Progettazione del Software

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

Diagramma degli stati e delle transizioni
Realizzazione

Realizzazione di classi con associato diagramma degli stati e delle transizioni

Realizzazione di oggetti reattivi in Java

- Innanzitutto una classe con associato un diagramma degli stati e delle transizioni è una classe legata allo 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.

Realizzazione di oggetti reattivi in Java

- Per rappresentare tale l’aspetto reattivo secondo il diagramma degli stati e delle transizioni dobbiamo fare in modo che gli oggetti reattivi si scambino eventi.

- Per fare ciò utilizzeremo il pattern Observable-Observer dove l’Observable è un oggetto di supporto speciale chiamato **Environment**.

- Gli oggetti reattivi per tanto implementeranno tutti una speciale interfaccia Java che chiameremo semplicemente **Listener**.

Realizzazione di oggetti reattivi in Java

- Iniziamo a descrivere tali oggetti reattivi approfondendo nell'ordine:
  - Come rappresentare in Java gli **eventi**
  - Come rappresentare in Java gli **stati**
  - Come rappresentare in Java le **transizioni**
- Poi illustreremo la realizzazione dell'**Environment.**
  - Inizialmente assumeremo ci concentreremo su una **realizzazione sequenziale dell'Environment** in cui tutti gli oggetti reattivi si scambiano eventi all'interno di un singolo thread.
  - Dopo vedremo una **realizzazione concorrente dell'Environment**

.

Realizzazione degli eventi

- Gli eventi sono rappresentati da oggetti di una classe Java **Evento**.
- La classe **Evento** rappresenta eventi generici, dotati di mittente e destinatario.
  - Quando il **destinatario** è **null** allora l'evento è in **broadcasting**;
  - Quando il **mittente** è **null** allora il **mittente non si è dichiarato** (rimanendo nascosto).
- Tutti gli eventi (dei diagrammi degli stati e delle transizioni) specifici per una data applicazione sono istanze di **classi derivate da Evento**.
- Gli oggetti (la cui classe più specifica è) **Evento** vengono usati direttamente solo per abilitare transizioni in casi particolari (esempio quando le transizioni non hanno un evento scatenate).

Realizzazione degli eventi

- Dal punto di vista tecnico la classe **Evento** e le sue derivate rappresentano **valori** (gli eventi / i messaggi scambiati): di fatto sono record che contengono riferimenti al mittente ed al destinatario, più (nelle classi derivate) eventuali parametri.

- Realizziamo quindi gli eventi come oggetti Java **immutabili** *(astrazione di valore realizzati con schema funzonale)* i cui metodi pubblici non fanno side-effect.

Realizzazione degli eventi

- Definiamo un opportuno **costruttore** che inizializza l'oggetto con tutte le informazioni necessarie

- Definiamo i metodi **get** per restituire dette informazioni

- Ridefiniamo **equals()** e **hashCode()** in modo che oggetti con le stesse informazioni risultino uguali

- *Non ridefiniamo clone() visto che sono oggetti immutibili e quindi possiamo avere condivisione di memoria.*

La classe Evento

```
public class Evento {
   private Listener mittente;
   private Listener destinatario;

   public Evento(Listener m, Listener d) {
      mittente = m;
      destinatario = d;
   }

   public Listener getMittente() { return mittente; }
   public Listener getDestinatario() { return destinatario; }

   public boolean equals(Object o) {
      if (o != null && getClass().equals(o.getClass())) {
         Evento e = (Evento) o;
         return mittente == e.mittente &&
                destinatario == e.destinatario;
      }
      else return false;
   }

  public int hashCode() { return mittente.hashCode() + destinatario.hashCode(); }
}
```

Chiamiamo listener gli oggetti che si scambino messaggi (vedi dopo)

m == null se non rilevante
d == null se evento in broadcasting

Esempio di ridefinizione di Evento

```
public class MioEvento extends Evento {

   private Object info;

   public MioEvento (Listener m, Listener d, Object i) {
      super(m,d);
      if (i == null) throw RuntimeException("Payload del messaggio mancante.");
      info = i;
   }
   public Object getInfo() { return info; }
   public boolean equals(Object o) {
      if (super.equals(o)) {
         MioEvento e = (MioEvento ) o;
         return info == e.info;
      }
      else return false;
   }
  public int hashCode() { return super.hashCode() + info.hashCode(); }

}
```

Gli eventi del diagramma degli stati e delle transizioni possono avere parametri, es info

Si noti la definizione dell'equals() che rimanda alla classe padre i controlli di base: oggetto o diverso da null e della stessa classe più specifica dell'oggetto di invocazione

Si noti l'uso di super

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

- Per rappresentare lo **stato corrente** avremo una specifica **variabile di stato** nell'oggetto reattivo.

- 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.

*Come detto noi faremo praticamente sempre uso di **enumerazioni**.*

Realizzazione degli stati

- Accanto alla **variabile di stato** per rappresentare lo **stato corrente,** se necessario si farà uso di eventuali **variabili di stato ausiliarie** per memorizzare dati necessari durante le transizioni

Nota, le variabili di stato ausiliarie servono a rappresentare informazioni necessarie alle transizioni che non siano già rappresentate nei campi dato dell'oggetto corrispondenti agli attributi del diagramma delle classi.

Rappresentazione dello stato in Java: codice

```
public class MioOggettoConStato  implements Listener {

   //Tutto l' oggetto secondo metodologia
   //piu' :

   //Gestione dello stato

   public static enum Stato {STATO_0, STATO_1, STATO_2, /*...*/ STATO_n}

   private Stato statocorrente = Stato.STATO_0;

   private Object varAux = null;

   public Stato getStato() {
      return statocorrente
   }

  //Gestione delle transizioni
  ...
}
```

Chiamiamo Listener gli oggetti che si scambino messaggi (vedi dopo)

Rappresentazione degli stati come enumerazione "MioOggettoConStato.Stato"

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 delle transizioni

- La gestione delle transizioni avviene in una funzione specifica **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.
  - La funzione **fired()** è specificata dall'interfaccia **Listener**.

Interfaccia Listener

```
public interface Listener {
   public Evento 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 Evento fired(Evento e) {
       ...
  }
}
```

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

Gestione delle transizioni

- Il corpo della funzione **fired()** è costituito da un **case** sullo stato corrente che definisce come si risponde all'evento passato come parametro di ingresso:

- In particolare per ogni **stato** (condizione nel case)
  - Controlla la **rilevanza dell'evento**;
  - Prende gli eventuali **parametri dell'evento**;
  - Controlla la **condizione** che seleziona la transizione tra quelle disponibili in nello stato stesso;
  - Esegue l'azione associata alla transizione:
    - Fa eventualmente **side-effect** sulle proprietà dell'oggetto e in generale di tutto la realizzazione del diagramma delle classi;
    - Crea e il **nuovo evento** da mandare e lo restituisce come risultato di **fired()**.

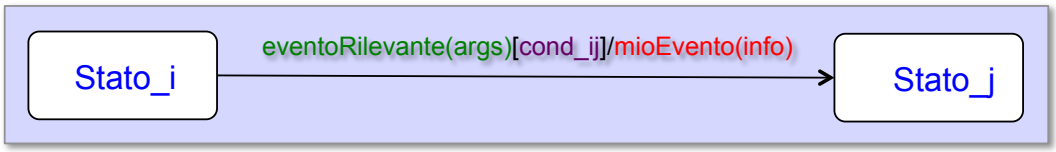
Gestione delle transizioni: codice

```
public class OggettoConStato implements Listener {
   ...
   //Gestine delle transizioni
   public Evento fired(Evento e) {
     if (e.getDestinatario() != this && e.getDestinatario() != null) return null;
     Evento nuovoevento = null;
     switch(statocorrente) {
     ...
     case Stato.STATO_i:
       if (e.getClass() == EventoRilevante.class)
        if (Cond_ij) {
          //fai qualcosa con l'evento: eventualmente con (EventRilevante) e.getArgs()
          nuovoevento = new MioEvento (this,destinatario,info);
          statocorrente = Stato.STATO_j;
         }
        break;
      ...
      default: throw new RuntimeException("Stato corrente non riconosciuto.");
     }
     return nuovoevento;
  }
}
```

filtra eventi non per this e non sono in broadcasting

Gestisce gli eventi rilevanti nello stato corrente

se la transizione non genera eventi lasciamo nuovo evento a null



Supporto per lo scambio degli eventi

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

- Per fare ciò l'environment deve essere dotato di **una coda di eventi separata 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

```
public class Environment {
 private HashMap<Listener, LinkedList<Evento>> codeEventiDeiListener;

 public Environment() {
  codeEventiDeiListener = new HashMap<Listener, LinkedList<Evento>>();
 }

 public void addListener(Listener lr) {
  codeEventiDeiListener.put(lr, new LinkedList<Evento>());
 }

 public void aggiungiEvento(Evento e) {
  // aggiunge evento e nella coda del destinatario di e
  // Pre: e <> null e.getDestinatario ha una coda associata in codeEventiDeiListener
   ...
 }

public void eseguiEnvironment() {
  // finche' ci almeno un evento da processare in una delle code
  // prendi tutti i primi elementi e mandali ai rispettivi Listener
   ...
        Evento ne = listener.fired(e); // chiamata all'esecuzione del listener
   ...
 }
}
```

Map che associa ad ogni Listener registrato una coda specifica

Funzione per registrare Listener

Funzione per aggiungere eventi anche esogeni (esterni)

Serve anche creare un evento iniziale per fare partire tutto il sistema

Manda in esecuzione il sistema

Invoca l'esecuzione di fired() dei Listener

Environment

```
public class Environment {
 private HashMap<Listener, LinkedList<Evento>> codeEventiDeiListener;

 public Environment() {
  codeEventiDeiListener = new HashMap<Listener, LinkedList<Evento>>();
 }
 public void addListener(Listener lr) {
  codeEventiDeiListener.put(lr, new LinkedList<Evento>());
 }
 public void aggiungiEvento(Evento e) {
  // aggiunge evento e nella coda del destinatario di e
  // Pre: e <> null e.getDestinatario ha una coda associata in codeEventiDeiListener
  Listener destinatario = e.getDestinatario();
  if (destinatario != null)
   // il messaggio e' per un destinatario specifico
   codeEventiDeiListener.get(destinatario).add(e);
  else {
   // destinatario == null significa che il messaggio e' in broadcasting
   Iterator<Listener> itn = codeEventiDeiListener.keySet().iterator();
   while (itn.hasNext()) {
    Listener lr = itn.next();
    codeEventiDeiListener.get(lr).add(e);
   }
  }
 }
  ...
```

Map che associa ad ogni Listener registrato una coda specifica

Funzione per registrare Listener

Funzione per aggiungere eventi anche esogeni (esterni)

Serve anche creare un evento iniziale per fare partire tutto il sistema

Environment

```
public class Environment {
 ...
  public void eseguiEnvironment() {
    boolean eventoProcessato;
    do {
      // finche' ci almeno un eventto da processare in una delle code
      // prendi tutti i primi elementi e mandali ai rispettivi Listener
      eventoProcessato = false;

      Iterator<Listener> it = codeEventiDeiListener.keySet().iterator();
      while (it.hasNext()) {
        Listener listener = it.next();
        LinkedList<Evento> coda = codeEventiDeiListener.get(listener);
        if (coda.isEmpty())
          continue;
        eventoProcessato = true;
        Evento e = coda.remove(0);

        Evento ne = listener.fired(e); // chiamata all'esecuzione del listener

        if (ne == null)
          continue;
        aggiungiEvento(ne); // aggiunge evento ne nella coda del destinatario di ne
      }
    } while (eventoProcessato);
  }
}
```

Manda in esecuzione il sistema

Chiamata all'esecuzione del Listener: aggiunge evento ne nella coda del destinatario di ne (se ne != null)

Esempio PlayList - Player

- Si veda il codice allegato.

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

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).

Realizzazione di oggetti reattivi concorrenti

- Come precedentemente avremo una rappresentazione in Java di:
  - **Eventi** e **stati** che sarà identica a prima.
  - **Transizioni** che sarà invece diversa per sfruttare la concorrenza.

- La realizzazione dell'**Environment sarà completamente concorrente.**
  - Ogni **oggetto reattivo** esegue il suo diagramma stati transizioni su un thread separato.
  - Anche l'**Environment** lavora su un thread separato.
  - L'Environment è dotato di una **struttura dati condivisa thread-safe** per lo scambio degli eventi. Tale struttura dati è accessibile dai thread degli oggetti reattivi oltre che dal thered dell'environment stesso.

Interfaccia Listener (modificata per concorrenza)

Per prima cosa modificare l'interfaccia **Listener**, rimuovendo l'evento restituito.

Gli eventi generati (che ora potranno essere 0, uno, o più) saranno inseriti direttamente nella struttura dati thread safe dell'Environment concorrente.

Interfaccia Listener (modificata per concorrenza)

```
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()

Realizzazione di oggetti reattivi concorrenti

Un oggetto reattivo, cioè che implementa l'interfaccia **Listener** sarà analogo a prima, tranne per la funzione **fired()**.

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 GiocatoreFire
  }
}
```

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 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

Realizzazione di oggetti reattivi concorrenti

La funzione **fired()** deve ora lavorare con il diagramma delle classi condiviso da tutti i thread. Quindi deve diventare analogo ad un **Task** dei diagrammi delle attività.

Per fare ciò la funzione **fired()** invocherà semplicemente sull'**Executor** l'esecuzione di un **funtore** che conterrà tutta la logica delle transizioni (ciò che prima stava nel corpo del **fired()** nel caso non concorrente).

Il nuovo funtore NON sarà pubblico e sarà accessibile solo dall'oggetto reattivo corrispondente.

Per fare ciò faremo usa della visibilità di **Package**.

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!*

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~~
- **Eventi restituiti**:
  - 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.getDestinata
    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.

Environment nel caso concorrente

- Nel nostro caso l'idea generale del pattern **observable-observer** dove l'observable è un **Environment** e gli observer sono **Listener** (di tipo concorrente).

- L'Environment questa volta è diviso in due classi che trattano aspetti separati:
  - Una classe detta **Environment** che conterrà una struttura dati thread safe formata da una **coda bloccante** per ciascun Listener dove memeorizzare gli eventi che gli arrivano e sono in attesa di essere processati.

  - Una classe detta **EsecuzioneEnvironment** che si occuperà degli aspetti dinamici relativi all'environment: la sua inizializzazione, attivazione e spegnimento.

Environment (stuttura dati thread safe di supporto per scambio eventi)

```
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) {...}

        public static Set<Listener> getInsiemeListener() {...}

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

        public static Evento prossimoEvento(Listener lr)  {...}

}
```

Nota!

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

Environment (stuttura dati thread safe di supporto per scambio eventi)

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 (stuttura dati thread safe c

Ha la stessa logica di aggiungiEvento nel caso senza concorrenza…

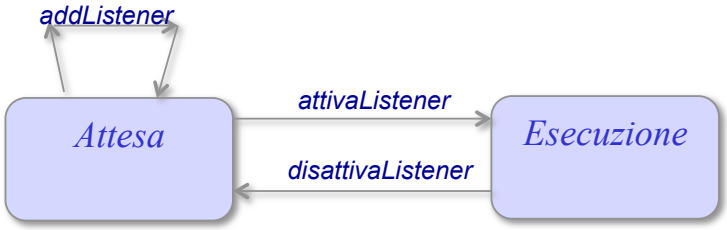
```
...
        public static void aggiungiEvento(Evento e) {
                // unico meccanismo per aggiungere eventi
                if (e == null) return;
                Listener destinatario = e.getDestinatario();
                if (destinatario != null && codeEventiDeiListener.containsKey(destinatario)) {
                        // 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

EsecuzioneEnvironment

- 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).



EsecuzioneEnvironment

```
public final class EsecuzioneEnvironment { //NB con final non si possono definire sottoclassi
        private EsecuzioneEnvironment() {
        }

        private static ConcurrentHashMap<Listener, Thread> listenerAttivi = null;

        public static enum Stato {
                Attesa, Esecuzione
        };

        private static Stato statocorrente = Stato.Attesa;

        public static synchronized void addListener(Listener lr) {
                if (statocorrente == Stato.Attesa) {...}
        }

        public static synchronized void attivaListener() {
                if (statocorrente == Stato.Attesa) { statocorrente = Stato.Esecuzione; ...}
        }
        public static synchronized void disattivaListener() {
                if (statocorrente == Stato.Esecuzione) { statocorrente = Stato.Attesa; ...}
        }

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

I Listener sono eseguiti in thread separati

EsecuzioneEnvironment

- La classe **EsecuzioneEnvironment** è svolge il suo lavoro attivando un thread per ogni oggetto Listener.

- Per fare ciò mantiene una struttura dati:

```
private static ConcurrentHashMap<Listener, Thread> listenerAttivi
```

dove ad ogni **Listener** viene associato l'oggetto Java **Thread** corrispondente al proprio thread.

EsecuzioneEnvironment

La struttura dati

```
private static ConcurrentHashMap<Listener, Thread> listenerAttivi
```

è manipolata dalle funzioni:

- **addListener()** che aggiunge una coppia **Listener**, **Thread** ad esso associato.

- **attivaLintener()** che manda in esecuzione ciascun **Listener**, invocando **start()** sul **Thread** ad esso associato.

- **disattivaLintener()** che:
  - **avvelena** ciascun thread mettendo in coda agli eventi di ciascun **Listener** un evento speciale **stop**;
  - Aspetta la fine del thread di ciascun **Listener**, con **join()**.

EsecuzioneEnvironment

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.

EsecuzioneEnvironment

```
        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.

EsecuzioneListener è l'eseguibile associato al Thread (vedi dopo).

EsecuzioneEnvironment

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;
        }
}
```


EsecuzioneListener

- Implementa il pattern **Funtore** con:
  - **Thread** come **esecutore**
  - **EsecuzioneListener** come **eseguibile**

(NB: esattamente come le attività complesse del diagramma delle attività)

- Pende il **prossimo evento** dalla coda associata al suo **Listener** e lancia il **fired()** del suo **Listener**.

- Tratta in modo speciale l'evento **Stop** (predefinito nel framework) che serve a fare terminare il thread.
  - **Non lo passa al fired()** dal suo Listener che deve rimanere associato al solo diagramma stati transizioni,
  - Ma lo **processa direttamente**.

EsecuzioneListener

```
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;
        }
}
```

Implementa il pattern funtore con:

- *Thread come esecutore*
- *EsecuzioneListener come eseguibile*

(NB: esattamente come le attività complesse del diagramma delle attività)

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

(NB: è processato qui, e non in fired() che deve rimanere associato al solo diagramma stati transizioni.)

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.*