Game`, che è anch'essa statica. Inoltre, per evitare che l'istanza venga copiata, sono stati marcati `delete` il *copy constructor* e il *copy assignment operator*: in questo modo, il compilatore non crea automaticamente questi metodi e quindi non c'è il rischio che la *Singleton* venga copiata.

```
class Game : public QObject
{
    ...
public:
    static Game& getInstance();
    ~Game();

    ...

    Game(Game const&) = delete;
    void operator=(Game const&) = delete;

    ...
};
```

4.2 Parent-Child relationship

In Qt è possibile utilizzare la relazione **"padre-figlio"**: quando un oggetto viene creato e allocato tramite il costrutto `new` è possibile passare al costruttore un oggetto che diventa suo **"padre"**. L'oggetto **"padre"** prende la proprietà dell'oggetto **"figlio"**. In questo modo, quando un oggetto "padre" viene distrutto, distruggerà automaticamente tutti i suoi oggetti "figli" all'interno del suo distruttore. Grazie a questo paradigma, la gestione della memoria risulta più semplice e si evitano *memory leak*.
Per poter applicare questo principio, è necessario che le classi siano sottotipi di `QObject`. `QObject` è la classe centrale di Qt: oltre a permettere la relazione "padre-figlio", rende possibile l'utilizzo del meccanismo di comunicazione chiamato *signals and slots*, tramite cui è possibile eseguire una funzione (*slot*) in seguito all'invio di un segnale (*signal*).
Per via della relazione "padre-figlio" non si è reso necessario utilizzare *smartpointers*: infatti, al momento della creazione alle classi è stato assegnato un padre che si occupa della sua distruzione. Le uniche classi a cui non viene assegnato un padre sono quelle che derivano da `Character`. In questo caso, è necessario avere il pieno controllo su quando queste classi vengono distrutte perchè devono essere eseguiti alcuni controlli in seguito alla loro distruzione, per esempio bisogna verificare se un giocatore non ha più personaggi a disposizione e quindi il gioco è terminato. Di conseguenza, la deallocazione di questi oggetti viene eseguita manualmente tramite il costrutto `delete`. Non vi sono comunque *memory-leak* perchè il distruttore della classe `Map` si occupa di deallocare tutti i personaggi rimasti in gioco.
Tuttavia, al posto dei *raw pointers*, in alcuni casi sono stati utilizzati i `QPointer` che è una classe di Qt che implementa un *guarded pointer*: quando l'oggetto puntato da *QPointer* viene distrutto, il puntatore viene automaticamente settato a `nullptr`. Questo evita che ci siano *dangling pointers*. I *guarded pointers* sono particolarmente utili quando è necessario mantenere un riferimento ad un oggetto la cui proprietà dipende da un'altra classe e quindi potrebbe essere distrutto mentre il riferimento è ancora attivo. Questo è il caso del membro `selectedEntity` nella classe `Game`: la proprietà non è della classe `Game`, è la classe `Map` che gestisce la distruzione dell'oggetto. Lo stesso principio non vale per il riferimento alla classe `Map` in `Game`, perchè la *ownership* è proprio della classe `Game`.

4.3 Qt Class Container e STL

Qt fornisce delle classi container che possono essere utilizzate in sostituzione o in parallelo alle classi container fornite dalla libreria STL. Le principali differenze riguardano i metodi implementati: i container di Qt hanno API compatibili con il mondo Qt e con il mondo STL e quindi i metodi a disposizione sono duplicati (per esempio abbiamo sia `count()` che `size()`). Dal momento che il progetto è stato sviluppato con l'utilizzo di Qt, si è scelto di utilizzare le classi container di Qt piuttosto che quelle disponibili in STL, ma lo stesso comportamento si sarebbe ottenuto semplicemente convertendo i container Qt in container STL. `QList<T>` è una delle classi container di Qt e fornisce delle funzionalità simili a `std::list`. Per esempio, grazie alle librerie di Qt è possibile creare degli *STL-Style iterators* come segue:

```
// STL-Style iterators used for learning purposes
QList<Character*> charactersList = map->getCharacters();
QList<Character*>::iterator enemy;
for (enemy = charactersList.begin(); enemy != charactersList.end(); ++enemy) {
    if ((*enemy)->getPlayerOwner() != selectedEntity->getPlayerOwner()) {
        if (!dynamic_cast<Magician*>(*enemy))
            (*enemy)->inflictDamage(2);
    }
}
```

Altre classi container di Qt sono `QVector<T>`, `QMap<Key, T>`, `QSet<T>` e `QHash<Key, T>`.

4.4 Overloading

Sulla mappa possono essere cliccati sia quadranti (*Tile*) che personaggi (*Character*). Di conseguenza, per definire la dinamica del gioco, è stato necessario fare *overloading* sul tipo dei parametri delle seguenti funzioni:

```
void onIdleState(Tile *tile);
void onMoveState(Tile *tile);
void onAttackState(Tile *tile);

void onIdleState(Character *character);
void onMoveState(Character *character);
void onAttackState(Character *character);
```

5 Classe Entity

La classe `Entity` è la classe base che rappresenta una qualunque entità nel gioco. Da questa classe ereditano `DynamicEntity` e `StaticEntity`: la prima implementa i metodi per muovere l'entità sulla mappa, mentre la seconda implementa un costruttore che necessita della posizione in modo da non poter più muovere l'entità una volta creata. La classe `Character` è sottotipo della classe `DynamicEntity` e di conseguenza è un entità che si può muovere sulla mappa. La classe `Castle`, invece, eredita da `StaticEntity` e di conseguenza non può muoversi. Dalla classe `Character` ereditano tutti i vari personaggi del gioco, come `Swordsman`, `Archer`, `Magician`, `Knight` e `Ninja`.

5.1 Classi Astratte

Le classi `Entity`, `DynamicEntity`, `StaticEntity` e `Character` sono classi astratte. Infatti, non è possibile istanziare una di queste classi nel gioco, ma è possibile utilizzarle solo come riferimento. Per fare ciò sono stati dichiarati dei metodi *pure virtual* che devono essere necessariamente implementati nelle classi derivate. Di seguito un esempio dei metodi *pure virtual* della classe `Character`:

```
void virtual resetProperties() = 0;
int virtual getMaxLifePoints() const = 0;
```

5.2 Ereditarietà multipla e problema del diamante

La classe `Ninja` è sottotipo sia di `Magician` che di `Knight`, sfruttando l'ereditarietà pubblica di entrambe le classi.

```
class Ninja : public Magician, public Knight
```

Ciò introduce il problema del diamante: la classe `Ninja` eredita da `Magician` e da `Knight`, i quali a loro volta ereditano da `Character`. In questo modo, `Ninja` contiene due istanze diverse di `Character`, una che deriva da `Character` e una che deriva da `Knight`. Per evitare questo problema è necessario dichiarare di tipo *virtual* l'ereditarietà delle classi `Magician` e `Knight` verso `Character`:

```
class Knight : virtual public Character
...
class Magician : virtual public Character
```

5.3 Overriding

La classe `Character` è la classe con più metodi e membri nella gerarchia delle entità. I metodi di tale classe sono sia *pure virtual* che non *virtual*. Per esempio:

- `resetProperties` è di tipo *pure virtual* perchè è necessario che ogni classe che eredita da `Character` implementi tale metodo, in quanto dipende proprio dal tipo dell'oggetto.
- i metodi *getters* e *setters* sono invece implementati direttamente nella classe `Character` perchè sono comuni a tutte le classi.

I metodi *virtual* vengono utilizzati invece nel diamante composto da `Knight` e `Magician`: queste due classi, infatti, implementano i metodi `resetProperties()` e `getMaxLifePoints()` (che sono *pure virtual* nella classe `Character`), ma anche la classe `Ninja` che eredita da queste due li implementa. A runtime verrà selezionato quale è il metodo corretto da eseguire. In particolare, per il metodo `resetProperties()` della classe `Ninja`, non c'è una ridefinizione esplicita ma viene richiamato il metodo della classe base `Knight`. Questo metodo è stato implementato anche nella classe `Ninja` per evitare problemi di *name clash*: infatti, se non fosse stato ridefinito, il compilatore non avrebbe saputo quale metodo chiamare, se quello di `Magician` o quello di `Knight`.

```
class Magician : virtual public Character {
    ...
    virtual int getMaxLifePoints() const;
    virtual void resetProperties();
    ...

class Knight : virtual public Character {
    ...
    virtual int getMaxLifePoints() const;
    virtual void resetProperties();
    ...

class Nninja : public Magician, public Knight {
    ...
    int getMaxLifePoints() const;
    void resetProperties();
    ...
```

5.4 Distruttori virtual

I distruttori delle classi che sono classi base di altre sono state definiti *virtual*. In questo modo, quando viene distrutta una entità, vengono chiamati in modo corretto tutti i distruttori della gerarchia a partire da quello del tipo effettivo dell'istanza.

5.5 Factory method

Per creare i vari personaggi è stata creata la classe `CharacterFactory` che implementa il design pattern chiamato *factory method*. In questo modo è possibile creare diversi personaggi e le varie funzioni restituiscono un puntatore a `Character` che punta ad una nuova istanza di un particolare sottotipo di `Character`.
Inoltre, dal momento che è sufficiente avere solo una classe come questa all'interno dell'applicazione, è stato applicato anche il pattern *singleton*.

```
class CharacterFactory
{
public:
    static CharacterFactory &getInstance();
    ~CharacterFactory();

    Character * createSwordsman();
    Character * createArcher();
    Character * createMagician();
    Character * createWarrior();
    Character * createNinja();

    CharacterFactory(CharacterFactory const&) = delete;
    void operator=(CharacterFactory const&) = delete;
private:
    CharacterFactory();
};
```