Informatica 3

LEZIONE 28: Macchine a stati finiti e Statecharts

- Modulo 1: Macchine a stati
- Modulo 2: Implementazioni di macchine a stati
- Modulo 3: Statecharts

Informatica 3

Lezione 28 - Modulo 1

Macchine a stati finiti

Introduzione

- I sistemi con comportamenti reattivi possono essere rappresentati tramite macchine a stati finiti e Statecharts
  - introduzione alle macchine a stati
  - possibili implementazioni in C++
  - cenni su Statecharts

Macchine a stati finiti

- Macchina a stati finiti (FSM):
  - formalismo per modellizzare un sistema reattivo
- Concetto di *stato*: informalmente rappresenta la condizione in cui si trova un sistema al variare del tempo
  - lo stato cattura gli aspetti rilevanti della storia del sistema
  - rappresenta una situazione nella vita del sistema nella quale il sistema compie delle attività

Macchine a stati estese (1)

- Una possibile interpretazione di stato in un sistema software è che ogni stato rappresenti un insieme di valori validi della memoria del programma
  - Es. variabile a 32 bit --> 2^{32}=4.294.967.296 stati
  - problema: troppi stati
    - 2^N, dove N=bit che costituiscono le variabili
- *Variabili di stato estese*: un programma può essere visto come costituito da aspetti qualitativi (stati) e quantitativi (variabili di stato estese)

  --> Macchine a stati estese

Macchine a stati estese (2)

Esempio:

- tastiera che dopo aver premuto 100.000 volte i tasti si rompe
  - macchina a stati:
    - 100.000 stati
    - la macchina passa da uno stato al successivo
    - L'ultimo stato rappresenta la situazione di "guasto"
  - macchina a stati estesa:
    - 2 stati: "funzionante", "guasta" + contatore
    - quando il contatore raggiunge il valore 100.000 la macchina passa dallo stato "funzionante" allo stato "guasta"

Condizioni di guardia

- Una macchina a stati estesa reagisce a stimoli basati non solo su aspetti qualitativi (stati) ma anche in funzione del valore delle variabili estese (es. contatore)
  - Es. quando il contatore della tastiera raggiunge un certo valore la macchina cambia stato
- Guardia: espressione booleana che viene valutata dinamicamente in base al valore delle variabili estese
  - influenzano il comportamento della macchina a stati, abilitando/disabilitando certe operazioni (es. cambiamento di stato)

Eventi

- *Evento*: occorrenza nel tempo o nello spazio di qualcosa che ha significato per il sistema
  - Es. evento: pressione di un tasto della tastiera
  - Può avere associato ad esso dei *parametri*, per trasferire informazioni quantitative legate all'evento stesso
    - Es. parametro: codice del tasto premuto

Azioni e transizioni

- Quando un evento viene rilevato dalla macchina a stati, la macchina a stati risponde all'evento compiendo delle *azioni*
  - cambiamento di una variabile
  - operazioni di input/output
  - generazione di nuovi eventi
  - cambiamento di stato
- Il passaggio da uno stato ad un altro prende il nome di *transizione di stato*
- L'evento che genera il cambiamento di stato prende il nome di *trigger*

Azioni e transizioni (2)

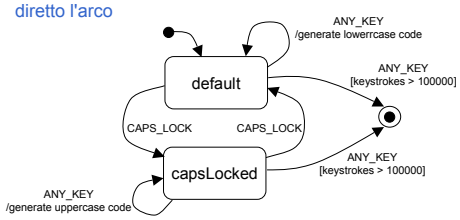
- In una macchina di stato estesa una transizione di stato può avere una guardia:
  - la transizione viene eseguita solamente se la guardia è verificata
    - Es. passaggio dallo stato "funzionante" allo stato "guasta" avviene solamente dopo che sono stati premuti 100000 tasti

Macchine di Mealy e di Moore

- Una macchina a stati ha due interpretazioni
  - Macchina di Mealy
    - le azioni sono associate alle transizioni di stato
  - Macchina di Moore
    - le azioni sono associate agli stati
    - Sono matematicamente equivalenti
    - In generale, la macchina di Moore richiede più stati per modellizzare lo stesso comportamento
    - Nella rappresentazione di sistemi software viene utilizzata essenzialmente la macchina di Moore

Diagramma delle transizioni di stato

- Rappresentazione grafica (macchina di Moore):
  - gli stati della FSM sono rappresentati dai nodi di un grafo
  - le transizioni tra stati vengono rappresentate tramite archi orientati (cioè caratterizzati da un verso)
  - ogni arco è identificato da un'etichetta che rappresenta le guardie che abilitano la transizione verso lo stato a cui è diretto l'arco



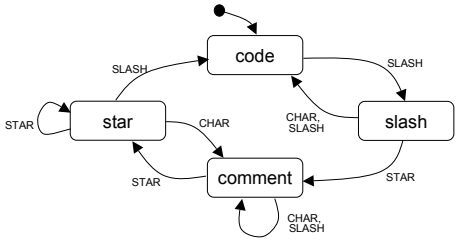
Informatica 3

Lezione 28 - Modulo 2

Implementazioni di macchine a stati

Esempio

Parser di commenti C:



Implementazione standard di una macchina a stati

- Interfacce della classe che rappresenta la macchina a stati
  - **init**: inizializza la macchina a stati, eseguendo la transizione iniziale
  - **dispatch**: prende un evento, lo valuta ed esegue le azioni associate all'evento
  - **tran**: esegue una transizione di stato arbitraria

Codice (1)

```
enum Signal {                          // enumeration for CParser signals
   CHAR_SIG, STAR_SIG, SLASH_SIG};
enum State {                           // enumeration for CParser states
   CODE, SLASH, COMMENT, STAR};

class CParser1 {
private:
   State myState;                      // the scalar state-variable
   long myCommentCtr;                  // comment character counter
   /* ... */                           // other CParser1 attributes
public:
   void init() { myCommentCtr = 0; tran(CODE); } // default transiton
   void dispatch(unsigned const sig);
   void tran(State target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
};
```

Codice (1)

```
enum Signal {                       // enumeration for CParser signals
   CHAR_SIG, STAR_SIG, SLASH_SIG};
enum State {                        // enumeration for CParser states
   CODE, SLASH, COMMENT, STAR};

class CParser1 {
private:
   State myState;                   // the scalar state-variable
   long myCommentCtr;               // comment character counter
   /* ... */                        // other CParser1 attributes
public:
   void init() { myCommentCtr = 0; tran(CODE); } // default transiton
   void dispatch(unsigned const sig);
   void tran(State target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
};
```


Codice (1)

```
enum Signal {                       // enumeration for CParser signals
   CHAR_SIG, STAR_SIG, SLASH_SIG};
enum State {                        // enumeration for CParser states
   CODE, SLASH, COMMENT, STAR};

class CParser1 {
private:
   State myState;                   // the scalar state-variable
   long myCommentCtr;               // comment character counter
   /* ... */                        // other CParser1 attributes
public:
   void init() { myCommentCtr = 0; tran(CODE); } // default transiton
   void dispatch(unsigned const sig);
   void tran(State target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
};
```


Codice (1)

```
enum Signal {                       // enumeration for CParser signals
   CHAR_SIG, STAR_SIG, SLASH_SIG};
enum State {                        // enumeration for CParser states
   CODE, SLASH, COMMENT, STAR};

class CParser1 {
private:
   State myState;                   // the scalar state-variable
   long myCommentCtr;               // comment character counter
   /* ... */                        // other CParser1 attributes
public:
   void init() { myCommentCtr = 0; tran(CODE); } // default transiton
   void dispatch(unsigned const sig);
   void tran(State target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
};
```


Codice (1)

```
enum Signal {                       // enumeration for CParser signals
   CHAR_SIG, STAR_SIG, SLASH_SIG};
enum State {                        // enumeration for CParser states
   CODE, SLASH, COMMENT, STAR};

class CParser1 {
private:
   State myState;                   // the scalar state-variable
   long myCommentCtr;               // comment character counter
   /* ... */                        // other CParser1 attributes
public:
   void init() { myCommentCtr = 0; tran(CODE); } // default transiton
   void dispatch(unsigned const sig);
   void tran(State target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
};
```

Codice (1)

```
enum Signal {                       // enumeration for CParser signals
   CHAR_SIG, STAR_SIG, SLASH_SIG};
enum State {                        // enumeration for CParser states
   CODE, SLASH, COMMENT, STAR};

class CParser1 {
private:
   State myState;                   // the scalar state-variable
   long myCommentCtr;               // comment character counter
   /* ... */                        // other CParser1 attributes
public:
   void init() { myCommentCtr = 0; tran(CODE); } // default transiton
   void dispatch(unsigned const sig);
   void tran(State target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
};
```



```
void CParser1::dispatch
     (unsigned const sig) {
   switch (myState) {
   case CODE:
      switch (sig) {
      case SLASH_SIG:
         tran(SLASH);
         break;
      }
      break;
   case SLASH:
      switch (sig) {
      case STAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT);
         break;
      case CHAR_SIG:
      case SLASH_SIG:
         tran(CODE);
         break;
      }
      break;
   case COMMENT:
      switch (sig) {
      case STAR_SIG:
         tran(STAR); break;
      case CHAR_SIG:
      case SLASH_SIG:
         ++myCommentCtr; break;
      }
      break;
   case STAR:
      switch (sig) {
      case STAR_SIG:
         ++myCommentCtr; break;
      case SLASH_SIG:
         myCommentCtr += 2;
         tran(CODE); break;
      case CHAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT); break;
      }
      break;
   }
}
```



```
void CParser1::dispatch
     (unsigned const sig) {
   switch (myState) {
   case CODE:
      switch (sig) {
      case SLASH_SIG:
         tran(SLASH);
         break;
      }
      break;
   case SLASH:
      switch (sig) {
      case STAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT);
         break;
      case CHAR_SIG:
      case SLASH_SIG:
         tran(CODE);
         break;
      }
      break;
   case COMMENT:
      switch (sig) {
      case STAR_SIG:
         tran(STAR); break;
      case CHAR_SIG:
      case SLASH_SIG:
         ++myCommentCtr; break;
      }
      break;
   case STAR:
      switch (sig) {
      case STAR_SIG:
         ++myCommentCtr; break;
      case SLASH_SIG:
         myCommentCtr += 2;
         tran(CODE); break;
      case CHAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT); break;
      }
      break;
   }
}
```



```
void CParser1::dispatch
     (unsigned const sig) {
   switch (myState) {
   case CODE:
      switch (sig) {
      case SLASH_SIG:
         tran(SLASH);
         break;
      }
      break;
   case SLASH:
      switch (sig) {
      case STAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT);
         break;
      case CHAR_SIG:
      case SLASH_SIG:
         tran(CODE);
         break;
      }
      break;
   case COMMENT:
      switch (sig) {
      case STAR_SIG:
         tran(STAR); break;
      case CHAR_SIG:
      case SLASH_SIG:
         ++myCommentCtr; break;
      }
      break;
   case STAR:
      switch (sig) {
      case STAR_SIG:
         ++myCommentCtr; break;
      case SLASH_SIG:
         myCommentCtr += 2;
         tran(CODE); break;
      case CHAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT); break;
      }
      break;
   }
}
```

```
void CParser1::dispatch
     (unsigned const sig) {
   switch (myState) {
   case CODE:
      switch (sig) {
      case SLASH_SIG:
         tran(SLASH);
         break;
      }
      break;
   case SLASH:
      switch (sig) {
      case STAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT);
         break;
      case CHAR_SIG:
      case SLASH_SIG:
         tran(CODE);
         break;
      }
      break;
   case COMMENT:
      switch (sig) {
      case STAR_SIG:
         tran(STAR); break;
      case CHAR_SIG:
      case SLASH_SIG:
         ++myCommentCtr; break;
      }
      break;
   case STAR:
      switch (sig) {
      case STAR_SIG:
         ++myCommentCtr; break;
      case SLASH_SIG:
         myCommentCtr += 2;
         tran(CODE); break;
      case CHAR_SIG:
         myCommentCtr += 2;
         tran(COMMENT); break;
      }
      break;
   }
}
```


Vantaggi/svantaggi

- Aspetti positivi:
  - codice compatto (variabile intera per rappresentare lo stato)
    e semplice
- Aspetti negativi:
  - non si può riusare il codice, tutti gli elementi della macchina a stati vanno codificati a mano
  - il tempo di gestione degli eventi non è costante, ma dipende dal numero degli stati

Tabella degli stati

- Altro approccio: utilizzare una tabella per rappresentare stati e transizioni

	CHAR_SIG	STAR_SIG	SLASH_SIG
code			do_nothing(), slash
slash	do_nothing(), code	a2(), comment	do_nothing(), cod
comment	a(1), comment	do_nothing(), star	a(1), comment
star	a2(), comment	a1(), star	a(2), code

- Ogni riga della tabella rappresenta uno stato, le colonne rappresentano i possibili eventi e ogni elemento della tabella contiene la coppia (azione, stato prossimo)

Tabella degli stati

- Altro approccio: utilizzare una tabella per rappresentare stati e transizioni

	CHAR_SIG	STAR_SIG	SLASH_SIG
code			do_nothing(), slash
slash	do_nothing(), code	a2(), comment	do_nothing(), cod
comment	a(1), comment	do_nothing(), star	a(1), comment
star	a2(), comment	a1(), star	a(2), code

- Ogni riga della tabella rappresenta uno stato, le colonne rappresentano i possibili eventi e ogni elemento della tabella contiene la coppia (azione, stato prossimo)

Tabella degli stati

- Altro approccio: utilizzare una tabella per rappresentare stati e transizioni

	CHAR_SIG	STAR_SIG	SLASH_SIG
code			do_nothing(), slash
slash	do_nothing(), code	a2(), comment	do_nothing(), cod
comment	a(1), comment	do_nothing(), star	a(1), comment
star	a2(), comment	a1(), star	a(2), code

- Ogni riga della tabella rappresenta uno stato, le colonne rappresentano i possibili eventi e ogni elemento della tabella contiene la coppia (azione, stato prossimo)
- Implementazione:
  - Azione: puntatore a funzioni membro
  - Stato prossimo: scalare che rappresenta lo stato

Codice

```
class StateTable {
public:
   typedef void (StateTable::*Action)();
   struct Tran {
      Action action;
      unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
      : myTable(table), myNsignals(nSignals), myNstates(nStates) {}
   virtual ~StateTable() {}                          // virtual xctor
   void dispatch(unsigned const sig) {
      register Tran const *t = myTable + myState*myNsignals + sig;
      (this->*(t->action))();
      myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```


Codice

```
class StateTable {
public:
   typedef void (StateTable::*Action)();
   struct Tran {
      Action action;
      unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
      : myTable(table), myNsignals(nSignals), myNstates(nStates) {}
   virtual ~StateTable() {}                          // virtual xctor
   void dispatch(unsigned const sig) {
      register Tran const *t = myTable + myState*myNsignals + sig;
      (this->*(t->action))();
      myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```


Codice

```
class StateTable {
public:
   typedef void (StateTable::*Action)();
   struct Tran {
      Action action;
      unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
      : myTable(table), myNsignals(nSignals), myNstates(nStates) {}
   virtual ~StateTable() {}                          // virtual xctor
   void dispatch(unsigned const sig) {
      register Tran const *t = myTable + myState*myNsignals + sig;
      (this->*(t->action))();
      myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```

Codice

```
class StateTable {
public:
   typedef void (StateTable::*Action)();
   struct Tran {
      Action action;
      unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
      : myTable(table), myNsignals(nSignals), myNstates(nStates) {}
   virtual ~StateTable() {}                          // virtual xctor
   void dispatch(unsigned const sig) {
      register Tran const *t = myTable + myState*myNsignals + sig;
      (this->*(t->action))();
      myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```


Codice

```
class StateTable {
public:
   typedef void (StateTable::*Action)();
   struct Tran {
      Action action;
      unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
      : myTable(table), myNsignals(nSignals), myNstates(nStates) {}
   virtual ~StateTable() {}                          // virtual xctor
   void dispatch(unsigned const sig) {
      register Tran const *t = myTable + myState*myNsignals + sig;
      (this->*(t->action))();
      myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```


Codice

```
class StateTable {
public:
   typedef void (StateTable::*Action)();
   struct Tran {
      Action action;
      unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
      : myTable(table), myNsignals(nSignals), myNstates(nStates) {}
   virtual ~StateTable() {}                          // virtual xctor
   void dispatch(unsigned const sig) {
      register Tran const *t = myTable + myState*myNsignals + sig;
      (this->*(t->action))();
      myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```


Codice

```
\\...

class CParser2 : public StateTable {        // CParser2 state machine
public:
   CParser2() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() { myCommentCtr = 0; myState = CODE; } // initial tran.
   long getCommentCtr() const { return myCommentCtr; }
private:
   void a1() { myCommentCtr += 1; }                  // action method
   void a2() { myCommentCtr += 2; }                  // action method
private:
   static StateTable::Tran const myTable[MAX_STATE][MAX_SIG];
   long myCommentCtr;                    // comment character counter
};

StateTable::Tran const CParser2::myTable[MAX_STATE][MAX_SIG] = {
   {{&StateTable::doNothing, CODE },
    {&StateTable::doNothing, CODE },
    {&StateTable::doNothing, SLASH}},
\\...
};
```

Codice

```
\\...

class CParser2 : public StateTable {        // CParser2 state machine
public:
   CParser2() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() { myCommentCtr = 0; myState = CODE; } // initial tran.
   long getCommentCtr() const { return myCommentCtr; }
private:
   void a1() { myCommentCtr += 1; }                  // action method
   void a2() { myCommentCtr += 2; }                  // action method
private:
   static StateTable::Tran const myTable[MAX_STATE][MAX_SIG];
   long myCommentCtr;                    // comment character counter
};

StateTable::Tran const CParser2::myTable[MAX_STATE][MAX_SIG] = {
   {{&StateTable::doNothing, CODE },
    {&StateTable::doNothing, CODE },
    {&StateTable::doNothing, SLASH}},
\\...
};
```


Codice

```
\\...

class CParser2 : public StateTable {        // CParser2 state machine
public:
   CParser2() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() { myCommentCtr = 0; myState = CODE; } // initial tran.
   long getCommentCtr() const { return myCommentCtr; }
private:
   void a1() { myCommentCtr += 1; }                  // action method
   void a2() { myCommentCtr += 2; }                  // action method
private:
   static StateTable::Tran const myTable[MAX_STATE][MAX_SIG];
   long myCommentCtr;                    // comment character counter
};

StateTable::Tran const CParser2::myTable[MAX_STATE][MAX_SIG] = {
   {{&StateTable::doNothing, CODE },
    {&StateTable::doNothing, CODE },
    {&StateTable::doNothing, SLASH}},
\\...
};
```


Codice

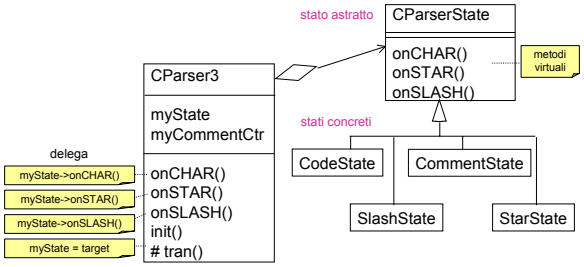
```
\\...

class CParser2 : public StateTable {        // CParser2 state machine
public:
   CParser2() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() { myCommentCtr = 0; myState = CODE; } // initial tran.
   long getCommentCtr() const { return myCommentCtr; }
private:
   void a1() { myCommentCtr += 1; }                  // action method
   void a2() { myCommentCtr += 2; }                  // action method
private:
   static StateTable::Tran const myTable[MAX_STATE][MAX_SIG];
   long myCommentCtr;                    // comment character counter
};

StateTable::Tran const CParser2::myTable[MAX_STATE][MAX_SIG] = {
   {{&StateTable::doNothing, CODE },
    {&StateTable::doNothing, CODE },
    {&StateTable::doNothing, SLASH}},
\\...
};
```


Vantaggi/svantaggi

- Aspetti positivi:
  - buone prestazioni per la gestione degli eventi (escludendo l'esecuzione dell'azione)
  - promuove la riusabilità del codice per un processore di eventi
  - la tabella degli stati è costante, per sistemi embedde può essere memorizzata in ROM
- Aspetti negativi:
  - la tabella degli stati è sparsa (spreco di spazio)
  - richiede un'inizializzazione abbastanza complicata

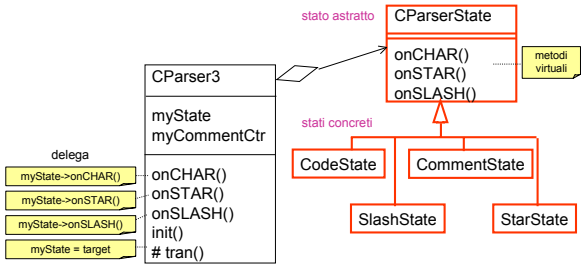
State Design Pattern

- Pattern basato su delega e polimorfismo



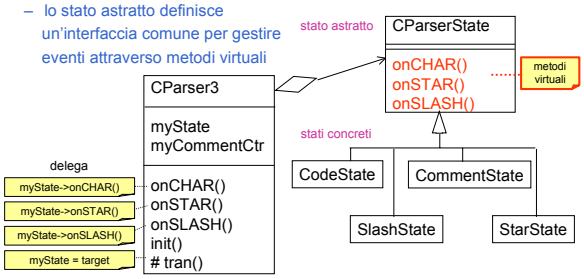
State Design Pattern

- Pattern basato su delega e polimorfismo
- Lo stato è rappresentato come un oggetto
- Gli stati concreti sono rappresentati come sotto-classi di uno stato astratto



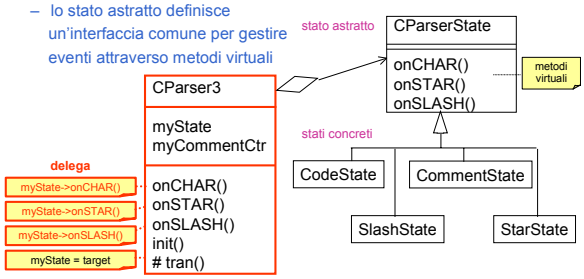
State Design Pattern

- Pattern basato su delega e polimorfismo
- Lo stato è rappresentato come un oggetto
- Gli stati concreti sono rappresentati come sotto-classi di uno stato astratto
  - lo stato astratto definisce un'interfaccia comune per gestire eventi attraverso metodi virtuali



State Design Pattern

- Pattern basato su delega e polimorfismo
- Lo stato è rappresentato come un oggetto
- Gli stati concreti sono rappresentati come sotto-classi di uno stato astratto
  - lo stato astratto definisce un'interfaccia comune per gestire eventi attraverso metodi virtuali



Codice

```
class CParser3;                 // Context class, forward declaration

class CParserState {                                // abstract State
public:
    virtual void onCHAR(CParser3 *context, char ch)  {}
    virtual void onSTAR(CParser3 *context) {}
    virtual void onSLASH(CParser3 *context) {}
};

class CodeState : public CParserState {      // concrete State "Code"
public:
    virtual void onSLASH(CParser3 *context);
};

//... altre classi (SlashState, CommentState, StarState)
```


Codice

```
class CParser3;                 // Context class, forward declaration

class CParserState {                                // abstract State
public:
    virtual void onCHAR(CParser3 *context, char ch)  {}
    virtual void onSTAR(CParser3 *context) {}
    virtual void onSLASH(CParser3 *context) {}
};

class CodeState : public CParserState {      // concrete State "Code"
public:
    virtual void onSLASH(CParser3 *context);
};

//... altre classi (SlashState, CommentState, StarState)
```


Codice

```
class CParser3 {                                    // Context class
   friend class CodeState;
   //...
   static CodeState    myCodeState;
   //...

   CParserState *myState;
   long myCommentCtr;

public:
   void init() { myCommentCtr = 0; tran(&myCodeState); }
   void tran(CParserState *target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
   void onCHAR(char ch) { myState->onCHAR(this, ch); }
   void onSTAR() { myState->onSTAR(this);  }
   void onSLASH() { myState->onSLASH(this); }
};
```


Codice

```
class CParser3 {                                    // Context class
   friend class CodeState;
   //...
   static CodeState    myCodeState;
   //...

   CParserState *myState;
   long myCommentCtr;

public:
   void init() { myCommentCtr = 0; tran(&myCodeState); }
   void tran(CParserState *target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
   void onCHAR(char ch) { myState->onCHAR(this, ch); }
   void onSTAR() { myState->onSTAR(this);  }
   void onSLASH() { myState->onSLASH(this); }
};
```

Codice

```
class CParser3 {                                    // Context class
   friend class CodeState;
   //...
   static CodeState    myCodeState;
   //...

   CParserState *myState;
   long myCommentCtr;

public:
   void init() { myCommentCtr = 0; tran(&myCodeState); }
   void tran(CParserState *target) { myState = target; }
   long getCommentCtr() const { return myCommentCtr; }
   void onCHAR(char ch) { myState->onCHAR(this, ch); }
   void onSTAR() { myState->onSTAR(this);  }
   void onSLASH() { myState->onSLASH(this); }
};
```

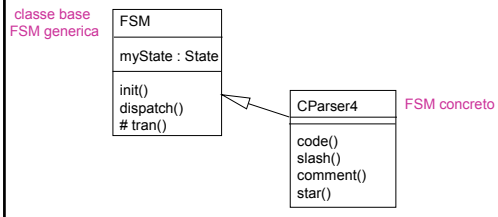

Codice

```
void CodeState::onSLASH(CParser3 *context) {
   context->tran(&CParser3::mySlashState);
}

void SlashState::onCHAR(CParser3 *context, char ch) {
   context->tran(&CParser3::myCodeState);
}

void SlashState::onSTAR(CParser3 *context) {
   context->myCommentCtr += 2;
   context->tran(&CParser3::myCommentState);
}

//...
```


Vantaggi/svantaggi

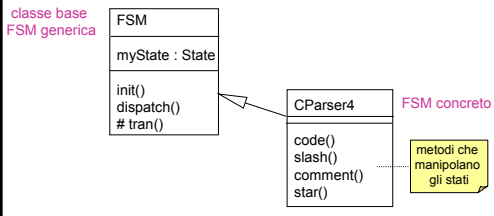
- Aspetti positivi:
  - definisce un'interfaccia generica e la specializza a seconda del comportamento
  - consente di ottenere le transizioni degli stati in modo efficiente (riassegnamento di un puntatore)
  - fornisce buone prestazione per la gestione degli eventi attraverso il meccanismo di late binding
  - è efficiente dal punto di vista della memoria (gli stati concreti senza attributi possono essere condivisi)
- Aspetti negativi:
  - compromette gli aspetti di incapsulamento delle classi: tutte le classi sono friend

Implementazione ottimale



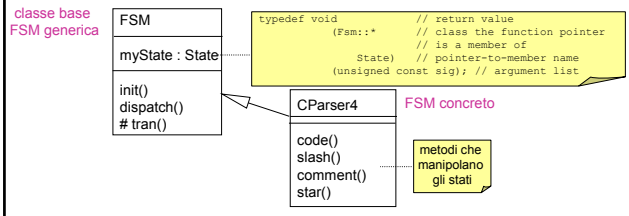
- L'implementazione combina elementi delle tre precedenti implementazioni, aggiungendo alcuni aspetti originali

Implementazione ottimale



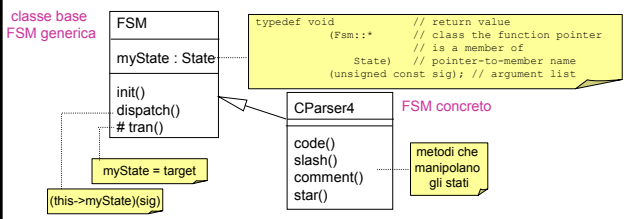
- L'implementazione combina elementi delle tre precedenti implementazioni, aggiungendo alcuni aspetti originali
- Aspetto innovativo: lo stato viene rappresentato come un metodo che manipola stati

Implementazione ottimale



- L'implementazione combina elementi delle tre precedenti implementazioni, aggiungendo alcuni aspetti originali
- Aspetto innovativo: lo stato viene rappresentato come un metodo che manipola stati
  - stato = puntatore ad una funzione membro

Implementazione ottimale



- L'implementazione combina elementi delle tre precedenti implementazioni, aggiungendo alcuni aspetti originali
- Aspetto innovativo: lo stato viene rappresentato come un metodo che manipola stati
  - stato = puntatore ad una funzione membro
  - transizione di stato --> assegnamento del puntatore

Vantaggi/svantaggi

- Aspetti positivi:
  - Semplice
  - Buone prestazioni per la gestione degli eventi, eliminando un livello di switch (sostituito con la dereferenziazione di un puntatore a funzione)
  - Efficiente transizione degli stati (assegnamento di un puntatore)
  - Scalabile e flessibile: è facile cambiare la topologia della macchina a stati

Informatica 3

Lezione 28 - Modulo 3

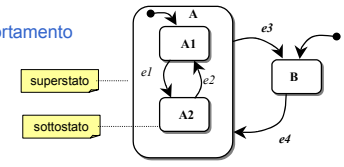
Statecharts

Statecharts

- Gli Statecharts (David Harel, 1987) sono macchine a stati estese con
  - stati gerarchici
  - stati concorrenti
  - azioni in ingresso ed in uscita

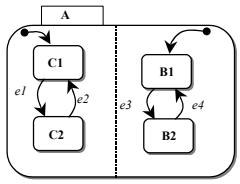
Stati gerarchici

- La gerarchia consente di realizzare il paradigma noto con il termine di *dividi et impera*
  - risolvere problemi complessi attraverso la loro scomposizione in sottoproblemi più semplici da risolvere.
- Forma di generalizzazione del concetto di stato ottenuta attraverso l'uso di diagrammi di stato *annidati*
- Un diagramma che ad alto livello si trova in un particolare stato dovrà trovarsi in un unico stato del diagramma annidato sottostante
  - relazione OR
- Riuso del comportamento



Stati concorrenti

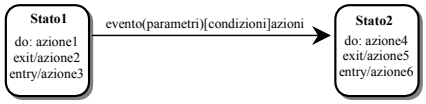
- Uno stato può contenere (aggregare) al suo interno diversi diagrammi a stati ognuno dei quali in grado di evolvere per conto proprio.
- Lo stato aggregante corrisponde alla combinazione di tutti gli stati dei diagrammi sottostanti e costituisce quella che viene chiamata una *relazione AND* tra gli stati.
  - Lo stato aggregato è equivalente allo stato del primo diagramma sottostante e (and) lo stato del secondo diagramma, ecc.



Azioni in ingresso e in uscita

- In uno Statechart tutti gli stati possono avere
  - azioni (opzionali) in ingresso, eseguite quando si entra nello stato
  - azioni (opzionali) in uscita, eseguite quando si esce dallo stato

  - Riassumendo:



Ereditarietà del comportamento

- Stati gerarchici vs. ereditarietà delle classi
  - nella programmazione OO l'ereditarietà tra classi è una relazione IS-A
    - Es. l'oggetto studente "è" appartenente alla classe persona --> tutte le operazioni che si applicano a persona si applicano anche a studente

  - tutte le caratteristiche di ereditarietà tra classi si applicano allo stesso modo anche agli stati annidati
    - gli stati annidati si basano sulla relazione di classificazione IS-A
    - la relazione IS-A diventa una relazione IS-IN
    - i sotto-stati ereditano il comportamento dei super-stati --> proprietà di *ereditarietà del comportamento*

Implementazione

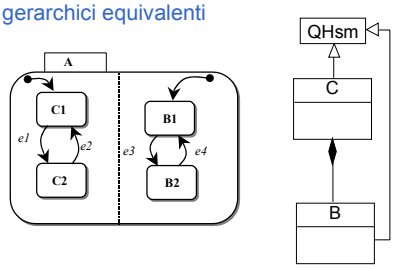
- Implementazione
  - negli FSM: stato --> puntatore a funzione membro

```
typedef void            // return value
          (Fsm::*       // class the function pointer
                        // is a member of
             State)     // pointer-to-member name
          (unsigned const sig); // argument list
```

  - negli Statecharts: il puntatore a funzione membro deve restituire il super-stato
    - si tratta di una definizione ricorsiva
      - typedef void (QHsm::*QPseudoState)(QEvent const *);
      - typedef QPseudoState (QHsm::*QState)(QEvent const *);

Stati concorrenti e stati gerarchici

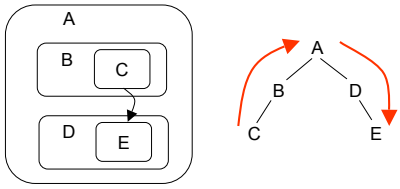
- Gli stati concorrenti possono essere trasformati in stati gerarchici equivalenti



- Soluzione: utilizzare la composizione di oggetti (relazione di aggregazione)

Azioni in ingresso e in uscita

- Quando si esegue una transizione vengono eseguite le azioni in uscita dello stato sorgente e dei suoi super-stati, fino al super-stato comune tra sorgente e destinazione; si eseguono quindi le azioni in ingresso di tutti i sotto-stati che portano allo stato di destinazione



Azioni in ingresso e in uscita

- Implementazione:
  - per uscire dalla configurazione dello stato corrente si segue la naturale direzione di navigazione attraverso gli stati
  - per entrare nello stato di destinazione occorre navigare nella direzione opposta
    - Soluzione: si registra il percorso di uscita (dallo stato di destinazione al super-stato comune senza eseguire alcuna azione)
      - si utilizza un segnale di evento riservato (empty) che fa in modo che ogni stato restituisca il proprio super-stato, senza altri effetti collaterali
      - una volta registrato il percorso di uscita lo si può utilizzare come percorso in ingresso, eseguendolo al contrario
    - Osservazione: le transizioni da uno stato ad un altro sono statiche (sorgente e destinazione della transizione non cambiano a tempo di esecuzione)
      - ottimizzazione: es. si calcola il percorso in ingresso solo la prima volta - viene memorizzato per le volte successive