Ingegneria Informatica *A.A. 2021-2022*
Programmazione Avanzata [38090 – MOD2] *Gianluca Maffioletti [1059062]*

RELAZIONE PROGETTO C++

1 – INTRODUZIONE

Il programma realizzato riguarda un gioco da tavolo "virtuale" che prende spunto dagli scacchi e dalla dama.
Per lo svolgimento sono previsti due giocatori e un tavolo da gioco di dimensione 8x8, con righe identificate da un numero (1-8) e colonne identificate da una lettera (A-H).
Ogni giocatore ha dei punti vita iniziali, decisi dai giocatori, e lo scopo del gioco è quello di sconfiggere l'avversario azzerando i suoi punti vita.
Per fare questo, ogni giocatore ha a disposizione delle pedine che può schierare sul tavolo con cui dovrà attaccare l'avversario o le pedine avversarie, oppure difendersi dagli attacchi delle pedine avversarie.

2 – FUNZIONAMENTO

Il gioco è suddiviso in turni.
Ad ogni turno ogni giocatore può effettuare una mossa:

- Posizionare una pedina;
- Unire due pedine;
- Muovere una pedina;
- Attaccare una pedina avversaria;
- Attaccare l'avversario.

2.1 – Posizionare una pedina

Una pedina da posizionare può essere un attaccante o un difensore.
Ogni pedina attaccante ha un valore di punti "attacco" iniziale pari a 1, mentre ogni pedina difensore ha un valore di punti "difesa" iniziale pari a 1.
Le pedine possono essere posizionate sulla colonna più esterna (colonna A per il primo giocatore e colonna H per il secondo giocatore).

2.2 – Unire due pedine

Quando due pedine si trovano in due celle adiacenti (non diagonale) possono essere unite per creare una sola pedina i cui punti "attacco" e/o "difesa" sono la somma dei rispettivi punti delle due pedine unite e la cui posizione è quella della prima pedina selezionata per unirsi.

Se unisco una pedina attaccante ad una pedina difensore ottengo una pedina attaccante-difensore, che quindi ha sia punti “attacco” che “difesa”.

2.3 – Muovere una pedina

Le pedine possono essere mosse di una cella.
Le pedine attaccante e difensore possono essere mosse in orizzontale, in verticale o in diagonale, mentre le pedine attaccante-difensore possono essere mosse solo in verticale o in orizzontale.

2.4 – Attaccare una pedina avversaria o l’avversario

Una pedina attaccante o attaccante-difensore può attaccare una pedina avversaria quando quest’ultima si trova in una cella adiacente.
Se la pedina attaccata è un attaccante, questa viene eliminata.
Se la pedina attaccata è un difensore o un attaccante-difensore, allora i suoi punti “difesa” vengono diminuiti del valore dei punti “attacco” della pedina attaccante. Se i punti “difesa” si annullano, la pedina attaccata viene eliminata.
Dopo l’attacco, i punti “attacco” della pedina attaccante vengono diminuiti di 1. Quando si azzerano: se la pedina è un attaccante questa viene eliminata; se la pedina è un attaccante-difensore questa diventa un difensore, mantenendo i punti “difesa”.
Quando una pedina attaccante si trova sulla colonna dell’avversario può attaccare direttamente l’avversario, i cui punti vita diminuiscono del valore dei punti “attacco” della pedina attaccante.
Quando i punti vita dell’avversario si azzerano, il giocatore vince e la partita termina.

3 – DIAGRAMMA DELLE CLASSI

Di seguito è riportato il diagramma delle classi (*Figura 1*).

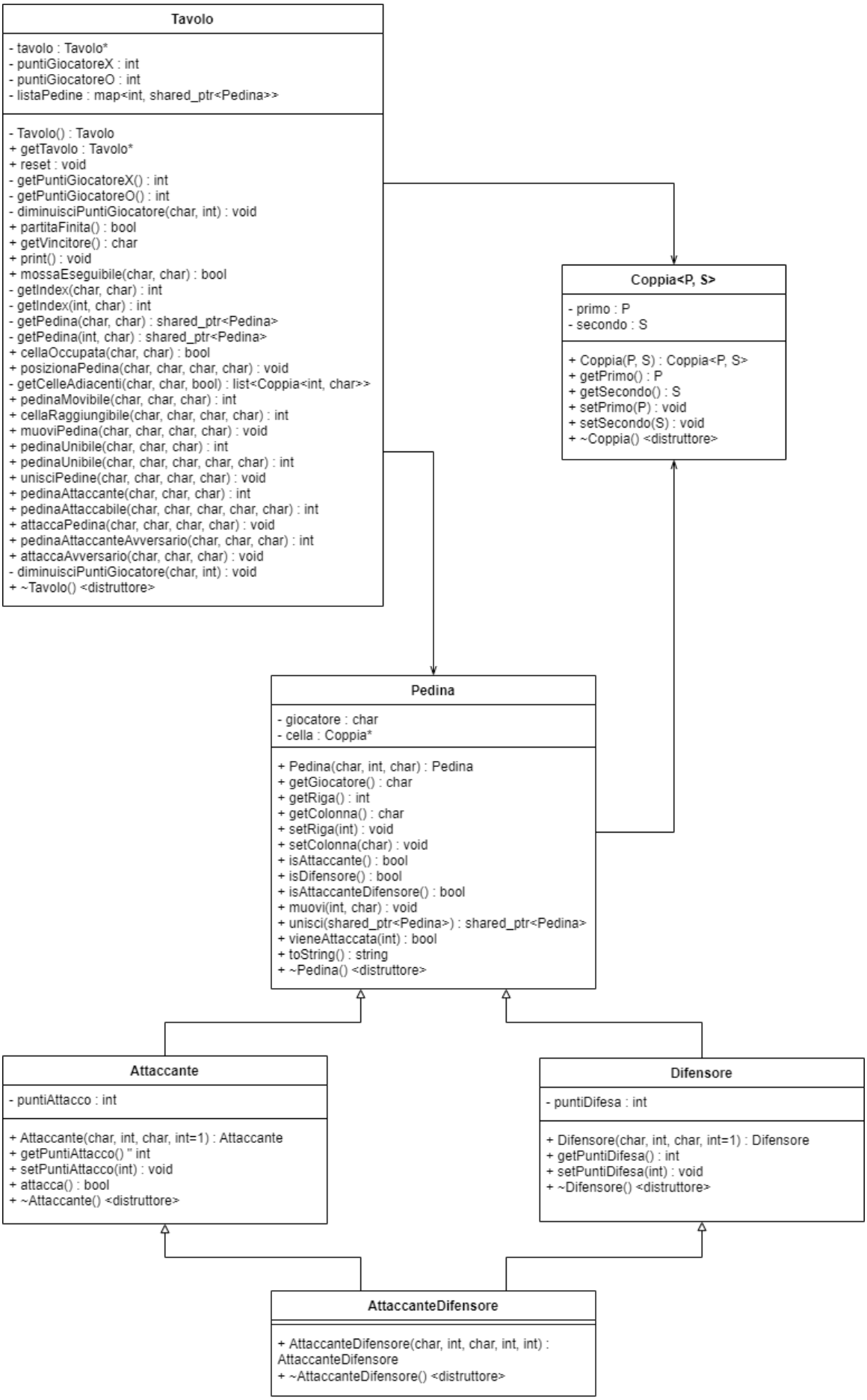

Figura 1 - Diagramma delle classi

4 – CLASSE PEDINA

La classe *Pedina* è la classe base che rappresenta le pedine che i giocatori posizionano sul tavolo.

4.1 – Abstract class

Non essendo possibile schierare una pedina generica nel gioco, si è scelto di rendere la classe *Pedina* astratta, definendo alcuni metodi *pure virtual* (paragrafo 4.5, *Figura 3*).

4.2 – Templates

La classe *Pedina* ha un campo *"cella"*, che è rappresentato da una coppia di valori *int* e *char* che indicano rispettivamente la riga e la colonna su cui è posizionata la pedina.
Per realizzare questo campo, è stata definita una classe generica *Coppia<P, S>* ricorrendo ai template del C++.
Questa classe è quindi dotata di due campi *"primo"* e *"secondo* rispettivamente appartenenti ai tipi parametro *P* e *S*, e dei metodi getter e setter, che verranno utilizzati dalla classe *Pedina* per manipolarne il contenuto.

4.3 – Ereditarietà multipla

A partire dalla classe *Pedina* vengono definite le classi *Attaccante* e *Difensore*, utilizzando l'ereditarietà pubblica. Le due classi differiscono per i campi *"puntiAttacco"* e *"puntiDifesa"* e dai relativi metodi getter e setter, oltre che dal metodo *"attacca()"* presente solo nella classe *Attaccante*.
Successivamente, si è definita la classe ibrida *AttaccanteDifensore*, realizzata con il meccanismo dell'ereditarietà multipla (*Figura 2*) a partire dalle due classi precedentemente descritte, costruendo così una struttura a diamante (come si vede in *Figura 1*).
Per evitare problemi di name clash e member duplication, l'ereditarietà delle classi *Attaccante* e *Difensore* dalla classe base *Pedina* sono state definite *virtual*.

```
class AttaccanteDifensore: public Attaccante, public Difensore {
```

Figura 2 - Ereditarietà AttaccanteDifensore

4.4 – Default parameters

Poiché, come descritto nel paragrafo 2.1, le pedine *Attaccante* e *Difensore* quando vengono posizionate hanno un valore iniziale rispettivamente dei *"puntiAttacco"* e dei *"puntiDifesa"* pari a 1, nel codice questo fatto è stato semplificato impostando nei costruttori il valore di default.

4.5 – Membri virtual e Overriding

I membri della classe base *Pedina* possono essere:

- *Non virtual*, come i campi *"giocatore"* e *"cella"*, con i relativi metodi getter e setter, e il metodo *"muovi()"*, che sono comuni a tutti i tipi di pedine;
- *Virtual*, come il metodo *"toString()"*, che viene richiamato anche nelle sottoclassi;
- *Pure virtual*, come i metodi *"isAttaccante()"*, *"isDifensore()"*, *"unisci()"* e *"vieneAttaccata()"*, le cui risposte dipendono esclusivamente dal tipo di pedina che li invoca, e perciò non possono essere definiti nella classe base.

I membri *virtual* (*Figura 3*) sono quindi soggetti ad Overriding nelle sottoclassi.

```
virtual bool isAttaccante() = 0; // pure virtual
virtual bool isDifensore() = 0; // pure virtual
bool isAttaccanteDifensore() {return isAttaccante() && isDifensore();}

void muovi(int, char);
virtual shared_ptr<Pedina> unisci(shared_ptr<Pedina>) = 0; // pure virtual
virtual bool vieneAttaccata(int) = 0; // pure virtual

virtual string toString();

virtual ~Pedina();
```

Figura 3 - Metodi classe Pedina

Nella sottoclasse *AttaccanteDifensore* per alcuni metodi (*Figura 4*) non vi è un'esplicita ridefinizione, ma semplicemente richiamano il metodo di una delle due super classi; ad esempio, il metodo *"isAttaccante()"* richiama l'omonimo metodo della super classe *Attaccante*, mentre il metodo *"isDifensore()"* richiama il metodo della super classe *Difensore*.
Per quanto riguarda invece il metodo *"attacca()"*, dichiarato *non virtual* nella super classe *Attaccante*, esso viene invocato dalla classe *AttaccanteDifensore* tramite la keyword *using* del C++.

```
AttaccanteDifensore(char, int, char, int, int);

bool isAttaccante() {return Attaccante::isAttaccante();} // override
bool isDifensore() {return Difensore::isDifensore();} // override

shared_ptr<Pedina> unisci(shared_ptr<Pedina>); // override
using Attaccante::attacca;
bool vieneAttaccata(int p) {return Difensore::vieneAttaccata(p);}

string toString(); // override

~AttaccanteDifensore();
```

Figura 4 - Metodi classe AttaccanteDifensore

4.6 – Distruttore

In questa struttura gerarchica, fondamentale è dichiarare *virtual* i distruttori delle classi base *Pedina*, *Attaccante* e *Difensore*, in modo tale da consentire di invocare il costruttore della classe dell'oggetto e non della classe del puntatore, in modo da richiamare tutti i distruttori risalendo la gerarchia dalla classe derivata dell'oggetto che è stato rimosso (*Figura 3*).
In particolare, in questo caso il campo *"cella"* della classe base *Pedina* è un puntatore ad un oggetto di tipo *Coppia*, perciò il distruttore della classe *Pedina* provvede a deallocare la memoria richiamando il metodo *delete* sul puntatore.

5 – CLASSE TAVOLO

La classe *Tavolo* è il gestore della partita: contiene i punti vita dei giocatori, la lista delle pedine schierate e i metodi per gestire le mosse dei giocatori.

5.1 – Design pattern Singleton

Per questa classe si è scelto di utilizzare il design pattern *Singleton*, in modo tale da consentire una sola istanza di *Tavolo*, siccome può esserci una sola partita attiva in un determinato istante.
Per fare questo, quindi, è stato definito un puntatore statico privato, un costruttore privato e un metodo getter pubblico per restituire la singola istanza della classe.

5.2 – STL e Smart Pointers

Per realizzare il campo *"listaPedine"* si è scelto di utilizzare la classe *map*, che rappresenta un contenitore associativo della *Standard Template Library* del C++, che consente di mantenere una lista di valori associati ad una chiave.
In questo caso, il valore è costituito da uno *shared pointer* alla classe *Pedina* mentre la chiave è un indice intero costruito univocamente a partire dagli indici di riga e colonna della pedina.
La classe *map* risulta efficiente nei vari metodi della classe *Tavolo* perché consente di ricavare lo *shared pointer* di una pedina in una specifica cella tramite il metodo *find()* (*Figura 5*), che prende in input il valore della chiave e restituisce il relativo valore con complessità logaritmica.

Per quanto riguarda la gestione delle pedine, come si è visto, si è scelto di utilizzare gli *smart pointers* per la loro efficienza e sicurezza nella gestione della memoria. In particolare, vengono utilizzati gli *shared pointers* per consentire a più puntatori di puntare al medesimo oggetto (siccome la quasi totalità dei metodi della classe Tavolo necessitano di accedere agli oggetti *Pedina*).
Grazie alla sottotipazione, gli *shared pointers* sono dichiarati come puntatori alla classe *Pedina* ma possono puntare ad oggetti delle tre sottoclassi.

```
shared_ptr<Pedina> Tavolo::getPedina(int riga, char colonna) {
    auto it = this->listaPedine.find(getIndex(riga, colonna));
    if (it == this->listaPedine.end())
        return NULL;
    return it->second;
}
```

Figura 5 - Metodo getPedina

Nel metodo *"mossaEseguibile()"*, che prende in input due caratteri rappresentanti il giocatore di turno e la mossa che vuole eseguire e restituisce *true* se la mossa è eseguibile o *false* altrimenti, si rende necessario non accedere puntualmente ad una specifica pedina ma scorrere l'intera lista di pedine, per controllare che una determinata condizione sia verificata su almeno una pedina.
Per fare questo, vengono utilizzati gli *iterator* della classe *map* (*Figura 6*). Per evitare eccessiva verbosità del codice, il tipo dell'*iterator* viene ricavato automaticamente grazie alla keyword *auto*.

```
for (auto it = this->listaPedine.begin(); it != this->listaPedine.end(); it++) {
    // ...
}
```

Figura 6 - Ciclo for con iterator

Ricordando che i campi *"puntiAttacco"* e *"puntiDifesa"*, così come i relativi getter e setter e il metodo *"attacca()"*, sono dichiarati nelle sottoclassi *Attaccante* e *Difensore*, ad essi non è possibile accedere dalla classe *Pedina*, e quindi anche dagli *smart pointers* contenuti nella *"listaPedine"*.
Quindi nei metodi *"unisci()"* delle tre sottoclassi e nei metodi *"attaccaPedina()"* e *"attaccaAvversario()"* della classe *Tavolo*, si è reso necessario effettuare un cast dal tipo *shared_ptr<Pedina>* allo *shared pointer* di una delle sottoclassi. Ciò viene realizzato con il metodo *dynamic_pointer_cast()*, come mostrato in *Figura 7*.

```
shared_ptr<AttaccanteDifensore> ad = dynamic_pointer_cast<AttaccanteDifensore>(pedina);
```

Figura 7 – Casting dinamico shared pointer

5.3 – Overloading

Siccome negli oggetti *Pedina* l'indice di riga è di tipo intero, mentre l'indice fornito da tastiera dall'utente è di tipo *char*, per semplificare il codice ed evitare troppi cast espliciti sono stati definiti i metodi *"getIndex()"* e *"getPedina()"* che prendono in input gli indici di riga e colonna e restituiscono rispettivamente l'indice univoco intero e lo *shared pointer* alla pedina nella cella specificata (se esiste, altrimenti *NULL*), con due segnature differenti (*Figura 8, Figura 5*), realizzando overloading sul tipo dei parametri.

```
int getIndex(char, char); // overloading
int getIndex(int, char); // overloading
shared_ptr<Pedina> getPedina(char, char); // overloading
shared_ptr<Pedina> getPedina(int, char); // overloading
```

Figura 8 - Overloading

6 – MAIN

Nel metodo *main()* innanzitutto viene definito uno *unique pointer* che punta alla singola istanza di *Tavolo*.
Con una serie di cicli viene strutturata l'alternanza dei turni e gli inserimenti da tastiera da parte dell'utente, gestendo i possibili errori.
All'inizio di ogni turno viene invocato il metodo *"print()"* della classe *Tavolo* che stampa le liste delle pedine di ciascun giocatore, oltre ai punti vita residui, e il tavolo da gioco con le pedine schierata (*Figura 9*).

```
~~~ Giocatore X - Punti vita: 1 ~~~
Lista pedine:
 - F2 (A) - Punti Attacco: 2

~~~ Giocatore O - Punti vita: 1 ~~~
Lista pedine:
 - H1 (A) - Punti Attacco: 1
 - H2 (D) - Punti Difesa: 1

    +---+---+---+---+---+---+---+---+
    | A | B | C | D | E | F | G | H |
+---+---+---+---+---+---+---+---+---+---+
| 1 |   |   |   |   |   |   |   | O | 1 |
+---+---+---+---+---+---+---+---+---+---+
| 2 |   |   |   |   |   | X |   | O | 2 |
+---+---+---+---+---+---+---+---+---+---+
| 3 |   |   |   |   |   |   |   |   | 3 |
+---+---+---+---+---+---+---+---+---+---+
| 4 |   |   |   |   |   |   |   |   | 4 |
+---+---+---+---+---+---+---+---+---+---+
| 5 |   |   |   |   |   |   |   |   | 5 |
+---+---+---+---+---+---+---+---+---+---+
| 6 |   |   |   |   |   |   |   |   | 6 |
+---+---+---+---+---+---+---+---+---+---+
| 7 |   |   |   |   |   |   |   |   | 7 |
+---+---+---+---+---+---+---+---+---+---+
| 8 |   |   |   |   |   |   |   |   | 8 |
+---+---+---+---+---+---+---+---+---+---+
    | A | B | C | D | E | F | G | H |
    +---+---+---+---+---+---+---+---+
```

Figura 9 - Vista di gioco