Progettazione del Software

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

Diagramma degli stati e delle transizioni

Il diagramma degli stati e delle transizioni

Il diagramma degli stati e delle transizioni viene definito per **una classe**, ed intende descrivere **l'evoluzione di un generico oggetto** di quella classe.

Il diagramma rappresenta le sequenze di stati, le risposte e le azioni, che un oggetto attraversa durante la sua vita in risposta agli stimoli ricevuti.

Uno **stato** rappresenta una situazione in cui un oggetto ha un insieme di proprietà considerate stabili

Una **transizione** modella un cambiamento di stato ed è denotata da:

Evento [Condizione] / Azione

Il diagramma degli stati e delle transizioni

Il significato di una transizione del tipo di quella qui mostrata è:

- se l'oggetto
  - si trova nello **stato** S_1 e
  - riceve l'***evento*** *E* e
  - la ***condizione*** *C* è verificata
- allora
  - attiva l'esecuzione dell'***azione*** *A* e
  - passa nello **stato** S_2.

Stato

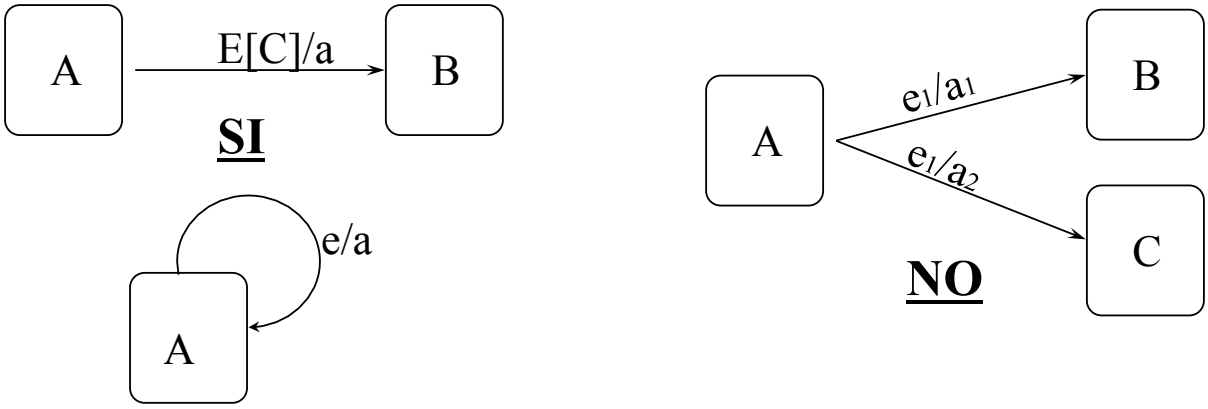
- Lo stato di un oggetto racchiude le proprietà (di solito statiche) dell'oggetto, più i valori correnti (di solito dinamici) di tali proprietà
- Una freccia non etichettata che parte dal "vuoto" ed entra in uno stato indica che lo stato è **iniziale**
- Una freccia non etichettata che esce da uno stato e finisce nel "vuoto" indica che lo stato è **finale**
- Stato iniziale e finale possono anche essere denotati da appositi simboli

stato iniziale

stato finale

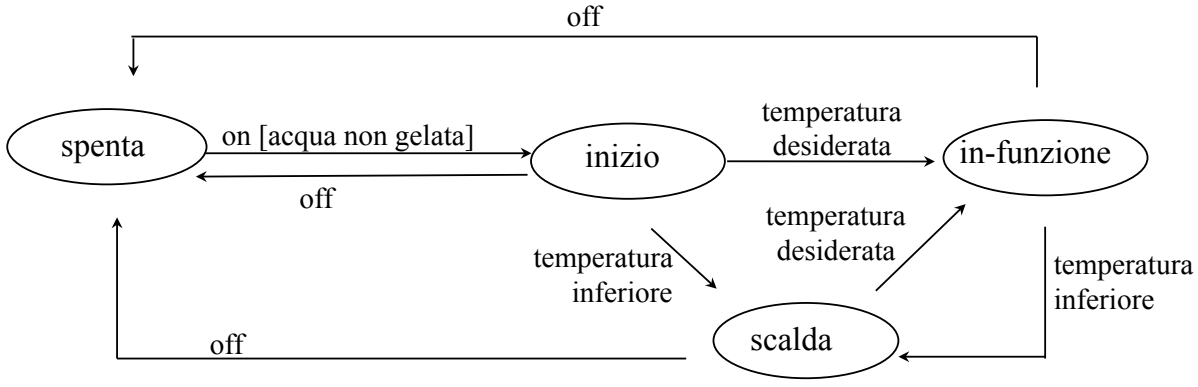
Transizione

- Ogni transizione connette due stati
- Il diagramma corrisponde ad un automa **deterministico** (transizioni dall stesso stato hanno eventi diversi), in cui un evento è un input, mentre un'azione è un output
- La condizione è detta anche "guardia" (guard)
- L'evento è (quasi) sempre presente (condizione e azione sono opzionali)



Esempio di diagramma degli stati e delle transizioni per la classe Caldaia

Descriviamo il diagramma degli stati e delle transizioni relativa ad una **classe** "Caldaia". In questo diagramma ogni transizione è caratterizzata solamente da eventi e condizioni (i cambiamenti di stato non hanno bisogno di azioni perché sono automatici)

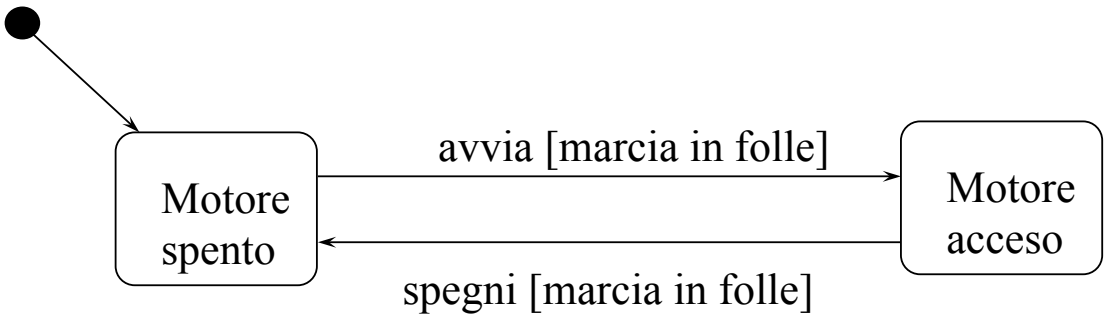


Esempio di diagramma degli stati e delle transizioni per la classe Motore

L'analisi dei requisiti ha evidenziato l'esistenza, nel diagramma delle classi, di una classe "Motore". Tracciare il diagramma degli stati e delle transizioni a partire da questi requisiti.

Un motore di automobile può essere spento o acceso, ma può essere avviato o spento solo se la marcia è in folle

Esempio di diagramma degli stati e delle transizioni per la classe Motore (soluzione)



Esercizio 1

Supponiamo che nel diagramma delle classi abbiamo rappresentato la classe “Menu a tendina”. Tracciare il diagramma degli stati e delle transizioni per tale classe, tenendo conto delle seguenti specifiche.

Un menu a tendina può essere visibile oppure no. Viene reso visibile a seguito della pressione del tasto destro del mouse, e viene reso invisibile quando tale tasto viene sollevato. Se si muove il cursore quando il menu è visibile, si evidenzia il corrispondente elemento del menu.

Esercizio 1: soluzione

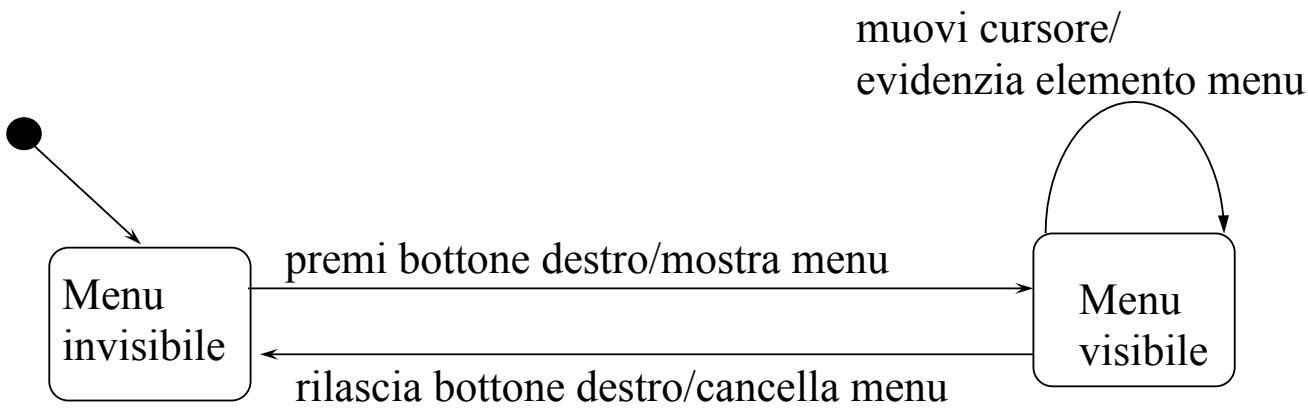
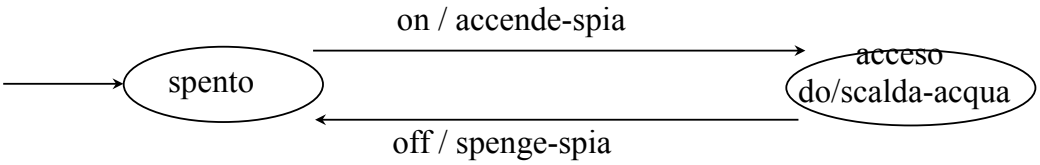


Diagramma degli stati e delle transizioni

Alcune volte vogliamo rappresentare dei processi che l'oggetto esegue senza cambiare stato. Questi processi si chiamano **attività**, e si mostrano negli stati con la notazione:

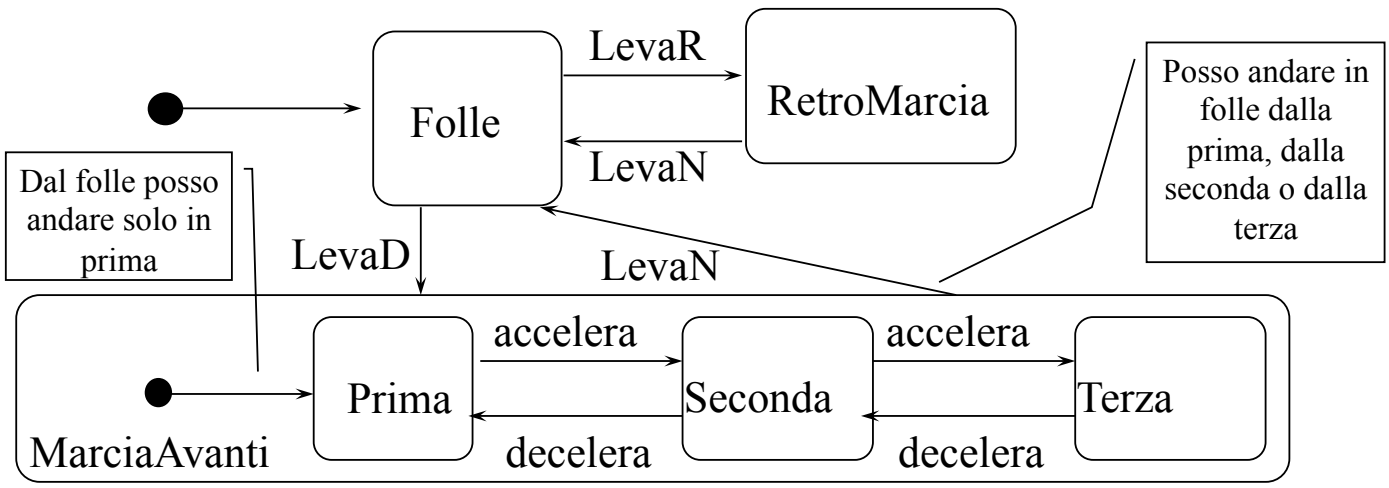
do / attività

Esempio (scaldabagno):



Stato composto

- Uno **stato composto** (o **macro-stato**) è uno stato che ha un nome, e che contiene a sua volta un diagramma
- Esiste uno stato iniziale del macro-stato
- I **sottostati** ereditano le transizioni in uscita del macro-stato



Aspetti metodologici nella costruzione del diagramma degli stati e delle transizioni

Un metodo comunemente usato per costruire il diagramma degli stati e delle transizioni prevede i seguenti passi

- *Individua gli stati di interesse*
- *Individua le transizioni*
- *Individua le attività*
- *Determina gli stati iniziali e finali*
- *Controllo di qualità*

Controllo di qualità del diagramma degli stati e delle transizioni

- Sono stati colti tutti gli aspetti insiti nei requisiti?
- Ci sono ridondanze nel diagramma?
- Ogni stato può essere caratterizzato da proprietà dell'oggetto?
- Ogni azione e ogni attività può corrispondere ad una operazione della classe?
- Ogni evento e ogni condizione può corrispondere ad un evento o condizione verificabile per l'oggetto?

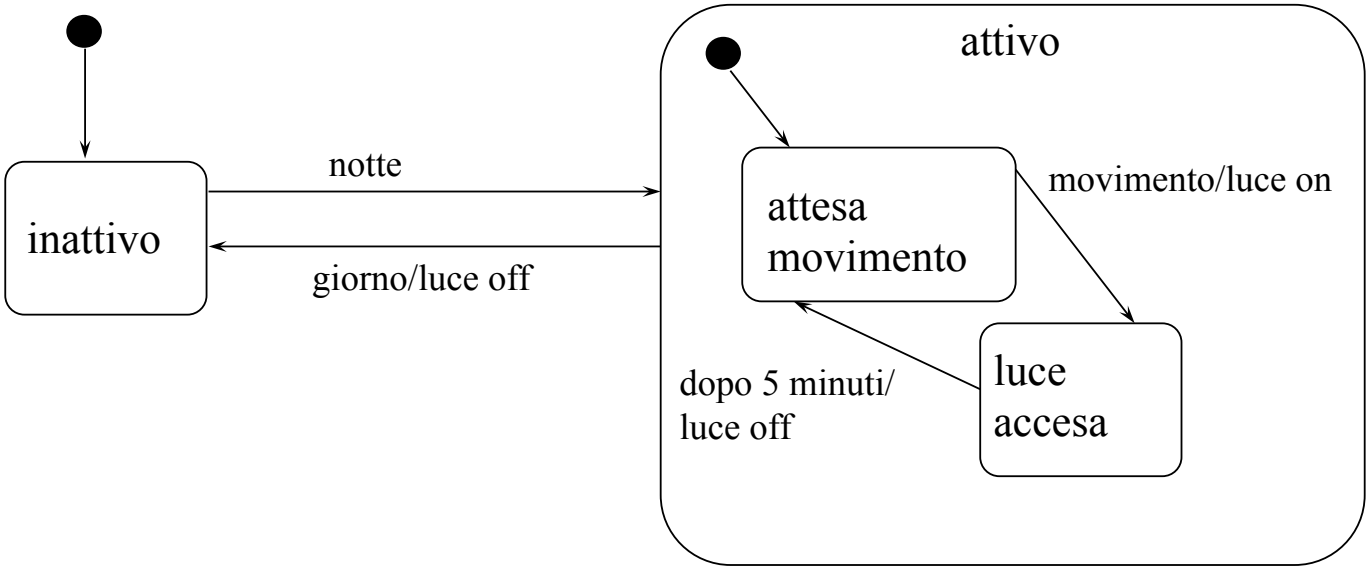
Esercizi su diagramma degli stati e delle transizioni

Esercizio 2

Supponiamo che nel diagramma delle classi abbiamo rappresentato la classe "InterruttoreAutomatico". Tracciare il diagramma degli stati e delle transizioni per tale classe, tenendo conto delle seguenti specifiche.

Un interruttore automatico collegato ad una cellula fotoelettrica e ad un sensore di movimento comanda una luce di un sottoscala che deve essere accesa solo di notte ed in presenza di movimento. Un'assenza di movimenti per cinque minuti consecutivi causerà lo spengimento della luce.

Esercizio 2: soluzione

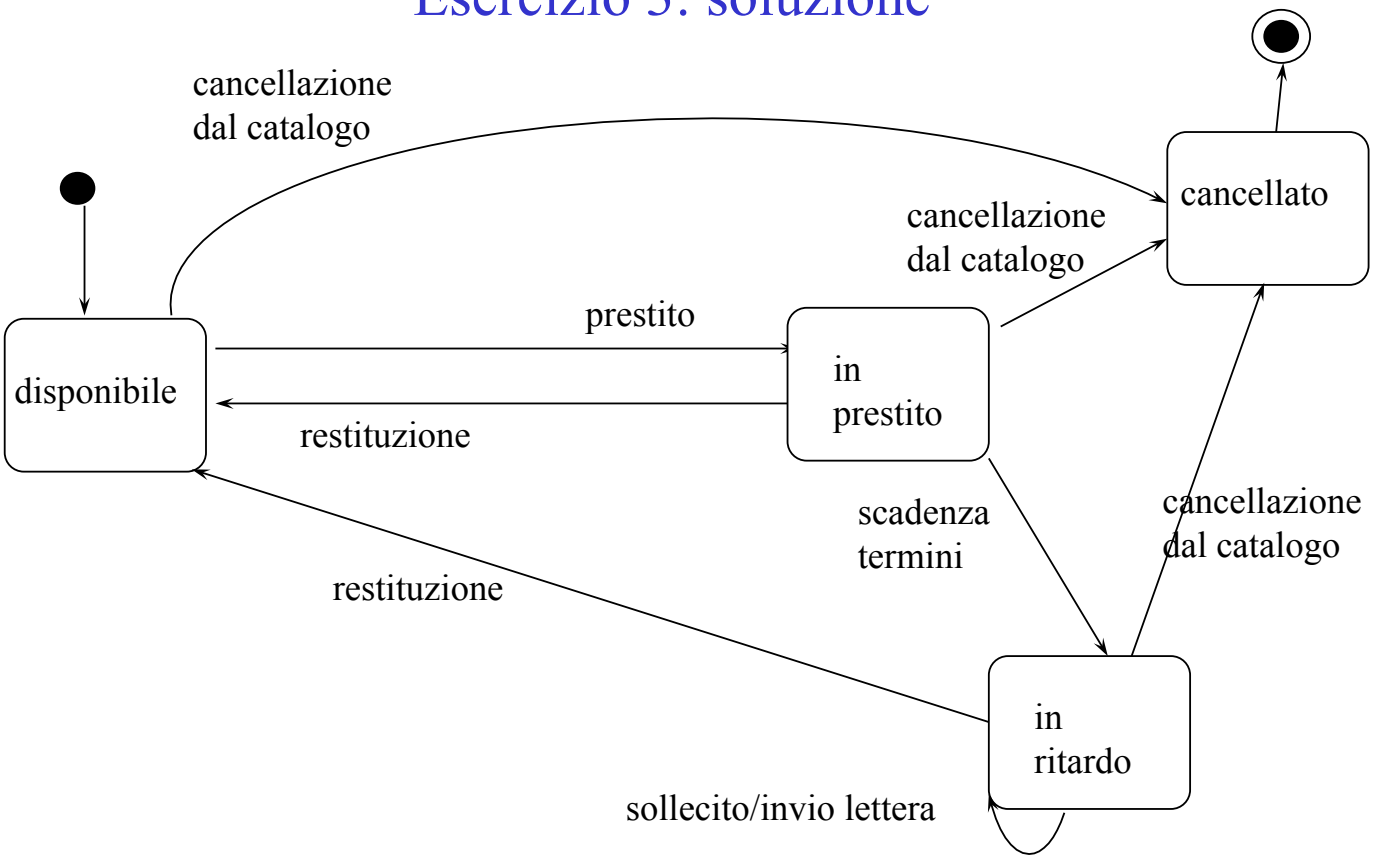


Esercizio 3

Supponiamo che nel diagramma delle classi abbiamo rappresentato la classe "Libro". Tracciare il diagramma degli stati e delle transizioni per tale classe, tenendo conto delle seguenti specifiche.

Una biblioteca può acquisire libri, che possono essere dati in prestito e successivamente restituiti. Quando scadono i termini del prestito, la restituzione è in ritardo, ed in tal caso la biblioteca può inviare (anche più volte) una lettera di sollecito. In ogni momento, un libro può essere cancellato dal catalogo.

Esercizio 3: soluzione

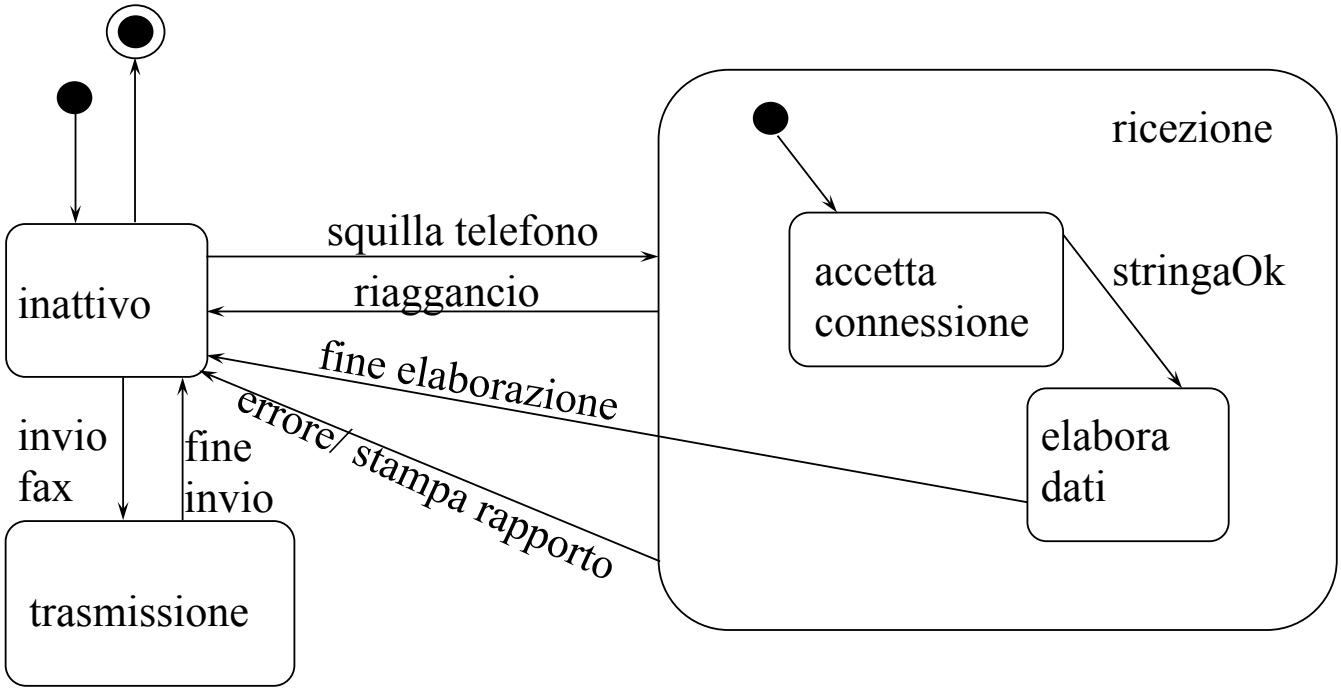


Esercizio 3

Supponiamo che nel diagramma delle classi abbiamo rappresentato la classe "Fax". Tracciare il diagramma degli stati e delle transizioni per tale classe, tenendo conto delle seguenti specifiche.

Un fax può essere inattivo, ricevente o trasmittente. Se il fax è inattivo, con il comando di invio fax si porta il dispositivo nello stato trasmittente, e con il comando di fine invio si riporta nello stato inattivo. Quando il fax è inattivo e si verifica una chiamata (segnalata da uno squillo del telefono), va in stato ricevente, e quindi accetta la connessione. Se la stringa iniziale è corretta, il fax elabora i dati, e infine ritorna inattivo. In ogni momento della trasmissione, il chiamante può riagganciare, facendo ritornare il fax nello stato inattivo. In ogni momento della trasmissione, se si verifica un errore in ricezione, il fax ritorna inattivo e stampa un rapporto di errore.

Esercizio 3: soluzione



Esercizio 4

Supponiamo che nel diagramma delle classi abbiamo rappresentato la classe "DispositivoPortatile". Tracciare il diagramma degli stati e delle transizioni per tale classe, tenendo conto delle seguenti specifiche.

Un dispositivo portatile per la comunicazione di emergenze può essere acceso o spento con lo stesso tasto "OnOff". Gli altri due tasti del dispositivo sono: "Emergenza" e "Invio". Per comunicare un'emergenza bisogna, nell'ordine, premere il tasto "Emergenza" e poi "Invio". Per disattivare la tastiera del dispositivo bisogna premere il tasto "Invio". Per riattivare la tastiera quando è stata precedentemente disattivata bisogna premere il tasto "Invio". In ogni momento si può spengere il dispositivo. In ogni circostanza, la pressione di un tasto non contemplato nella descrizione precedente non produce alcun effetto.

Esercizio 4: commento (1)

Per comodità, numeriamo i requisiti.

1. Un dispositivo portatile per la comunicazione di emergenze può essere acceso o spento con lo stesso tasto "OnOff".
2. Gli altri due tasti del dispositivo sono: "Emergenza" e "Invio".
3. Per comunicare un'emergenza bisogna, nell'ordine, premere il tasto "Emergenza" e poi "Invio".
4. Per disattivare la tastiera del dispositivo bisogna premere il tasto "Invio".
5. Per riattivare la tastiera quando è stata precedentemente disattivata, bisogna premere il tasto "Invio".
6. In ogni momento si può spengere il dispositivo.
7. In ogni circostanza, la pressione di un tasto non contemplato nella descrizione precedente non produce alcun effetto.

Esercizio 4: commento (2)

- I requisiti 1 e 2 affermano che esistono tre simboli nell'alfabeto di input ("OnOff", "Emergenza" e "Invio").
- Il requisito 1 implica l'esistenza di (almeno) due stati: "acceso" e "spento".
- Il requisito 6 suggerisce che è conveniente modellare lo stato "acceso" come macro-stato.

Esercizio 4: commento (3)

- I requisiti 3 e 4 implicano l'esistenza di altri tre stati, tutti interni al macro-stato "acceso":
  - "tastiera attivata": lo stato iniziale del macro-stato
  - "tastiera disattivata": lo stato a cui si giunge con la pressione del tasto "Invio"
  - "pronto": lo stato a cui si giunge dopo la pressione del tasto "Emergenza"
- Le transizioni fra stati sono dettate dai requisiti 3, 4, 5 e 7.
- Le transizioni sono tutte prive di condizioni.
- Si ha l'azione di "comunica emergenza" in corrispondenza della transizione dallo stato "pronto" a quello "tastiera attivata"

Esercizio 4: soluzione

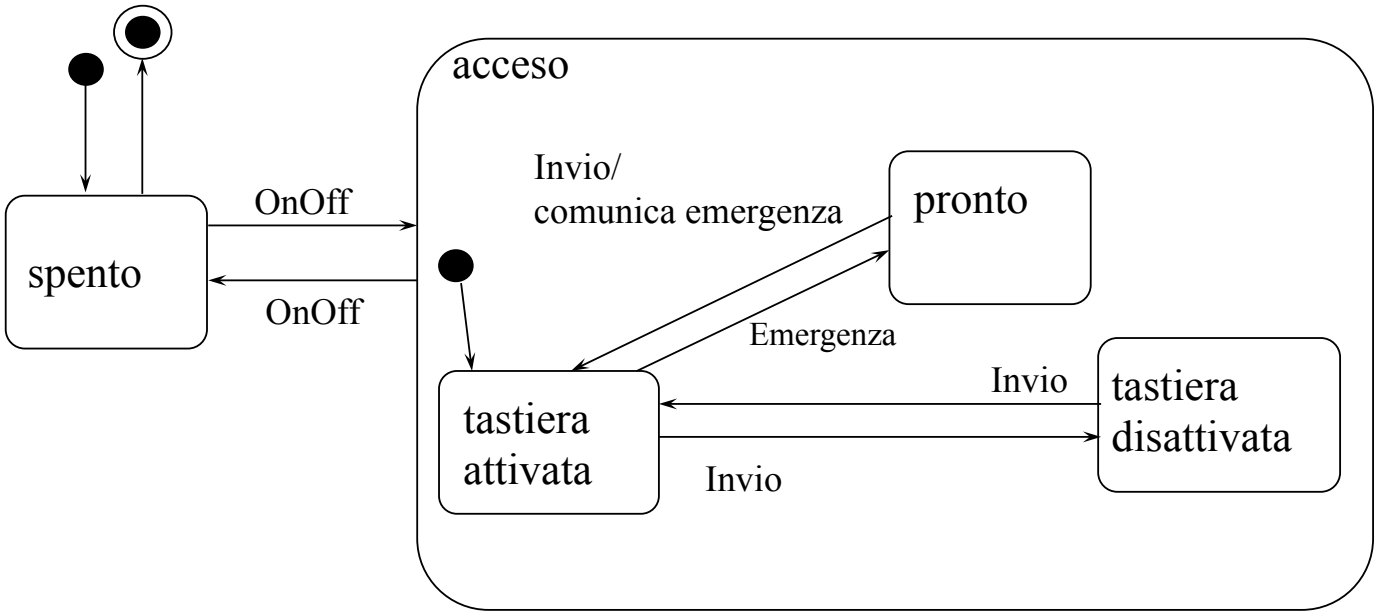


Diagramma degli stati e delle transizioni di oggetti reattivi

Principi generali

- Assumiamo di avere diversi **oggetti reattivi**, cioè con associato un diagramma stati-transizioni.
- Assumiamo che l'interazione sia basata su scambio esplicito di **eventi**
- Assumiamo che gli eventi abbiano un **mittente** ed un **destinatario**
  In particolare ammettiamo
  - **Messaggi punto-punto**: un oggetto manda un messaggio ad un altro oggetto
  - **Messaggi in broadcasting**: un oggetto manda un messaggio a tutti gli altri oggetti (vedremo esempi successivamente).
- Inoltre gli eventi possono avere **parametri** con specifico contenuto informativo (il cosiddetto *payload* del messaggio)
- Una **azione** può a sua volta lanciare un **evento** (uno solo per semplicità) per un'altro oggetto o in broadcasting.

Osservazioni

- Nel diagramma delle classi per semplicità identifichiamo **l'azione** stessa con **l'evento lanciato**.

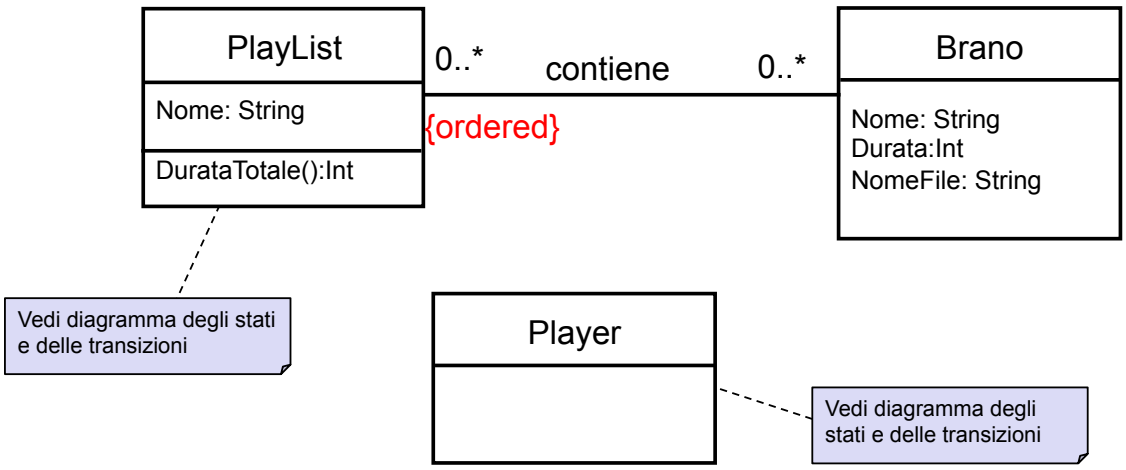
- Diamo una specifica dettagliata di ciò che avviene ad ogni transizione:
  - Quali **eventi** sono **recepiti** e **lanciati** (e a chi)
  - Come cambiano eventuali **variabili di stato ausiliarie** associate allo stato dell'oggetto (vedi sotto)
  - Come **cambia l'istanziazione del diagramma delle classi**

- Le **variabili di stato ausiliarie** non sono di interesse per il cliente servono solo alla corretta realizzazione delle azioni associate alle varie transizioni. Quindi non vanno confuse con con gli attributi dell'oggetto stesso.

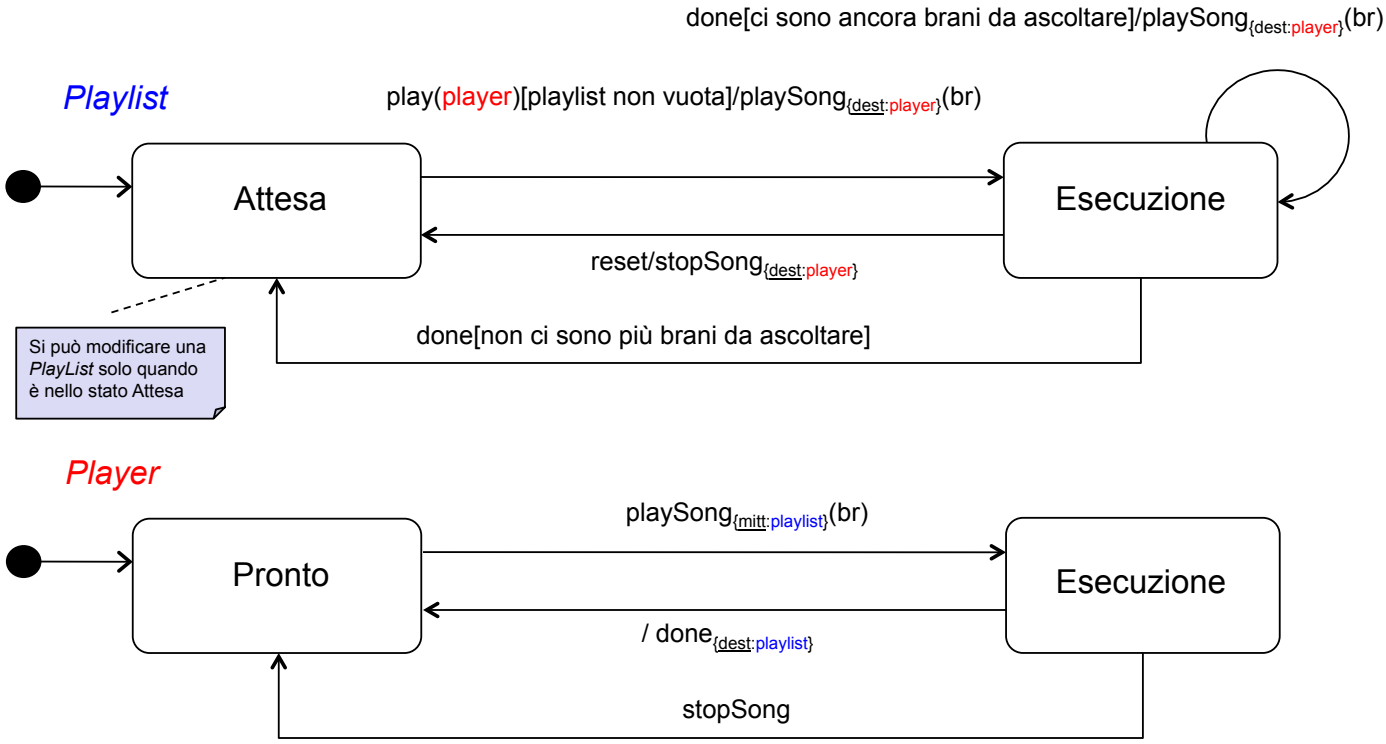
- Il **diagramma degli stati e transizioni** è sempre corredato da detta **specifica** che ne chiarisce in dettaglio la semantica.

Diagramma delle classi

- Consideriamo il seguente diagramma delle classi:
  - PlayList e Brano li abbiamo già incontrati in precedenza
  - Player è una classe che non contiene alcun dato (ma a cui è associato un diagramma stati e transizioni



Diagrammi degli stati e delle transizioni



Specifica degli stati di PlayList

InizioSpecificaStatiClasse PlayList

Stato: {Attesa, Esecuzione}
Variabili di stato ausiliarie:
player: Player
prossimobrano: intero

Stato iniziale:
stato = Attesa
player = --
prossimobrano = --

"--" sta per non definito

FineSpecifica

Specifica delle transizioni di PlayList

InizioSpecificaTransizioniClasse PlayList

Transizione: Attesa→ Esecuzione
play(player)[playlist non vuota]/playSong$_{\{dest:player\}}$(br)

Evento: play(player:Player)

Condizione: this.contiene non vuoto

Azione:
pre: *evento*.dest = this
post: *nuovoevento* = playSong{mitt = this, dest = player}(br: Brano)
and
this.player = player and
this.prossimobrano = 0 and
<this,br> in contine and
posizione(this,br) = this.prossimobrano

"evento" denota l'evento ricevuto, "mitt" e "dest", denotano il mittente e il destinatario dell'evento

"nuovoevento" denota l'evento da mandare con l'azione

…
FineSpecifica

Specifica delle transizioni di PlayList

InizioSpecificaTransizioniClasse PlayList

…

Transizione: Esecuzione→ Esecuzione
done[ci sono ancora brani da ascoltare]/playSong$_{\{dest:player\}}$(br)

Evento: done

Condizione: this.prossimobrano < |{b | <this,b> in contiene}|

Azione:
pre: evento.dest = this and evento.mitt = this.player
post: nuovoevento =
playSong{mitt = this, dest = this.player}(br: Brano) and
this.player = Pre(this.player) and
this.prossimobrano = Pre(this.prossimobrano)+1 and
<this,br> in contine and
indexOf(contienr(this,br)) = this.prossimobrano

…
FineSpecifica

Specifica delle transizioni di PlayList

InizioSpecificaTransizioniClasse PlayList

…

Transizione: Esecuzione→ Attesa
done[non ci sono brani da ascoltare]

Evento: done

Condizione: this.prossimobrano >= |{b | <this,b> in contiene}|

Azione:
pre: evento.dest = this and evento.mitt=this.player
post: this.prossimobrano = -- and
this.player = --

…
FineSpecifica

Specifica delle transizioni di PlayList

InizioSpecificaTransizioniClasse PlayList

…

Transizione: Esecuzione→ Attesa
reset/stopSong{dest:player}

Evento: reset

Condizione: --

Azione:
pre: evento.dest = this
post: nuovoevento = stopSong{mitt = this, dest = Pre(this.player)}
this.prossimobrano = -- and
this.player = --

FineSpecifica

Specifica degli stati di Player

InizioSpecificaStatiClasse Player

Stato: {Pronto, Esecuzione}
Variabili di stato ausiliarie:
playlist: PlayList
brano: Brano

Stato iniziale:
stato = Pronto
playlist = --
brano = --

FineSpecifica

Specifica delle transizioni di Player

InizioSpecificaTransizioniClasse Player

Transizione: Pronto→ Esecuzione
playSong$_{\{mitt:playlist\}}$(br)

Evento: playSong(br:Brano)

Condizione: --

Azione:
pre: evento.dest = this
post: this.playlist= evento.mitt and
this.brano = br

…
FineSpecifica

Specifica delle transizioni di Player

InizioSpecificaTransizioniClasse Player

…

Transizione: Esecuzione→ Pronto
/ done{dest:playlist}

Evento: -- /*generato da se stesso*/

Condizione: --

Azione:
pre: --
post: nuovoevento = done{mitt = this, dest = Pre(this.player)}
this.playlist= -- and
this.brano = --

…
FineSpecifica

Specifica delle transizioni di Player

InizioSpecificaTransizioniClasse Player

…

Transizione: Esecuzione→ Pronto
stopSong

Evento: stopSong

Condizione: --

Azione:
pre: stopSong.dest = this and stopSong.mitt = Pre(this.player)}
post: this.playlist= -- and
this.brano = --
FineSpecifica

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

Progetto e realizzazione

- 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.

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

- *Ma prima di prendere queste decisioni dobbiamo stabilire lo schema generale di gestione degli eventi nella nostra applicazione.*

Decisioni preliminari sulla gestione degli eventi

- Per realizzare gli oggetti "reattivi", cioè con associato un diagramma degli stati e delle transizioni, sono possibili varie scelte. In particolare:
  - Gli **eventi sono chiamate a funzioni** che modificano lo stato dell'oggetto reattivo:
    - È uno schema realizzativo idoneo soprattutto quando l'interazione con gli oggetti è pilotata da un cliente esterno al sistema.
    - È stato utilizzato in vecchie edizioni di questo corso (Vecchio Prog. SW 1)
  - Gli **eventi sono messaggi** che oggetti reattivi si scambiano:
    - È uno schema realizzativo che permette una forte interazione tra i vari oggetti del sistema.
    - Porta a realizzare parti del programma **orientate agli eventi** (**event-based programming** – molto usato per interfacce grafiche, video-giochi, realizzazione di dispositivi reattivi)

Noi focalizziamo su questo!

Eventi come messaggi

- In questa edizione del corso noi ci **focalizzeremo su eventi come messaggi**, mettendo in piedi un "framework" opportuno per la gestione degli eventi.

- Tale gestione degli eventi verrà **inizialmente** proposta considerando un **unico flusso di controllo** (programmazione basata su eventi, realizzata con programmi sequenziali).

- Successivamente renderemo i **flussi di controllo** dei singoli oggetti indipendenti e **concorrenti** (programmazione basata su eventi, realizzato con programmi multi-thread).

Eventi come messaggi

- Assumeremo che ogni **evento** o messaggio abbia un **mittente** ed un **destinatario** esplicito, realizzando *connessioni point-to-point*.

- Inoltre permetteremo di mandare **messaggi in broadcasting** a tutti gli oggetti reattivi del sistema, cioè *connessioni broadcasting*

- Per semplicità **non affronteremo** il caso in cui i messaggi abbiano **più di un destinatario**, ma non siano in broadcasting, cioè *connessioni multicasting*. Va comunque osservato che quanto proposto può facilmente essere esteso a questo caso.

- Ammetteremo che un mittente possa non dichiararsi quando manda un messaggio (perché conoscere il mittente è irrilevante, o per altri motivi)

Realizzazione degli eventi

- Gli eventi sono rappresentati da oggetti di una classe Java **Evento**.
- 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 Eventi**.

- Inoltre gli oggetti la cui classe più specifica è proprio Evento vengono usati 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** (i cui metodi pubblici non fanno side-effect):
- 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 immutabili 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 MioEsempio(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)) {
         EventoEsempio e = (EventoEsempio) o;
         return info == e.info;
      }
      else return false;
   }
  public int hashCode() { return super.hashCode() + info.hashCode(); }
   public String toString() {
      return "EventoEsempio(" + getMittente() + " -> " + getDestinatario() + " : " + info + ")";
   }
}
```

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

- 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).

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
- Il corpo della funzione **fired()** 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.

Gestione delle transizioni: codice

```
public class OggettoConStato implements Listener {
   ...
   //Gestine delle transizioni
   public Evento fired(Evento e) {
     if (e.getCalled() != this && e.getCalled() != 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: 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 gusto 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

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()

Environment

Map che associa ad ogni Listener registrato una coda specifica

```
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);
   }
  }
 }
 ...
```

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 sono eventi da processare nelle 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.*