Progettazione del Software

Giuseppe De Giacomo & Massimo Mecella
Dipartimento di Informatica e Sistemistica
SAPIENZA Università di Roma

Diagramma degli stati e delle transizioni: concorrenza

Progetto e realizzazione di classi con associato diagramma degli stati e delle transizioni nel caso di concorrenza

Progetto e realizzazione di diagrammi stati e transizioni in presenza di concorrenza

- Ora andiamo a studiare classi con associato un diagramma degli stati e delle transizioni in presenza di attività concorrenti.

- La classe naturalmente è una classe legata ad altre classi secondo il diagramma delle classi. Quindi vale tutto ciò che è stato detto in precedenza, relativamente alla rappresentazione degli attributi, alla partecipazione ad associazioni, alla responsabilità sulle associazioni stesse, ecc.

- In più ci si dovrà occupare del suo aspetto "reattivo" come modellato dal diagramma degli stati e delle transizioni, tenendo presente che l'attività di ricezione elaborazione e invio di eventi è solo una delle attività della applicazione che agiscono sul diagramma delle classi.

Progetto e realizzazione di diagrammi stati e transizioni in presenza di concorrenza

- Di fatto andremo ad associare a ciascun oggetto reattivo un thread separato per la gestione degli eventi.

- Avremo quindi un applicazione in cui conviveranno thread dedicati alle attività del diagramma delle attività e thread dedicati alla gestione degli eventi.

- Questo richiederà da una parte la realizzazione di un environment più sofisticato.

- Dall'altra una gestione degli stati e delle transizioni che gestisca la concorrenza in modo opportuno (come già facciamo per le attività del diagramma delle attività).

Progetto e realizzazione

- Per rappresentare tale l'aspetto reattivo secondo il diagramma degli stati e delle transizioni dobbiamo:
  - Rappresentare gli stati
  - Rappresentare le transizioni
    - Rappresentare gli eventi
    - Rappresentare le condizioni
    - Rappresentare le azioni

- *Come nel caso non concorrente considereremo eventi come messaggi che vengono scambiati reciprocamente tra i vari oggetti reattivi.*

Realizzazione degli stati

- Tipicamente rappresenteremo lo stato di un oggetto reattivo (con associato diagramma degli stati e delle transizioni) facendo uso di una specifica **rappresentazione degli stati** del diagramma
- Scelte tipiche sono:
  - una **enumerazione Java**: per costruire una costante per ogni stato del diagramma associando alle stesse un tipo (l'enumerazione stessa).

  - Una serie di **constanti intere** individuali, una per ciascuno stato del diagramma (questa soluzione è peggiore della prima perché Java non associa a queste costanti un tipo specifico ma solo il tipo intero).

Realizzazione degli stati

- Altre scelte sono possibili:
  - L'uso diretto dei valori assunti dagli attributi dell'oggetto (ma è raro che questo sia possibile).
  - Una rappresentazione booleana degli stati (come attraverso flip-flop cfr. Corso di Calcolatori Elettronici) – utile per esempio, quando gli stati sono costituiti da variabili associate a specifici dispositivi.

- *Noi faremo praticamente sempre uso di **enumerazioni**.*

Realizzazione degli stati

- Accanto alla rappresentazione degli stati avremo una specifica **variabile di stato** che contiene lo **stato corrente** dell'oggetto. Tale variabile sarà del tipo scelto per rappresentare gli stati del diagramma, quindi:
  - una variabile di tipo enumerazione, se gli stati sono rappresentati da una enumerazione;
  - una variabile intera, i cui valori ammissibili sono solo quelli associati alle costanti corrispondenti agli stati, nel caso gli stati sono rappresentati da costanti intere.

- Inoltre se necessario si farà uso di eventuali **variabili di stato ausiliarie** per memorizzare dati necessari durante le transizioni (che ovviamente non siano già rappresentati nei campi dato dell'oggetto corrispondenti agli attributi del diagramma delle classi).

Gestione degli stati e delle transizioni con concorrenza

```
public class Giocatore implements Listener {

  // gestione stato

  public static enum Stato {
    ALLENAMENTO, INGIOCO, FINEGIOCO
  }

  Stato statocorrente = Stato.ALLENAMENTO; //nota visibilita' package

  double trattopercorso; // nota visibilita' package

  public Stato getStato() {
    return statocorrente;
  }

  public void fired(Evento e) {
    Executor.perform(new GiocatoreFired(this, e));
  }
}
```

Giocatore implementa Listener, dichiarando implicitamente di essere un oggetto attivo (con fired)

Rappresentazione degli stati come enumerazione "Giocatore.Stato"

La rappresentazione dello stato non è essenzialmente cambiata.

Variabile di stato per denotare lo stato corrente.
Si noti l'inizializzazione con lo stato iniziale (qui fatta a tempo di compilazione, poteva anche essere fatta dal costruttore)

Eventuali variabili di stato ausiliarie (private) da usare nella gestione delle transizioni

Funzione per conoscere lo stato corrente secondo il diagramma degli stati e delle transizioni

Gestione degli stati e delle transizioni con concorenza

```
public class Giocatore implements Listener {

  // gestione stato

  public static enum Stato {
    ALLENAMENTO, INGIOCO, FINEGIOCO
  }

  Stato statocorrente = Stato.ALLENAMENTO; //nota visibilita' package

  double trattopercorso; // nota visibilita' package

  public Stato getStato() {
    return statocorrente;
  }

  public void fired(Evento e) {
    Executor.perform(new GiocatoreFired(this, e));
  }
}
```

Si noti che le variabili di stato e variabili di stato ausiliarie hanno una visibilità a livello di package.

La funzione fired è cambiata:
- *Non restituisce eventi*
- *Delega la gestione delle transizioni ad un Funtore!*

Interfaccia Listener

```
public interface Listener {
   public void fired(Evento e);
}
```

L'interfaccia Listener prevede la sola funzione fired() ...
... che dato un evento esegue la transizione e eventualmente restituisce un nuovo evento
o null se il nuvo evento non c'e'

```
public class OggettoConStato implements Listener {
   ...
  // Gestine delle transizioni
   public void fired(Evento e) {
        ...
  }
}
```

Ogni oggetto reattivo implementa Listener, mettendo a disposizione una implementazione opportuna di fired()

Gestione delle transizioni

- La gestione delle transizioni avviene nella funzione **fired()**.
- Questa prende come parametro **l'evento scatenante** della transizione ~~e restituisce in uscita il **nuovo evento** lanciato dalla azione della transizione, oppure **null** in caso l'azione non lanci eventi~~
- Si noti:
  - Se **fired()** restituisce un evento come output della funzione a quale thread lo rende disponibile? A quello corrente soltanto!!!
  - Invece noi dobbiamo renderlo disponibile a oggetti che lavorano su thread separati. Per fare ciò faremo uso di una esplicita istruzione di **inserimento dell'evento nell'environment**:

    Environment.aggiungiEvento(new Evento(...));

- Il corpo della funzione **fired()** deve essere eseguito concorrentemente agli altri thread e può accedere al diagramma delle classi! Quindi deve essere è costituito da una **chiamata** all'esecuzione di un **funtore di tipo Task**.

Gestione delle transizioni

- Il funtore che realizza **fired()** associato ad una classe *NomeClasse* lo chiameremo sempre ***NomeClasse*Fired**.

- Questo funtore non deve essere acceduto dai clienti della classe perché contiene codice di interesse solo per la classe stessa: per fare ciò gli diamo visibilità a livello di package e lo metteremo nel package della classe (che invece è pubblica).

- Il codice del funtore deve poter accedere alle variabili di stato in generale, ecco perché ora queste hanno visibilità a livello di package (invece che private).

Gestione delle transizioni

- Il corpo funtore (cioè la funzione **esegui()**) fa quello che nel caso non concorrente fa direttamente **fired()**. Cioè è costituito da un **case** sullo stato corrente che definisce come si risponde all'evento in ingresso:
  - Controlla la **rilevanza dell'evento**;
  - Controlla la **condizione** che seleziona la transizione;
  - Prende gli eventuali **parametri dell'evento**,
  - Fa eventualmente **side-effect** sulle proprietà (campi dati) dell'oggetto;
  - Crea e il **nuovo evento** da mandare e ~~lo restituisce~~ lo inserisce nell'environment con un'istruzione del tipo:
    Environment.aggiungiEvento(new Evento(…));

Gestione delle transizioni: codice

```
class GiocatoreFired implements Task {
 private boolean eseguita = false;
 private Giocatore g;
 private Evento e;

 public GiocatoreFired(Giocatore g, Evento e) {  this.g = g; this.e = e; }

public synchronized void esegui(Executor exec) {
  if (eseguita || exec == null || (e.getDestinatario() != g && e.getDes          () != null))
   return;
  eseguita = true;
  switch (g.getStato()) {
  ...
  case INGIOCO:
   if (e.getClass() == Bastone.class)
    if (g.trattopercorso < 100) {
      ...
      Environment.aggiungiEvento(new Bastone(g, g));
      g.statocorrente = Stato.INGIOCO;
    } else { // trattopercorso >= 100
      ...
    }
   break;
  ...
  default:
   throw new RuntimeException("Stato corrente non riconosciuto.");
  }
 }

public synchronized boolean estEseguita() { return eseguita; }
}
```

Implementa il pattern funtore ed in particolare l'interfaccia Task.

Gestisce gli eventi rilevanti nello stato corrente con esattamente la stessa logica del caso non concorrente.

Mette l'evento nell'environment.

Supporto per lo scambio degli eventi: pattern observable-observer

- Lo scambio degli eventi segue sostanzialmente il **pattern Observable-Observer**:
  - Un oggetto **observable** registra, attraverso la funzione **addListener()**, i sui **observer** (chiamati tipicamente Listener i Java)
  - Ogni observer, per ragioni storiche tipicamente chiamato Listener in Java, implementa una speciale funzione qui chiamata **fired()**
  - Quando l'**observable** vuole **notificare** qualcosa chiama su ciascun **observer** registrato (memorizzato in un insieme) il suo metodo **fired()**.
- Si noti che la comunicazione observable-observer è unidirezionale: l'observable comunica (chiamando fired()) ai suoi observer l'avvenimento di qualcosa.

Supporto per lo scambio degli eventi

- Nel nostro caso l'idea generale del pattern **observable-observer** va adattata in modo opportuno, visto che tutti gli oggetti reattivi ricevono eventi ma anche lanciano eventi (**comunicazione bidirezionale**).

- Per realizzare tale comunicazione bidirezionale faremo uso di un particolare oggetto **environment**, che agisce da canale di comunicazione:
  - Tutti gli **oggetti reattivi manderanno all'environment** i propri eventi
  - **L'environment si occuperà di inoltrare ciascun evento** al giusto destinatario (che, si ricorda, è scritto sull'evento stesso).

Supporto per lo scambio degli eventi

- Relativamente all'environment faremo le seguenti assunzioni:
  - L'environment lancia ad ogni turno un evento per observer (o meglio Listener);
  - Per ciascun Listener l'environment garantisce che l'ordine di inoltro dei messaggi è l'ordine di arrivo degli stessi.

- Per fare ciò l'environment deve essere dotato di **una coda di eventi per ciascun Listener**:
  - Avere una struttura dati separata per ciascun Listener garantisce la gestione indipendente di ciascun Listener e la possibilià di lanciare un evento per ciascun Listener ad ogni passo
  - Il fatto che tale struttura dati sia una coda garantisce l'ordinamento giusto dei messaggi

Environment

Nota!

```
public final class Environment { // NB con final non si possono definire sottoclassi

        private Environment() { // NB non si possono costruire oggetti Environment
        }

        private static ConcurrentHashMap<Listener, LinkedBlockingQueue<Evento>> codeEventiDeiListener =
                new ConcurrentHashMap<Listener, LinkedBlockingQueue<Evento>>();

        public static void addListener(Listener lr, EsecuzioneEnvironment e) {
                if (e == null) return;
                codeEventiDeiListener.put(lr, new LinkedBlockingQueue<Evento>());
                // Nota Listener inserito ma non attivo
        }

        public static Set<Listener> getInsiemeListener() {
                return codeEventiDeiListener.keySet();
        }

        public static void aggiungiEvento(Evento e) {...}

        public static Evento prossimoEvento(Listener lr)
                 throws InterruptedException {
                // nota NON deve essere synchronized!!!
                return codeEventiDeiListener.get(lr).take();
        }

}
```

Tutti devono avere accesso all'environment sempre! L'environment stesso gestisce accesso concorrente alle proprie strutture dati attraverso ConcurrentHashMap e soprattutto LinkedBlockingQueue

Environment

Ha la stessa logica di aggiungiEvento nel caso senza concorrenza…

```
...
        public static void aggiungiEvento(Evento e) {
                // unico meccanismo per aggiungere eventi
                if (e == null) return;
                Listener destinatario = e.getDestinatario();
                if (destinatario != null && codeEventiDeiListener.containsKey(destinatario)) {
                        // evento per un destinatario attivo
                        try {
                                codeEventiDeiListener.get(destinatario).put(e);
                        } catch (InterruptedException e1) {
                                e1.printStackTrace();
                        }
                } else if (destinatario == null) {
                        // evento in broadcasting
                        Iterator<Listener> itn = codeEventiDeiListener.keySet().iterator();
                        while (itn.hasNext()) {
                                Listener lr = itn.next();
                                try {
                                        codeEventiDeiListener.get(lr).put(e);
                                } catch (InterruptedException e1) {
                                        e1.printStackTrace();
                                }
                        }
                }
        }
...
```

… ma ora utilizza code bloccanti(!!!) e map che permettono l'accesso concorrente

Environment

La classe EsecuzioneEnvironment è equipaggiata con tre metodi, rispettivamente per aggiungere Listener , per far partire tutti i listener, per fermare tutti i listener.

Queste funzioni però possono essere usate solo in certi stati, vedi diagramma degli stati (qui usato per limitare le invocazioni hai metodi e non per scambiare eventi.

```
/* Serve ad aggiungere i singoli listener, ed ad attivarli e disattivali (tutti insieme)
 * Nota: implementa il seguente diagramma degli stati e delle transizioni :
 *
 *   addListener
 *
 *      __
 *     |  |   --attivaListener->
 *  --> Attesa                 Esecuzione
 *          <--disattivaListener-
 */
public final class EsecuzioneEnvironment { //NB con final non si possono definire sottoclassi
        private EsecuzioneEnvironment() {
        }

        public static enum Stato {
                Attesa, Esecuzione
        };

        private static Stato statocorrente = Stato.Attesa;
        private static ConcurrentHashMap<Listener, Thread> listenerAttivi = null;
        ...
```

I Listener sono eseguiti in thread separati

Faremo uso di un evento speciale Stop che useremo per segnalare ai vari listener di terminare.

Environment

```
public static synchronized void addListener(Listener lr) {
        if (statocorrente == Stato.Attesa) {
                Environment.addListener(lr, new EsecuzioneEnvironment());
                //NB: Listener inserito ma non attivo
        }
}

public static synchronized void attivaListener() {
        if (statocorrente == Stato.Attesa) {
                statocorrente = Stato.Esecuzione;
                System.out.println("Ora attiviamo i listener");
                listenerAttivi = new ConcurrentHashMap<Listener, Thread>();
                Iterator<Listener> it = Environment.getInsiemeListener().iterator();
                while (it.hasNext()) {
                        Listener listener = it.next();
                        listenerAttivi.put(listener, new Thread(
                                        new   EsecuzioneListener(listener)));
                }
                Iterator<Listener> i = listenerAttivi.keySet().iterator();
                while (i.hasNext()) {
                        Listener l = i.next();
                        listenerAttivi.get(l).start();
                }
        }
}
```

Attiva tutti i Listener presenti nell'environment.

Environment

Disattiva tutti i Listener presenti nell'environment, mandandogli l'evento stop.

```
        public static synchronized void disattivaListener() {
                if (statocorrente == Stato.Esecuzione) {
                        statocorrente = Stato.Attesa;
                        System.out.println("Ora fermiano i listener");
                        Environment.aggiungiEvento(new Stop(null, null));
                        // NB: a questo punto i listener non sono ancora fermi
                        // ma l'evento Stop e' stato inserito nella coda di ciascuno di loro
                        // e questo evento quando processato li disattivera'
                        Iterator<Listener> it = listenerAttivi.keySet().iterator();
                        while (it.hasNext()) {
                                Listener listener = it.next();
                                try {
                                        Thread thread = listenerAttivi.get(listener);
                                        thread.join();
                                } catch (InterruptedException e1) {
                                        e1.printStackTrace();
                                }
                        }
                }
        }

        public static synchronized Stato getStato() {
                return statocorrente;
        }
}
```


Environment

Esegue ciascun Listener in un thread separato

Stop è un evento speciale che serve a fare terminare il thread.

```
class EsecuzioneListener implements Runnable { //NB: non e' pubblica, serve solo nel package
        private boolean eseguita = false;
        private Listener listener;

        public EsecuzioneListener(Listener l) {
                listener = l;
        }

        public synchronized void run() {
                if (eseguita) return;
                eseguita = true;
                while (true) {
                        try {
                                Evento e = Environment.prossimoEvento(listener);
                                if (e.getClass() == Stop.class) return;
                                listener.fired(e);
                        } catch (InterruptedException ex) {
                                ex.printStackTrace();
                        }
                }
        }

        public synchronized boolean estEseguita() {
                return eseguita;
        }
}
```

Esempio Staffetta con attività concorrenti

- Si veda il codice allegato.

- *Nota il codice allegato va considerato parte integrante di queste slide e va studiato e compreso interamente.*