C++ - Macchina a Stati Finiti con il Functor Pattern

Franco Diaspro

Programmatore Senior C++ presso Aesys

INDICE

Introduzione

Le strategie per implementare una macchina a stati nei linguaggi di programmazione sono le più disparate, in riferimenti se ne possono trovare alcune. Studiando il Functor Pattern, ho avuto l'intuizione che questo pattern è ideale per questa attività. Con questo obiettivo e per rendere un servizio alla didattica sia di chi usa i functor che di chi sviluppa macchine a stati finiti ho cominciato a sviluppare il progetto che descriverò in quest'articolo.
Il codice del progetto si può trovare al seguente link:

https://github.com/fdiaspro/funct_state_machine/releases/tag/vers_unique_ptr .

Mentre il caso con i thread possiamo trovarlo al seguente link:

https://github.com/fdiaspro/funct_state_machine/releases/tag/Thread_Vers .

Descrizione macchina a stati da implementare

La macchina a stati finiti è una modalità per descrivere un processo che si basa su due entità

- Lo stato
- L'evento che implica la transizione di stato

Nella figura seguente ne possiamo vedere un esempio: abbiamo modellato un motore elettrico che tramite un potenziometro può

- Accendersi/Partire
- Variare la velocità
- Fermarsi
- Spegnersi

La modellizzazione del processo è molto semplice: si parte da uno stato IDLE nel quale si permane a meno che non venga dato un comando con velocità superiore a zero ed in questo caso il motore va in START. In START se si varia la velocità si va in CHANGE SPEED se la velocità e maggiore di zero, altrimenti in STOP. In CHANGE SPEED si rimane fintanto che non si dà velocità pari a zero, in questo caso si va in STOP. Da STOP o si dà velocità maggiore zero e si torna in CHANGE SPEED, oppure con zero si torna in IDLE (spegnendo il motore).

Questo semplice processo lo implementeremo utilizzando come linguaggio il C++14.

Introduzione al functor pattern

Andiamo a descrivere in questo paragrafo una versione semplificata del functor pattern. Questa metodologia di programmazione utilizza i template: grazie a questa tecnica possiamo indicare un approccio (policy) per affrontare problematiche che saranno poi implementate attraverso le classi che svilupperemo: le classi che implementano la policy sono dette "policy classes". Specificando come template diversi tipi di classi possiamo utilizzare questo modello in un'infinità di casi.

```
template <class R, class P>
class FunctorImpl {
protected:

public:
        virtual R operator()(P &p) = 0;

        virtual ~FunctorImpl() {}

};

// functor
template <class R, class P>
class Functor {
public:
        typedef R ResultType;
        typedef P ParmType;

        typedef FunctorImpl<ResultType, ParmType> Impl;

        Functor();

         Functor(std::unique_ptr<Impl> pImpl) :spImpl_(pImpl) {
        }

        ResultType operator()(ParmType &p) {
                return (*spImpl_)(p);
        }

        virtual ~Functor() ;
protected:

        std::unique_ptr<Impl> spImpl_;

};
```

La policy che stiamo ora osservando ci dice che abbiamo una classe P che utilizziamo come Parametro dell'operator() (il Functor) e poi ci restituirà una classe di tipo R come Risultato.

Quindi ricapitolando abbiamo tre attori

1) Una classe che rappresenta i parametri in ingresso
2) Una classe che implementa la policy
3) Una classe che è il risultato di una elaborazione

Con queste elementi in mente possiamo affrontare il successivo paragrafo.

Implementare la macchina a stati con la policy del functor

Nel precedente paragrafo abbiamo visto un modello (policy) che ha un classe, utilizzata come parametro, una classe che attraverso un suo operatore riceve questo parametro e lo trasforma in un risultato. Ora pensiamo a cosa è una macchina: uno stato riceve un evento ed attraverso una regola va in un'altro stato (che potrebbe essere il medesimo). Quindi se riprendiamo i tre punti visti nel paragrafo precedente possiamo riepilogare nel modo seguente

1) P → Event Data
2) R → State
3) operator() → Transition Link

Quindi la classe P assume il ruolo di evento, la classe R assume il ruolo di stato mentre il functor rappresenta l'operatore che governa la transizione di stato e che restituirà il nuovo stato.

```
template <typename payload>
class EventData
{
public:

        virtual payload& getEventData() = 0;
        virtual ~EventData() {};

};
```

Sopra l'implementazione della classe EventData, mentre di seguito la classe che definisce la policy per la macchina a stati che semplicemente specializza il functor visto nel paragrafo precedente.

```
#include <string>
#include "EventData.h"
#include "Functor.h"

template <class State, class EventData>
class StateMachine : FunctorImpl<State,   EventData>
{
public:


        virtual ~StateMachine() {};

        virtual State operator() (EventData& p) = 0;


};
```

Possiamo cominciare a pensare alla classe che rappresenta lo stato ed a quelle che contiene l'operatore che governa la transizione come la stessa classe: nel prossimo paragrafo vedremo come.

Implementazione Macchina a Stati

Con gli elementi visti in precedenza possiamo passare ad implementare una soluzione software per il processo descritto nel paragrafo Descrizione macchina a stati da implementare. Per prima cosa andiamo a definire la classe EventData che nel nostro caso sarà la velocità

```
template <typename payload>
class EventData
{
public:

        virtual payload& getEventData() = 0;
        virtual ~EventData() {};

};
```

In questo caso la velocità si può rappresentare con un intero positivo, ma in altri casi si hanno altre tipologie di evento, queste si possono modellare con un'opportuna classe e poi utilizzarla nella specializzazione della classe EventData.

Adesso dobbiamo definire gli stati, per questo definiremo una classe che avrà memoria dello stato attuale, della velocità attuale ed avrà un operatore che sarà in grado di portarci nel nuovo stato, nel caso di arrivo di un evento: per fare questo specializziamo la StateMachine. Come R (Stato) utilizzeremo un puntatore alla classe engineMachineState, in questo modo il compilatore accetterà la definizione di una classe per la specializzazione prima che essa sia stata completamente definita.

Il functor sarà virtuale puro perchè sarà poi definito nelle classi che implementano gli stati che governano le transizioni:

```
typedef enum {
        IDLE =0,
        START,
        STOP,
        CHANGE_SPEED,
        MAX_ENUM_EMS
} engine_machine_state;

std::ostream& operator<<(std::ostream& os, engine_machine_state& p);

class engineMachineState : public StateMachine<engineMachineState*, Speed>
{
        engine_machine_state ems{ IDLE };
        unsigned int         speed{ 0 };
protected:
        void printInfo() ;
public:
    engineMachineState(){};
    ~engineMachineState(){};
        engineMachineState(engine_machine_state _ems, unsigned int _speed) :
                        ems(_ems), speed(_speed){};
        engine_machine_state& getEms() { return ems; };
        unsigned int          getSpeed() { return speed; };
        void setEms(engine_machine_state _ems) { ems = _ems; };
        void setSpeed(unsigned int _speed) { speed = _speed; };
        virtual engineMachineState* operator()(Speed &p)=0;
        friend std::ostream& operator<<(std::ostream& os, engine_machine_state& p);
};
```

Ora avremo per ogni stato una classe che specifica lo stato e garantisce la coerenza delle successive transizioni. Andiamo a vedere il caso dello stato IDLE. In questo caso implementiamo, come descritto nel diagramma degli stati, l'accensione oppure il permanere in IDLE

```
class IdleState : public engineMachineState
{
public:

        IdleState(): engineMachineState(IDLE, 0) {printInfo() ;};
        engineMachineState* operator()(Speed &p) ;
        virtual ~IdleState(){};

};
```

```
engineMachineState* IdleState::operator()(Speed &p)
{
        engineMachineState* ptr;

        std::cout << __PRETTY_FUNCTION__ << std::endl;

        if (p.getEventData() == 0)
        {
                ptr = new IdleState();
        }
        else
        {
                ptr = new StartState(p.getEventData());
        }


        return ptr;
}
```

Quindi il functor, che implementa la transizione dello stato, verifica il valore della velocità: se è zero restituisce un altro IDLE STATE, mentre se è maggiore di zero, porta il motore in START e fissa propriamente la velocità.

Analogamente sono stati sviluppati gli altri stati, il codice completo si può trovare in https://github.com/fdiaspro/funct_state_machine/releases/tag/vers_unique_ptr .

Esecuzione della Macchina a Stati

Ora andiamo a vedere come usare la macchina a stati implementata; nella figura di seguito ne vediamo un esempio.

```
int main(int argc, char** argv)
{
        cout << "state machine sample" << endl;

    std::unique_ptr<engineMachineState> prtEms (new IdleState() );

    for (auto speedEvent : eventData)
    {
        std::unique_ptr<engineMachineState> ptAppoggio ( prtEms->operator()(speedEvent) );

        prtEms =std::move(ptAppoggio);
    }
}
```

Carichiamo una serie di eventData in un vettore, nel nostro caso è un vettore interno all'applicazione ma gli eventData potrebbero arrivare da una coda che riceve messaggi di rete. Un ciclo for prenderà dalla coda gli elementi dal vettore e li fornirà alle classi della macchina a stati determinandone l'evoluzione come mostrato di seguito

```
state machine example
state IDLE
speed 0
virtual engineMachineState* IdleState::operator()(Speed&)
state IDLE
speed 0
virtual engineMachineState* IdleState::operator()(Speed&)
state START
speed 20
virtual engineMachineState* StartState::operator()(Speed&)
state CHANGE_SPEED
speed 30
virtual engineMachineState* ChangeSpeedState::operator()(Speed&)
state CHANGE_SPEED
speed 40
virtual engineMachineState* ChangeSpeedState::operator()(Speed&)
state STOP
speed 0
virtual engineMachineState* StopState::operator()(Speed&)
state IDLE
speed 0

RUN FINISHED; exit value 0; real time: 120ms; user: 30ms; system: 30ms
```

Implementazione macchina a stati con due generatori di eventi.

Ora andiamo a vedere un caso più simile alla realtà: quando gli eventi vengono generati da diverse sorgenti. Per il caso di un motore elettrico possiamo pensare a diversi potenziometri in modo da avere più punti di comando. Facciamo l'ipotesi che nel nostro caso ci siano solo due punti di comando, simuleremo questo caso con due thread che riempiono separatamente il vector degli eventi.

Gli oggetti della standard template library (stl) sono thread safe a livello di singola operazione ma possono esserci problemi quando si accede da più thread: ad esempio se il thread 1 salva dati in uno std:vector, può accadere che un altro thread li cancella, così quando lui li va a cercare non li troverà più. Servono perciò policy che ci permettono di non dover gestire le concorrenze.

Le due sorgenti le abbiamo implementate con due thread che riempiono lo standard vector eventData attraverso la loro thread function. La thread function riceve per reference il vettore degli eventi e lo riempie con una cadenza che passiamo ugualmente dall'esterno. Per semplicità passiamo anche la velocità da impostare, questo ci permette di capire quale thread genera l'evento (è una semplificazione che torna utile nell'esecuzione quando stampiamo la risposta). La cadenza di generazione degli eventi ha valori temporali alti per un processo real time. Come già detto nel precedente paragrafo, nel nostro caso riempiamo noi il vector ma in un caso reale questi eventi potrebbero arrivare dalla rete o prodotti sul campo da sensori o trasduttori.

```
void sendEvent( std::vector<Speed>& eventQueue, unsigned int sleepTime, unsigned int speed)
{

    std::chrono::milliseconds dura( sleepTime *1000 );
    std::this_thread::sleep_for( dura );

    eventQueue.push_back( *new Speed(speed) );
    std::this_thread::sleep_for( dura );

    eventQueue.push_back( *new Speed(speed) );
    std::this_thread::sleep_for( dura );

    eventQueue.push_back( *new Speed(speed) );
    std::this_thread::sleep_for( dura );

    eventQueue.push_back( *new Speed(speed) );
    std::this_thread::sleep_for( dura );

    eventQueue.push_back( *new Speed(speed) );
    std::this_thread::sleep_for( dura );
    std::cout <<__PRETTY_FUNCTION__<<"\tend Thread Function" << std::endl;
}
```

Il blocco che governa le transizioni di stato l'abbiamo tolto dal main e l'abbiamo messo in un thread separato che guida l'evoluzione della macchina a stati. In questo modo lasciamo il main al suo mestiere: istanziare e coordinare risorse. La funzione moveEngine sarà quella che implementa l'evoluzione della macchina a stati; contiene un ciclo while all'infinito, questo è un caso tipico delle applicazioni real time: è buona norma mettere una funzione di sleep in ciclo così definito, altrimenti potrebbe finire che assorbe tutte le risorse dell'host mandando la macchina in crash.

```
void moveEngine(std::vector<Speed>& eventQueue, std::shared_ptr<engineMachineState>& prtEms )
{
    auto start = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double >  elapsed;

    while ( true )
        {
            auto now = std::chrono::high_resolution_clock::now();
            elapsed = now-start;
            if (eventQueue.size() > 0 )
            {
                auto speedEvent = eventQueue.back();
                eventQueue.pop_back() ;
                    std::cout<< "\t\t -------------------" <<std::endl
                    << "arrived speedEvent  " << speedEvent.getEventData()
                    << "\telapsed  " << elapsed.count() <<std::endl;
                std::unique_ptr<engineMachineState> ptAppoggio ( prtEms->operator()(speedEvent) );
                prtEms =std::move( ptAppoggio );
            }
            std::chrono::milliseconds dura( 100 );
            std::this_thread::sleep_for( dura );
        }
  }
```

A questo punto ci rimane di dare uno sguardo al main che come detto si limita ad istanziare i tre thread che implementano il processo ed attendere la fine dei due thread che generano gli eventi.

```
int main(int argc, char** argv)
{
    std::shared_ptr<engineMachineState> prtEms ( new IdleState() );
    std::vector<Speed> eventData;
    eventData.push_back( *new Speed(10) );//accendiamo il motore

    int time_1=3, time_2=4;
    int speed1=90, speed2=0;
    //engineMachineState* prtEms= (new IdleState());
    std::thread t1(sendEvent, std::ref(eventData),  time_1,speed1);
    std::thread t2(sendEvent, std::ref(eventData),  time_2, speed2);
    std::thread t3(moveEngine, std::ref(eventData), std::ref(prtEms) );

    t1.join();
    t2.join();
  }
```

Una cosa che mi piace sottolineare è che in questo paragrafo non abbiamo assolutamente parlato degli oggetti che implementano la macchina a stati ma ci siamo limitati a descrivere le modalità

con cui usare questi oggetti. Questo dimostra l'astrazione progettuale che permettono le tecniche dei modelli di progettazione (design pattern) grazie poi all'implementazione con linguaggi di programmazione orientati agli oggetti. Di seguito mostriamo una parte dell'esecuzione del programma

```
                    --------------------
arrived speedEvent  90  elapsed  9.00684
virtual engineMachineState* StopState::operator()(Speed&)
state START
speed 90
                    --------------------
arrived speedEvent  90  elapsed  12.0036
virtual engineMachineState* StartState::operator()(Speed&)
state CHANGE_SPEED
speed 90
                    --------------------
arrived speedEvent  0   elapsed  12.0052
virtual engineMachineState* ChangeSpeedState::operator()(Speed&)
state STOP
speed 0
                    --------------------
arrived speedEvent  90  elapsed  15.0054
virtual engineMachineState* StopState::operator()(Speed&)
state START
speed 90
                    --------------------
arrived speedEvent  0   elapsed  16.0037
virtual engineMachineState* StartState::operator()(Speed&)
state STOP
speed 0
void sendEvent(std::vector<Speed>&, unsigned int, unsigned int) end Thread Function
                    --------------------
arrived speedEvent  0   elapsed  20.006
virtual engineMachineState* StopState::operator()(Speed&)
state IDLE
speed 0
void sendEvent(std::vector<Speed>&, unsigned int, unsigned int) end Thread Function
terminate called without an active exception
```

Possiamo vedere come gli eventi vengano elaborati quasi in real time e vediamo come il programma termina quando le due thread-function dei generatori di eventi terminano.

Conclusioni

In conclusione vogliamo ricordare l'obiettivo di questa attività: fornire una metodologia per l'implementazione di una macchina a stati. Abbiamo implementato questa metodologia in C++14 cercando di mostrare a livello basilare le proprietà dei nuovi oggetti. La disponibilità di oggetti sempre più complessi, uniti ad una strategia di programmazione appropriata riesce a semplificare lo sviluppo del codice migliorando il carico di attività all'interno dei gruppi di lavoro. Inoltre se il codice adotta dei modelli che separano i ruoli (duty) si rende più immediata lo sviluppo, la condivisione e la manutenzione del codice stesso.

Riferimenti

1. State Machine Design in C++, CodeProject, by David Lafreniere, 20 Feb 2019
2. C++ State Machine with Threads, CodeProject, by David Lafreniere 26 Jan 2019
3. Modern C++ Design: Generic Programming and Design Patterns Applied, By Andrei Alexandrescu, Addison Wesley, February 01, 2001
4. Learn Advanced C++ Programming, by John Purcell, Udemy, Corso Online
5. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, Scott Meyers, O'Reilly, dic 2014