Modellazione discreta, per eventi e per attività

Simulazione – Lezione n. 5

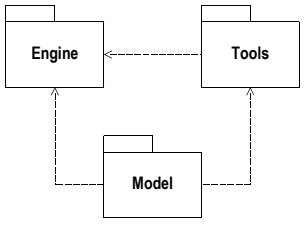
Corso di Laurea in Informatica Applicata
Università di Pisa, sede di La Spezia

Contenuti

- Modellazione discreta per eventi
- Motore a eventi
- Modellazione discreta per attività
- Motore ad attività
- Motore (e metodo) a tre fasi

Generica architettura di un sim

- Cominciamo a guardare dentro motore e modello



Motori e modelli, in generale

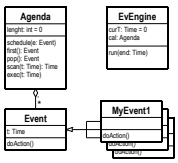
- Operazioni: eventi ed attività
  - Strutture dati per le cose da fare e i tempi a cui farle
  - Algoritmi per trovarle e farle al tempo corretto
  - Tutte e sole, preservando sequenza e contemporaneità
- Meccanismi di agenda
  - Programmare gli istanti interessanti (eventi)
  - Far scorrere il tempo
  - Accorgersi degli eventi correttamente
  - Eseguire le azioni per aggiornare lo stato del sistema (attività)
- Modello: specifica di istanti e azioni
- Nota: eventi/istanti, attività/azioni

Modellazione per (e motori a) eventi

- Metodo in uso fin dagli anni ’60
  - Proposto nel ’63 (Markowitz et al.)
  - Adottato nelle prime versioni di SIMSCRIPT
  - Largamente adottato fino agli anni ’80
- Simulatori efficienti e semplici
  - Modelli compatti
  - Codice, in generale, non facilmente manutenibile (!)
- Caratteristiche
  - Motore sequenziale
  - Evento: un istante in cui accade un cambiamento di stato
  - Azione: tutto ciò che succede nel sistema a un dato evento

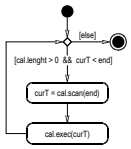
Motore a eventi, strutture OO

- L’agenda (lista dinamica)
  - Programmazione (inserimento)
  - Ricerca di eventi da far scattare
  - Esecuzione delle azioni
- Il motore
  - Scorrere il calendario, a esaurimento o fino al tempo dato
- Il modello
  - Eventi: classi da specializzare
  - Specializzazione dell’azione
  - Eventuale specializzazione di dati e metodi propri dell’evento



Motore a eventi, esecuzione

- Ricerca degli eventi
  - Il tempo avanza al primo evento in agenda antecedente la fine della simulazione
- Esecuzione delle azioni
  - Tutti gli eventi al tempo corrente scattano e sono rimossi
- Ciclo infinito
  - Fino alla fine della simulazione o all'esaurimento dell'agenda



Agenda: metodi scan() ed exec()

- Avanzamento del tempo

```
Time Agenda::scan(Time end) {
  if (lenght > 0) return min(end, first().t);
  else return end;
}
```

- Esecuzione di tutti gli eventi (contemporanei)

```
void Agenda::exec(Time curT) {
  while (lenght > 0 && first().t == curT)
    pop().doAction();
}
```

- Agenda: ordinata sul tempo (e non solo)

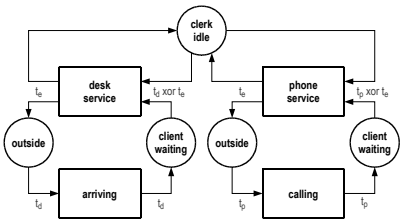
Motore: metodo run()

- Esecuzione di tutta la simulazione

```
void EvEngine::run(Time end) {
  while (curT < end && cal.lenght > 0) {
    curT = cal.scan(end);
    cal.exec(curT);
  }
}
```

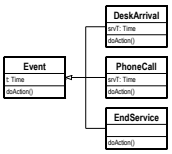
- Note
  - Eventi *esterni* ed *interni*
  - Esterni: non dipendono dal sistema, programmati prima
  - Interni: programmati (e generati) nei metodi doAction()
  - Classificazione da identificare durante la modellazione

Impiegato tormentato, per eventi



Ingredienti

- Tre eventi, da specializzare
  - Arrivi e chiamate, fine dei servizi
  - Tempo dell'evento, tempo di servizio
  - Metodi per aggiornare lo stato
- Due code
  - Sportello e centralino, `dQ`, `cQ`
  - Basta conservare il tempo di servizio
- Una variabile di stato
  - Per la disponibilità dell'impiegato, `ckSt`
- *Entia non sunt multiplicanda praeter necessitatem*



Eventi: arrivo e chiamata

- Arrivo di un cliente allo sportello

```
DeskArrival::doAction() {
  if (ckSt == idle) {
    ckSt = servingDesk;
    cal.schedule(new EndService(curT+this.srvT));
  } else
    dQ.push(this.srvT)
}
```

- Note
  - Priorità fra eventi
  - DeskArrival prima di PhoneCall
  - Necessaria quando l'impiegato è in ozio

Eventi: fine di un servizio

- Fine di un servizio

```
EndService::doAction() {
  if (!dQ.isEmpty()) {
    ckSt = servingDesk;
    cal.schedule(new EndService(curT+dQ.pop()));
  } else
  if (!cQ.isEmpty()) {
    ckSt = servingCall;
    cal.schedule(new EndService(curT+cQ.pop()));
  } else
    ckSt = idle;
}
```

- Priorità, rispetto arrivi e chiamate
  - Logicamente ininfluente, minime variazioni di efficienza

Note e considerazioni

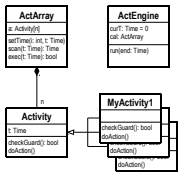
- Caratteristiche del motore a eventi
  - Efficiente, semplice ed estremamamente compatto
  - Codice praticamente scritto
  - Sono da sistemare visibilità, disallocazione, stampe
- Il metodo doAction()
  - Produce i cambiamenti di stato per tutto il sistema
  - Il codice può “toccare” tutte le entità
  - Cambiare un’entità implica modificare codice “condiviso”
  - Difficoltà di intervento e possibilità di errori
- Strutture dati dinamiche e precaricate

Modellazione per (e motore ad) attività

- Metodo in uso fin dagli anni ’60
  - Proposto in origine nel ’62 (Buxton & Laski)
  - Adottato nelle prime versioni di CSL (Esso & IBM)
  - Antenato del metodo delle tre fasi (con cui è a volte confuso)
- Partizionare la logica del modello
  - Confinare i comportamenti per avere maggior manutenibilità
  - Pagando qualcosa in termini di efficienza
- Caratteristiche
  - Motore sequenziale
  - Azioni semplici, “atomiche”, (~ eventi × condizioni)
  - Gli eventi sono gli istanti a cui le attività iniziano

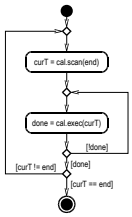
Motore ad attività, strutture OO

- L'agenda (vettore statico)
  - Programmazione (modifica)
  - Ricerca di attività da far scattare
  - Esecuzione delle attività abilitate
- Il motore
  - Scorrere il calendario, fino al tempo dato
- Il modello
  - Attività: classi da specializzare
  - Specializzazione dell'azione e della guardia
  - Eventuale specializzazione di dati associati all'attività



Motore ad attività, esecuzione

- Avanzamento del tempo
  - Il tempo avanza alla prima attività in agenda antecedente la fine della simulazione
- Esecuzione delle azioni
  - Tutte le attività al tempo corrente con guardie verificate scattano
  - Finché nessuna attività è verificata
- Ciclo infinito
  - Fino alla fine della simulazione (o fino a quando nessuna guardia è verificata)



Agenda: metodo scan()

- Avanzamento del tempo

```
Time ActArray::scan(Time end) {
  Time t = end;
  for (int i=0; i<n; i++) {
    t = min(t, a[i].time)
  }
  return t;
}
```

- Note
  - La ricerca richiede la scansione di tutta l'agenda, sempre
  - Nel motore a eventi il costo è nell'inserzione
  - Costo costante (attività) vs costo variabile (eventi)

Agenda: metodo exec()

- Esecuzione delle attività con guardia verificata

```
bool ActArray::exec(curT: Time) {
  bool r = true;
  for (int i=0; i<n; i++) {
    if (a[i].time == curT && a[i].checkGuard()) {
      a[i].doAction();
      r = false;
    }
  }
  return r;  // true: nothing triggered, all done
}
```

- Note
  - Ancora una scansione di tutta l'agenda
  - L'azione di un'attività potrebbe verificarne un'altra

Motore: metodo run()

- Esecuzione di tutta la simulazione

```
void ActEngine::run(Time end) {
  while (curT < end) {
    curT = cal.scan(end);
    done = false;
    while (!done)
      done = cal.exec(curT);
  }
}
```

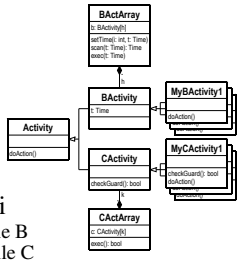
- Note
  - Scansioni multiple dell'agenda
  - Le azioni devono riprogrammare i tempi delle attività
  - In qualche modo anche quelli legati a eventi esterni

Il metodo delle tre fasi

- Inefficienza del motore ad attività
  - Valutazione del tempo e delle guardie eseguita più volte
  - Un'ottimizzazione che implica una prospettiva di analisi
- Tipi di attività
  - Attività di tipo B (bound, book-keeping):
    sappiamo quando accadono, dipendono solo dal tempo
  - Attività di tipo C (conditional, cooperative)
    dipendono da condizioni, a volte determinate da cooperazione
- Modello
  - Più frammentato, alcune attività si spezzano in una B e una C
  - Con l'abitudine, più naturale di quanto sembri

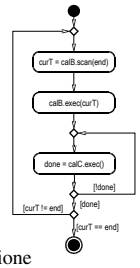
Motore a 3 fasi, strutture dati

- Due classi di attività
  - Metodo doAction() per tutte
  - Attività B, tempo
  - Attività C, guardia
- Tabelle separate
  - Programmazione e scansione solo per le B
  - Esecuzione per tutte
- Specializzazione delle azioni
  - Riprogrammazione dei tempi delle B
  - Cambiamenti sulle condizioni delle C



Motore a 3 fasi, esecuzione

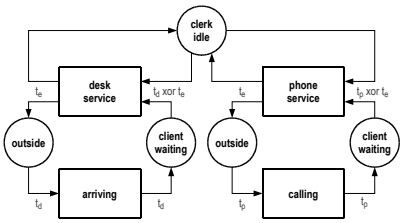
- Il tempo avanza alla prima B
- Esecuzione delle attività B
  - Al tempo corrente
  - Non hanno guardie
- Esecuzione delle attività C
  - Tutte le C le cui guardie sono verificate
  - Finché nessuna è verificata
- Terminazione
  - È stato raggiunto il tempo di fine simulazione



Considerazioni

- Modifiche del codice
  - Il metodo scan() non cambia
  - Il metodo exec() è diverso nei due casi (potato)
- Effetti sull'efficienza
  - La scan() è eseguita su un vettore più corto
  - La exec() sulle B è eseguita una volta sola
  - La exec() sulle C cicla, ma su un vettore più corto
- Note d'epoca
  - Implementazione OO con chiamate di metodi
  - In origine, tabelle per istruzioni di salto condizionato

Impiegato tormentato, per attività, a 3f



Identificazione delle attività

- Identificare gli stati “attivi”
  - Attività per entrare in uno stato attivo
  - Attività per concluderlo
- Attività dell’impiegato
  - ACkSvDb, ACkSvPb, inizio servizi, sportello e telefono
  - ACkSvDe, ACkSvPe, fine dei servizi, sportello e telefono
- Attività dei clienti
  - AClDArv, AClPCll, arrivo o telefonata del cliente
  - AClDEnd, AClPEnd, fine dei servizi, sportello e telefono
- Gli stati, comunque, conviene non dimenticarli

Specificare le attività

- Definire tempi e guardie
  - Attività B, dipendenti dal tempo
    - AClDArv, AClPCll: quando un cliente arriva
    - ACkSvDe, ACkSvPe: quando il servizio termina
    - AClDEnd, AClPEnd: quando un cliente è servito
  - Attività C, dipendenti da condizioni
    - ACkSvDb, ACkSvPb: se ci sono clienti in coda
- Definire le azioni
  - Modificare lo stato, le code, su cui valgono le condizioni
  - Riprogrammare i tempi delle attività B
  - Due code e una variabile per lo stato dell’impiegato

Vettori delle attività

- Vettore delle B

```
0 AClDArv  td                dQ.push(b[0].srvT);
1 AClPCll  tp                cQ.push(b[1].srvT);
2 ACkSvDe  te                ckSt = idle;
3 ACkSvPe  te                ckSt = idle;
4 AClDEnd  te                // maybe print data
5 AClPEnd  te                // maybe print data
```

- Vettore delle C

```
0 ACkSvDb  ckSt == idle      ckSt = servingDesk;
           && !dQ.isEmpty()  b[2].t = curT+dQ.pop();
1 ACkSvPb  ckSt == idle      ckSt = servingCall;
           && dQ.isEmpty()   b[3].t = curT+cQ.pop();
           && !cQ.isEmpty()
```


Considerazioni

- Note varie
  - L'impiegato va sempre idle
  - I clienti si mettono sempre in coda
  - Ordine in C significativo (una condizione è semplificabile)
  - Lo stato dell'impiegato va gestito, è una condizione
  - Servire significa ridefinire il tempo delle ACkSv*e
  - Attività inutili (possono essere utili per registrare dati)
- Un problema
  - Chi ridefinisce i tempi di AClDArv e AClPCll?
  - Strutture dati statiche, la simulazione deve ridefinire i dati
  - Soluzione: nelle azioni delle attività stesse
  - Generazione dei tempi di arrivo/chiamata e di servizio

Attività 3f vs eventi

- Diagrammi dei cicli di attività e metodo
  - Il metodo risulta naturale
  - C'è maggiore corrispondenza con il diagramma
- Implementazione del modello
  - Codice più frammentato (6 pezzi contro 3)
  - Non sempre disaccoppiato:
    le attività "dell'impiegato" ACkSv*b devono conoscere
    le attività "del cliente" Acl*End per ridefinirne i tempi
- Generazione dei dati
  - Una necessità da trasformare in una buona idea

Sim UniPisa LaSpezia

Riferimenti

- G. Gallo, *Note di simulazione*, cap. 2
- M. Pidd, *Computer Simulation in Management Science*, capp. 5 e 6