Progetto c++

AA 2021/2022

Corso di P. avanzata

Prof. Angelo Gargantini

1 LUGLIO

Nicola Zambelli
Matricola:1053015

Introduzione

Il software realizzato permette di implementare una coda elettronica nei loghi dove si vogliono sostituire file di attesa fisiche, perchè comportano assembramenti o semplicemente percè si vogliono implementare delle politiche di gestione della fila più efficenti.
Gli utenti che possono utilizzare l'applicazione possono essere di due tipologie:

- Cliente: colui che si iscrive ad una determinata coda e attende il proprio turno.
- Gestore: colui che implementa la coda eletronica e ne gestisce le regole.

Il cliente può avere una priorità alta o standard, i clienti con priorità alta hanno il diritto di prelazione su quelli con priorità standard. Essi possono iscriversi a una coda di proprio interesse, il gestore accetta la richiesta e lo inserisce in coda, successivamente il gestore può servire il prossimo utente, scegliendo il cliente che ha la priorità più alta, in caso di pari priorità sceglie il cliente che ha fatto prima la richiesta.

Funzionamento

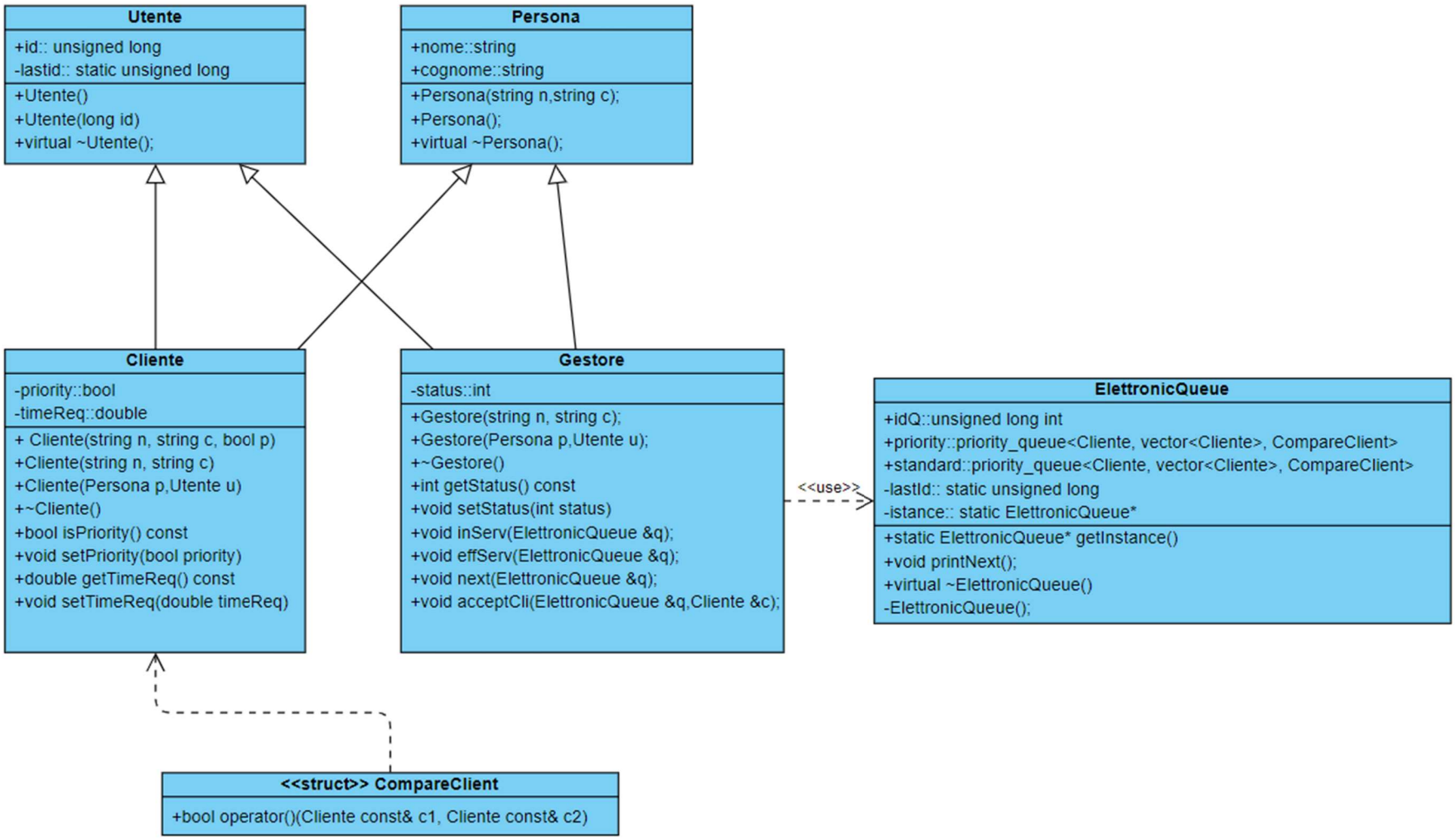
Librerie Utilizzate:

- `<iostream>` :input/output console
- `<memory>` : implementazione smart pointers
- `<time.h>` : implementazione dei tempi delle richieste
- `<unistd.h>` : funzionalità legate al tempo(es. sleep())
- `<queue>` : implementazione priority queue

Class Diagram UML

Le classi Implementate sono:

- Utente
- Persona
- Cliente
- Gestore
- ElettronicQueue


Utente
+id:: unsigned long
-lastid:: static unsigned long
+Utente()
+Utente(long id)
+virtual ~Utente();
Persona
+nome::string
+cognome::string
+Persona(string n,string c);
+Persona();
+virtual ~Persona();
Cliente
-priority::bool
-timeReq::double
+ Cliente(string n, string c, bool p)
+Cliente(string n, string c)
+Cliente(Persona p,Utente u)
+~Cliente()
+bool isPriority() const
+void setPriority(bool priority)
+double getTimeReq() const
+void setTimeReq(double timeReq)
Gestore
-status::int
+Gestore(string n, string c);
+Gestore(Persona p,Utente u);
+~Gestore()
+int getStatus() const
+void setStatus(int status)
+void inServ(ElettronicQueue &q);
+void effServ(ElettronicQueue &q);
+void next(ElettronicQueue &q);
+void acceptCli(ElettronicQueue &q,Cliente &c);
<<use>>
ElettronicQueue
+idQ::unsigned long int
+priority::priority_queue<Cliente, vector<Cliente>, CompareClient>
+standard::priority_queue<Cliente, vector<Cliente>, CompareClient>
-lastId:: static unsigned long
-istance:: static ElettronicQueue*
+static ElettronicQueue* getInstance()
+void printNext();
+virtual ~ElettronicQueue()
-ElettronicQueue();
<<struct>> CompareClient
+bool operator()(Cliente const& c1, Cliente const& c2)

Scelte implementative

Ereditarieta multipla

Per prima cosa si è definita la classe Utente che contiene che contiene un id univoco generato nel costruttore dopodichè la classe Persona contiene delle informazioni sull'individuo che interagisce cpn il sistema. Le classi Cliente e Gestore possono essere considerati come istanze che derivano sia da Persona che da Utente, perciò la suluzione è stata quella di applicare l'ereditarietà multipla:

```
class Utente {/*...*/}
class Persona {/*...*/}
class Cliente: public Persona, public Utente {/*...*/}
class Gestore:public Persona, public Utente {/*...*/}
```

Distruttore Virtual

Tutti i distruttori delle classi che ammettono sottoclassi sono stati dichiarati virtual, in modo tale che la distruzione di un oggetto puntato da un riferimento della superclasse richiami in modo corretto i distruttori delle sottoclassi.

```
virtual ~Utente(){/*...*/}
virtual ~Persona(){/*...*/}
```

Initializer list

I Costruttori delle classi Cliente e Gestore sono stati implementati utilizzando il meccanismo dell'initializer list, semplificando due compiti: il costruttore chiama i costruttori delle superclassi e li inizializza come tali (una sorta di *super* di *java*), inoltre con la stessa strategia vengono definiti altri costruttori che si appoggiano a quello di default della stessa classe(*this* di *java*).

```
Cliente::Cliente(string n, string c,bool p):
 Utente(), Persona(n,c),priority(p),timeReq(0){} //super
Cliente::Cliente(string n, string c):
 Cliente(n,c,false){} //this
```

Overriding: metodi virtual

Alcuni metodi sono stati definiti virtual, in modo da poter utilizzare il polimorfismo offerto dal linguaggio. Ad esempio:

```
virtual void print(){
cout<<"Utente n°"<<this->id<<endl;
}//classe utente
void print(){
cout<<"Utente n°"<<this->id<<", "<<this->nome<<" "
<<this->cognome<<endl;
}//classe Cliente
```

Overload dei metodi

Come accennato nel paragrafo suul'initializer list si è attuato l'overload dei metodi, definendo diverse funzioni con lo stesso nome ma differente segnatura, in modo da differenziarne il comportamento in base ai parametri passati.In particolare, è stato utilizzato con i costruttori.

```
Cliente(string n, string c, bool p);
Cliente(string n, string c);
Cliente(Persona p,Utente u);
```

Singleton pattern

Per garantire l'unicità della classe ElettronicQueue, questa è stata definita secondo il pattern Singleton, definendo il costruttore privato. In questo modo non `e possibile creare istanze della classe al di fuori della stessa. Tuttavia, é possibile accedere all istanza tramite il metodo getInstance(), che restituisce un puntatore all'istanza della classe.
In questo modo, si può accedere ai metodi per eseguire operazioni ElettronicQueue
Si riporta di seguito la definizione della classe ElettronicQueue:

```
class ElettronicQueue {
public:
     priority_queue<Cliente, vector<Cliente>, CompareClient> priority;
     priority_queue<Cliente, vector<Cliente>, CompareClient> standard;
     unsigned long idQ;
     static ElettronicQueue* getInstance();
     void printNext();
     virtual ~ElettronicQueue();
private:
     static unsigned long lastIdQ;
     static ElettronicQueue* instance;
     ElettronicQueue();
};
```

STL: std:: priority_queue

All'interno dell'applicazione è stato utilizzato il container priority_queue fornito dalla *Standard Template Library*. In particolare, la classe ElettronicQueue ha due attributi di nome *priority* e *standard* che rappresentano rispettivamente la coda dei Clienti con priorità alta e bassa. L'implementazione chiede di fornire la politica di confronto tra due oggetti della *priority_queue*. A tale scopo è stata implementata la struct *CompareClient* nel file *Cliente.h* che contiene il metodo *operator()* che confronta due Clienti. Il Cliente con *TimeReq* più basso va servito prima e sarà in cima alla *priority_queue*. Di seguito si riporta la struct CompareClient:

```
struct CompareClient {
    bool operator()(Cliente const& c1, Cliente const& c2) {
        return c1.timeReq > c2.timeReq;
//ritorna vero se é ordinato in base al tempo di richiesta
     } };
```

Smart pointers:unique_ptr

Nel progetto realizzato, si è ricorso all'utilizzo degli *smart pointers* per semplificare l' utilizzo di puntatori agli oggetti che permettono di evitare problemi quali *dangling pointers e memory leaks*, tipici dell'utilizzo non corretto dei puntatori. In particolare, si è utilizzato *l'unique_ptr* per implementere l'*ElettronicQueue*. In questo modo, non può essere copiato o passato per parametri ai metodi. Tale vincolo non è una limitazione in quanto tutti i metodi che operano sull clase *ElettronicQueue* hanno gli argomenti passati per parametro.

```
int main() { /* .... */
unique_ptr<ElettronicQueue> eq(ElettronicQueue::getInstance());
//ElettronicQueue* eq= ElettronicQueue::getInstance();
```