SAPIENZA Università di Roma, Facoltà di Ingegneria

Corso di
PROGETTAZIONE DEL SOFTWARE
Laurea in Ingegneria Informatica
Prof. Giuseppe De Giacomo & Prof. Massimo Mecella
A.A. 2008/09

Progettazione e realizzazione di diagrammi delle attività in JAVA

Classi come funzioni

Nei linguaggi orientati agli oggetti, ed in particolare in JAVA è spesso necessario realizzare classi che rappresentano funzioni.

Chiamiamo nel seguito **funtore** una classe che rappresenta una funzione, cioè le cui istanze rappresentano attivazioni (computazioni) di una funzione.

Il loro utilizzo in JAVA è comune. Per esempio gli oggetti che implementano l'interfaccia `Runnable` sono oggetti *funtore* che rappresentano (oltre eventualmente ad altro) la computazione descritta in `run()`. Essi vanno eseguiti costruendo un oggetto `Thread` che funge appunto da *esecutore* e chiamando su tale oggetto il metodo `start()` che a sua volta chiama il metodo `run()` dell'oggetto `Runnable` associato al `Thread`.

Esempio Runnable

```
package esempioRunnerConcorrenti;

public class Runner implements Runnable {

   private String nome;

   public Runner(String n) {
      nome = n;
   }

   public void run() {
      for (int i = 0; i < 25; i++) {
         System.out.println(nome + " sta girando");
      }
   }
}

package esempioRunnerConcorrenti;

public class Main {
   public static void main(String[] args) {
      // Crea attivazioni del Runner
      Runner r1 = new Runner("Alpha");
      Runner r2 = new Runner("Beta");
```



```
      // Attivazioni passata all'esecutore (cioe' Thread)
      Thread alpha = new Thread(r1);
      Thread beta = new Thread(r2);
      // Esecuzione dell'attivazione (start() chiama run()
      alpha.start();
      beta.start();
   }
}
```

Il pattern Funtore

Approfondiamo la nozione di funtore, cioè una classe le cui istanze rappresentano attivazioni di funzioni. Lo facciamo trasformando una funzione statica in un funtore.

Consideriamo un esempio di funzione:

```
package funtore;

public class Funzione {
   public static double function(Object o, int i) {
      // fa cose con o, i
      System.out.println("faccio cose con " + o + " e " + i);
      // restituendo ris
      return i;
   }
}
```


Il pattern Funtore

La corrispondente classe funtore avrà la seguente forma:

```
package funtore;

public class Funtore implements Runnable {

   private boolean eseguita = false;

   private Object o;
   private int i;
   private double ris;

   public Funtore(Object oo, int ii) {
      o = oo;
      i = ii;
   }

   public synchronized void run() {
      if (eseguita)
         return;
      eseguita = true;
      // fa cose con o, i
      System.out.println("faccio cose con " + o + " e " + i);
```



```
      // restituendo ris
      ris = i;
   }

   public synchronized double getRis() {
      if (!eseguita)
         throw new RuntimeException();
      return ris;
   }

   public synchronized boolean estEseguita() {
      return eseguita;
   }

}
```

Il pattern Funtore

Discutiamone i vari aspetti.

- Il nome della funzione è ora diventato il nome della classe.
- I parametri della funzione sono ora passati in ingresso al costruttore della classe e vengono memorizzati in opportuni campi dati privati della classe stessa.
- Il risultato viene anche esso memorizzato in un campo privato della classe e reso disponibile attraverso `getRis()`.
- Il corpo della funzione è adesso il corpo del metodo `run`. In questo caso genererà effettivamente la computazione voluta.

Il pattern Funtore

Vogliamo che ogni oggetto della classe `Funtore` corrisponda esattamente ad una attivazione della funzione `funzione()`. Per fare ciò procediamo come segue:

- Introduciamo una variabile booleana `eseguita` che indica se l'attivazione corrente del funtore è stata eseguita o meno e che inizialmente è posta a `false`.

- Il metodo responsabile dell'esecuzione (`run()` in questo caso) verifica se la variabile `eseguita` è `true`,

  – se non lo è esce senza fare altro, poiché la computazione è già stata eseguita (o potremmo dire che l'attivazione corrente della funzione è stata "consumata") e non deve essere eseguita una seconda volta. *Si*

Il pattern Funtore

noti però che ovviamente è possibile generare una nuova attivazione del funtore costruendo un nuovo oggetto.

  – altrimenti pone la variabile `eseguita` a `true` ed esegue il metodo.

- Il metodo `getRis()`, che restituisce il risultato al cliente, può essere chiamato solo se `eseguita` è `true`, cioè se il metodo responsabile dell'esecuzione (`run()`) è stato effettivamente eseguito.

- Infine diamo al cliente la possibilità di leggere la variabile `eseguita`, attraverso il metodo `estEseguita()`, per evitare un uso scorretto del funtore.

Il pattern Funtore

C'è un ultimo aspetto da commentare: se assumiamo, come in questo caso, che il funtore venga usato in un programma multithread, allora dobbiamo renderlo "thread safe", cioè dobbiamo evitare che esecuzioni multiple dei metodi di un dato oggetto corrompano la correttezza delle informazione riportate nei suoi campi dati. Per fare ciò rendiamo tutti i metodi pubblici del funtore `synchronized`.

In questo modo, per ogni oggetto della classe `Funtore`, sarà in ogni momento in esecuzione un solo metodo tra quelli pubblici messi a disposizione dall'oggetto stesso (cioè tra `run()`, `getRis()`, `estEseguita()`), rispecchiando pienamente il comportamento di una attivazione di funzione.

Il pattern Funtore

Concludiamo questa introduzione al pattern funtore mostrando come un cliente deve invocare una funzione rappresentata da un funtore.

In particolare confrontiamo la chiamata ad una funzione con la chiamata al corrispondente funtore, facendo uso del nostro esempio:

Il pattern Funtore

```
package funtore;

public class MainFunzione {
   public static void main(String[] args) {
      // usando FUNZIONI
      Object o = new Object();
      double ris = Funzione.function(o, 10);
      System.out.println(ris);
   }
}
```


Il pattern Funtore

```
package funtore;

public class MainFuntore {
   public static void main(String[] args) {
      // usando FUNTORI
      Object o = new Object();
      Funtore f = new Funtore(o, 10);
      Thread t = new Thread(f);
      t.start();
      try {
         t.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println(f.getRis());
   }
}
```


Il pattern Funtore

Discutiamo i vari aspetti della chiamata ad un funtore

- Prima di tutto deve essere costruito un oggetto `Funtore` corrispondente alla attivazione della funzione `Funzione`.
- Questo oggetto va passato all'*esecutore* (`Thread` in nostro caso).
- L'esecutore richiama il metodo del funtore che effettivamente computa la funzione (`run()` chiamato da `start()`, nel nostro caso).
- Infine una volta terminata l'esecuzione di questo metodo, il risultato è reso disponibile attraverso `getRis()`.

  L'uso del join del thread generato con il thread del `main()` *in questo esempio è solo contingente, l'importante è dare la possibilità al metodo*

`run()` *di iniziare (e quindi completare visto che tutti i metodi del funtore sono* `synchronized`*) prima di richiedere il risultato.*

Si noti che complessivamente il cliente risulta più complesso che nel caso della chiamata ad una funzione, ma anche più flessibile. Per esempio, è possibile far restituire più di in risultato ad un funtore, cosa impossibile nel caso di funzioni.

Realizzazione dei diagrammi delle attività UML

Un diagramma delle attività descrive una **attività complessa**, che potremmo in prima approssimazione considerare come una funzione, o meglio una operazione o un processo, in cui il flusso di controllo stesso che costituisce il corpo della funzione è di primario interesse.

Si noti usiamo il termine funzione per semplicità, in realtà tipicamente delle funzioni interesso solo il risultato prodotto, mentre per le attività complesse interessa il flusso delle sotto attività coinvolte. Infatti è assolutamente possibile avere attività complesse che non producono alcun risultato, o che non terminano affatto (per esempio, il sistema operativo di un calcolatore è una siffatta attività).

- Altre **sotto attività complesse**.

Tali attività sono organizzate secondo uno **flusso di controllo** anche ramificato in diversi flussi concorrenti.

Tipi di attività

In un diagramma delle attività associato ad una attività complessa (inclusa l'attività principale) troviamo varie **sotto attività**. I tipi di sotto attività sono fondamentalmente i seguenti:

- **Attività atomiche** atte ad interagire con la realizzazione del diagramma delle classi UML, chiamiamo tali attività **task**;
- **Operazioni di ingresso/uscita**, cioè **attività** atte ad effettuare operazioni di ingresso uscita, o che comunque si rivolgono all'esterno del sistema realizzato.
- **Operazioni per la gestione degli eventi**, cioè attività atte a gestire gli eventi degli oggetti dotati di diagramma di transizione;

Attività atomiche: task

I task sono le attività atomiche fondamentali. Esse servono ad accedere e a modificare le classi JAVA che corrispondono alle classi e alle associazioni del diagramma delle classi.

Tali classi saranno in generale accedute concorrentemente da diverse attività. Quindi dobbiamo garantirne l'integrità a fronte di accessi concorrenti.

Per regolare questo accesso concorrente, non possiamo semplicemente aggiungere `synchronized` a tutti i metodi delle classi realizzate. Infatti é immediato accorgersi che questo porterebbe a *deadlock*.

Facciamo un esempio. Consideriamo due classi A e B ciascuna delle quali ha link verso l'altra (per esempio abbiamo una associazione con responsabilità doppia che le lega). Supponiamo che una prima attività parta da un oggetto a istanza di A e navighi i link verso gli oggetti B a cui è connesso. Tra questi supponiamo esista un oggetto b. Intanto

un seconda attività accede a b e ne naviga i link verso oggetti istanza di A. Supponiamo che tra questi oggetti ci sia a. E' immediato verificare che se sincronizziamo tutti i metodi di A e di B si crea un deadlock: la prima attività accede ad a impedendo altri accessi ad a, intanto la seconda attività accede a b impedendo altri accessi a b. La prima attività ora naviga i link da A verso B invocando qualche metodo degli oggetti B legati ad a. Ma tra questi c'è b a cui l'accesso è bloccato, quindi l'attività viene bloccata in attesa di avere l'accesso a b. Intanto la seconda attività naviga i link verso A invocando qualche metodo degli oggetti di A legati b, ma tra questi cè a al quale l'accesso è bloccato. Di conseguenza la seconda attività si blocca in attesa di avere accesso ad a. Risultato: entrambe le attività sono bloccate in attesa rispettivamente dell'altra, cioè abbiamo un *deadlock*.

```
public final class Executor {
   private Executor(){}
   public synchronized static void perform(Task t) {
      t.esegui(new Executor());
   }
}
```

Attività atomiche: task

Risolviamo il problema dell'accesso concorrente facendo uso del pattern funtore e richiedendo che l'esecutore di questi funtori si prenda carico della sincronizzazione. Più precisamente:

- Richiediamo che le attività atomiche che accedono al diagramma delle classi (o meglio alla sua realizzazione in JAVA) implementano una interfaccia specifica `Task`:

```
package _framework;

public interface Task {
   public void esegui(Executor e);
}
```

- I task sono eseguiti da un esecutore specifico `Executor`:

```
package _framework;
```


Attività atomiche: task

Commentiamo il framework presentato.

- Le attività atomiche che accedono al diagramma delle classi implementano una interfaccia specifica `Task` che include un solo metodo `esegui()`.

- Tale metodo è pseudo privato (cfr. lo schema realizzativo per associazioni con responsabilità doppia) perchè richiede come parametro un oggetto `Executor`, ma `Executor` ha un costruttore privato. Quindi `esegui()` può essere attivato solo dai metodi di `Executor`, cioè da `perform()`.

- `Executor` è l'esecutore dei task e li esegue attraverso il suo unico metodo: il metodo statico `perform()`.

- Il metodo `perform()` è `synchronized` il che implica che se una attivazione di `perform()` è in esecuzione in un thread, tutte le altre eventuali attivazioni `perform()` su altri thread rimangono in attesa.

  In altre parole: *Un solo* `Task` *alla volta può essere in esecuzione e quindi modificare il diagramma delle classi.*

  In questo modo garantiamo di non poter corrompere il diagramma delle classi a causa si accessi concorrenti, cioè rendiamo il diagramma delle classi *thread safe*.

```
      eseguita = true;
      TipoLinkPossedere poss1 = new TipoLinkPossedere(personaInput_1,
            contoInput);
      ManagerPossedere.inserisci(poss1);
      TipoLinkPossedere poss2 = new TipoLinkPossedere(personaInput_2,
            contoInput);
      ManagerPossedere.inserisci(poss2);
   }

   public synchronized boolean estEseguita() {
      return eseguita;
   }

}
```

Esempio di attività atomica

```
package attivita_atomiche;

import _framework.*;
import conto.*;
import persona.*;
import possedere.*;

public class AperturaConto implements Task {

   private boolean eseguita = false;

   private Persona personaInput_1;
   private Persona personaInput_2;
   private Conto contoInput;

   public AperturaConto(Persona p1, Persona p2, Conto c) {
      personaInput_1 = p1;
      personaInput_2 = p2;
      contoInput = c;
   }

   public synchronized void esegui(Executor e) {
      if (e == null || eseguita == true)
         return;
```


Esempio di attività atomica

Commentiamo l'esempio:

- Segue il pattern funtore implementando l'interfaccia `Task`.
- Prende tre parametri di ingresso e non restituisce nulla, ma fa side-effect sul diagramma delle classi.
- Il corpo del funtore è il corpo di `esegui()` che appunto effettua l'accesso al diagramma delle classi
- Si noti il controllo che il parametro `Executor e` non sia `null` per non inficiare il meccanismo della chiamata pseudo-privata: l'unico metodo che può generare oggetti `Executor` è `Executor.perform()`.

- in questo modo i `Task` possono essere eseguiti solo da `Executor`, come richiesto.

Operazioni di ingresso/uscita

Per le operazioni di ingresso/uscita presenti in una attività complessa, non diamo alcun a strategia implementativa specifica. Semplicemente le realizzeremo come funzioni o funtori con opportuni esecutori.

- Ovviamente rimangono validi i principi base della modulatità: basso accoppiamento tra i moduli, alta coesione all'interno di ciascun modulo, alta information hiding, e alto interfacciamento esplicito tra moduli.

- Questo suggerisce di racchiudere in uno o più moduli separati le operazioni di ingresso/uscita rendendo così (la realizzazione in JAVA de) il diagramma delle attività indipendente dall'interfaccia utente scelta per l' ingresso/uscita.

- Per esempio si può raccogliere in un'unica classe un insieme di metodi statici che servono per le varie operazioni di ingresso/uscita, di modo che (la realizzazione in JAVA de) il diagramma delle attività faccia uso di chiamate a questi metodi per l'ingresso/uscita, e il loro corpo si occupi dell'interfacciamento con l'utente.

Operazioni di ingresso/uscita: esempio

```
package attivita_io;

import persona.*;
import conto.*;

public class AttivitaIO {
   // per ora su System.out, come esercizio lasciato agli studenti
   // e/o ad esercitazione, metterlo in ambiente grafico

   public static RecordPersonaPersonaConto LeggiDatiConto() {
      RecordPersonaPersonaConto result = new RecordPersonaPersonaConto();
      result.marito = new Persona("DUIBNIO17H501F", "Pippo", "De Pippis");
      result.moglie = new Persona("HIOHOIH60G230T", "Clarabella", "De Claris");
      String codice = leggiCodice();
      int saldo = leggiSaldoIniziale();
      result.conto = new Conto(codice, saldo);
      return result;
   }

   private static String leggiCodice() {
      // per ora simula l'interazione usando delle costanti;
      // come esercizio lasciato agli studenti
      // e/o ad esercitazione, metterlo in ambiente grafico
      return "00002005";
```

```
    }

    private static int leggiSaldoIniziale() {
        // per ora simula l'interazione usando delle costanti;
        // come esercizio lasciato agli studenti
        // e/o ad esercitazione, metterlo in ambiente grafico
        return 1000;
    }

    public static void AperturaOK() {
        System.out.println("Apertura del conto avvenuta con successo");
    }

    public static void StampaOK() {
        System.out.println("Il conto per fortuna non e' in rosso :-)");
    }

    public static void StampaKO() {
        // per ora su System.out, come esercizio lasciato agli studenti
        // e/o ad esercitazione, metterlo in ambiente grafico
        System.out.println("ALLARME: non ci sono piu' soldi nel conto");
    }

    public static void MandaMessaggio(String messaggio) {
        System.out.println(messaggio);
    }
}
```

per lo scambio dei messaggi che sia adatto alla concorrenza. Naturalmente assumeremo che le azioni associate alle transizioni dei diagrammi degli stati e delle transizioni avvengano attraverso il meccanismo dei task introdotto sopra per garantire l'integrità dell'informazione (degli oggetti JAVA che realizzano l'istanziazione corrente) del diagramma delle classi (si veda dopo).

Operazioni di gestione eventi

Questo sono operazioni che per l'inizializzazione del meccanismo i gestione degli eventi, l'attivazione e disattivazione di tale meccanismo, e l'introduzione di eventi da inviare ai vari oggetti dotati di (una realizzazione in JAVA del) diagramma degli stati e delle transizioni. Tutte queste operazioni hanno a che fare con il meccanismi di gestione degli eventi che è chiaramente completamente esterno al diagramma delle attività.

Come per le operazioni di ingresso/uscita, anche per le operazioni di gestione eventi non diamo alcuna strategia realizzativa specifica. Semplicemente le realizzeremo come funzioni o funtori con opportuni esecutori.

Assumeremo però che gli oggetti dotati di (una realizzazione in JAVA del) diagramma degli stati e delle transizioni vengano eseguiti su thread separati dai thread delle attività e di avere un meccanismo di gestione degli eventi

Attività complesse

Realizziamo le attività complesse (sia l'attività principale che le sotto attività complesse) come dei funtori il cui esecutore è la classe `Thread` di JAVA. Più precisamente:

- il funtore stesso implementa `Runnable`;
- il codice da eseguire corrispondente al flusso di controllo dell'attività complessa è nel corpo del metodo `run()`;
- l'esecutore del funtore è la classe `Thread` predefinita in JAVA che a sua volta chiama il codice da eseguire (`run()`) attraverso `start()`.

Si noti che in questo modo tutte le attività complesse verranno eseguite su thread separati. Useremo `join()` (metodo definito in `Thread`) per correlarne l'esecuzione con l'esecuzione dell'attività che le ha invocate. In altre parole attiveremo attività complesse attraverso un fork (`start()`) dopo di che aspetteremo l'esecuzione della stessa attraverso un join (`join()`). Naturalmente in questo modo possiamo attivare più sotto attività da eseguire concorrentemente se necessario.

Nota: per semplicità assumeremo che ogni volta che il diagramma delle attività richieda un fork su più attività concorrenti, queste siano sempre complesse (anche se poi all'interno contengono solo una attività atomica).

Costrutti sequenziali per il controllo di flusso

Per quanto riguarda i **costrutti sequenziali** del diagramma delle attività (cioè le **sequenze**, le **guardie**, i **nodi condizionale** e i **nodi merge**), faremo uso direttamente delle strutture di controllo sequenziali messe a disposizione da JAVA. Useremo quindi:

- sequenza di istruzioni
- `if-else`, `while`, `do-while`.

Costrutti concorrenti per il controllo di flusso

Per quanto riguarda i **costrutti concorrenti** del diagramma delle attività, useremo:

- `Thread.start()` per il **fork**
- `Thread.join()` per il **join**.

Per semplicità, e senza perdita di generalità, richiederemo che ciascun ramo di un fork sia incapsulato in una sotto attività complessa.

Attività complessa: esempio

```
package attivita_complesse;

import attivita_atomiche.*;
import attivita_io.*;
import persona.*;
import _framework.*;
import conto.*;

public class AttivitaPrincipale implements Runnable {

   private boolean eseguita = false;

   public synchronized void run() {
      if (eseguita == true)
         return;
      eseguita = true;

      Persona marito;
      Persona moglie;
      Conto conto;

      //operazione IO
      RecordPersonaPersonaConto ris =
         AttivitaIO.LeggiDatiConto();
```

```
marito = ris.marito;
moglie = ris.moglie;
conto = ris.conto;

//task
AperturaConto ac = new AperturaConto(marito, moglie, conto);
Executor.perform(ac);

//operazione IO
AttivitaIO.AperturaOK();

while (numeroOperazioni(conto) < 15) { //NB verifica attraverso un task

   //attivita' complessa
   Thread ramo1 = new Thread(new AttivitaSottoramo1(moglie, conto));
   ramo1.start(); //FORK: primo ramo

   //attivita' complessa
   Thread ramo2 = new Thread(new AttivitaSottoramo2(marito, conto));
   ramo2.start(); //FORK: secondo ramo

   try {
      ramo1.join(); //JOIN: primo ramo
      ramo2.join(); //JOIN: seconso ramo
   } catch (InterruptedException e) {
      e.printStackTrace();
      }

   }

   //task
   Saldo s= new Saldo(conto);
   Executor.perform(s);

   if (s.getSaldo() > 0)
      //operazione IO
      AttivitaIO.StampaOK();
   else
      //operazione IO
      AttivitaIO.StampaKO();
}

private int numeroOperazioni(Conto conto) {
   //task
   NumeroOperazioni no = new NumeroOperazioni(conto);
   Executor.perform(no);
   return no.getNum();
}

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

Attività complessa: esempio

Commentiamo brevemente l'esempio:

- Innanzitutto si nota immediatamente che `AttivitaPrincipale` realizza come richiesto il pattern funtore, avendo come esecutore la classe `Thread`.

- Il corpo del funtore costituito dal corpo di `run` contiente l'intero processo associato alla attività complessa.

- Vengono definite tre variabili per l'attività: `moglie`, `marito`, `conto`.

- Il flusso di controllo è inizialmente sequenziale:

  - Legge i dati dal conto (si noti che `AttivitaIO.LeggiDatiConto()` restituisce un record `ris` formato da tre campi dati pubblici: `marito`, `moglie`, `conto`. Questa attività di ingresso/uscita prenderà in qualche modo i dati necessari dall'utente dell'applicazione.

  - Apre il conto stesso immettendo i dati raccolti nel diagramma delle classi attraverso una opportuna attività atomica, il task `AperturaConto`.

  - Manda in output l'ok sulla apertura conto (operazione di ingresso/uscita).

- A questo punto troviamo un `while`. Si noti che il test stesso del `while` è basato su un task (qui incapsulato in un metodo privato `numeroOperazioni()`) poiché richiede l'accesso al diagramma delle classi.

- Nel corpo del `while` abbiamo un fork in due sottoattività complesse concorrenti, `AttivitaSottoramo1` e `AttivitaSottoramo2` e poi il join delle stesse.

- All'uscita del `while` troviamo un task per recuperare l'ammontare del conto e un `if-else` utilizzato per selezionare la giusta operazione di uscita da effettuare.

Per completezza includiamo il codice delle altre attività complesse e delle attività atomiche utilizzate.

```
   }

   public synchronized boolean estEseguita() {
      return eseguita;
   }

}
```

Attività complessa: AttivitaSottoramo1

```
package attivita_complesse;

import attivita_atomiche.*;
import conto.*;
import persona.*;
import _framework.*;

public class AttivitaSottoramo1 implements Runnable {

   private boolean eseguita = false;
   private Persona persona;
   private Conto conto;

   public AttivitaSottoramo1(Persona p, Conto c) {
      persona = p;
      conto = c;
   }

   public synchronized void run() {
      if (eseguita == true)
         return;
      eseguita = true;
      Executor.perform(new Depositare(persona, conto));
      Executor.perform(new Prelevare(persona, conto));
```


Attività complessa: AttivitaSottoramo2

```
package attivita_complesse;

import attivita_atomiche.*;
import conto.*;
import persona.*;
import _framework.*;

public class AttivitaSottoramo2 implements Runnable {

   private boolean eseguita = false;
   private Persona persona;
   private Conto conto;

   public AttivitaSottoramo2(Persona p, Conto c) {
      persona = p;
      conto = c;
   }

   public synchronized void run() {
      if (eseguita == true)
         return;
      eseguita = true;
      Executor.perform(new Prelevare(persona, conto));
   }
```

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

Attività atomica (task): AperturaConto

```
package attivita_atomiche;

import _framework.*;
import conto.*;
import persona.*;
import possedere.*;

public class AperturaConto implements Task {

   private boolean eseguita = false;

   private Persona personaInput_1;
   private Persona personaInput_2;
   private Conto contoInput;

   public AperturaConto(Persona p1, Persona p2, Conto c) {
      personaInput_1 = p1;
      personaInput_2 = p2;
      contoInput = c;
   }

   public synchronized void esegui(Executor e) {
      if (e == null || eseguita == true)
         return;
```



```
      eseguita = true;
      TipoLinkPossedere poss1 = new TipoLinkPossedere(personaInput_1,
            contoInput);
      ManagerPossedere.inserisci(poss1);
      TipoLinkPossedere poss2 = new TipoLinkPossedere(personaInput_2,
            contoInput);
      ManagerPossedere.inserisci(poss2);
   }

   public synchronized boolean estEseguita() {
      return eseguita;
   }

}
```

Attività atomica (task): Depositare

```
package attivita_atomiche;

import attivita_io.*;
import _framework.*;
import conto.*;
import persona.*;

public class Depositare implements Task {

   private boolean eseguita = false;
   private Persona persona;
   private Conto conto;

   public Depositare(Persona p, Conto c) {
      persona = p;
      conto = c;
   }

   public synchronized void esegui(Executor e) {
      if (e == null || eseguita == true)
         return;
      eseguita = true;
      AttivitaIO.MandaMessaggio(persona.getNome()
            + " sta depositando dei soldi ...");
```

```
      conto.aumenta(persona, (int) (Math.random() * 900));
      AttivitaIO.MandaMessaggio("... il conto ora ammonta a "
            + conto.getSaldo());
   }

   public synchronized boolean estEseguita() {
      return eseguita;
   }

}
```

Attività atomica (task): Prelevare

```
package attivita_atomiche;

import attivita_io.*;
import conto.*;
import persona.*;
import _framework.*;

public class Prelevare implements Task {

   private boolean eseguita = false;
   private Persona persona;
   private Conto conto;

   public Prelevare(Persona p, Conto c) {
      persona = p;
      conto = c;
   }

   public synchronized void esegui(Executor e) {
      if (e == null || eseguita == true)
         return;
      eseguita = true;
      AttivitaIO.MandaMessaggio(persona.getNome()
            + " sta facendo una spesa ...");
```



```
      conto.diminuisci(persona, (int) (Math.random() * 1000));
      AttivitaIO.MandaMessaggio("... il conto ora ammonta a "
            + conto.getSaldo());
   }

   public synchronized boolean estEseguita() {
      return eseguita;
   }

}
```

Attività atomica (task): NumeroOperazioni

```
package attivita_atomiche;

import _framework.*;
import conto.*;

public class NumeroOperazioni implements Task {

   private boolean eseguita = false;

   private Conto conto;
   private int num;

   public NumeroOperazioni(Conto c) {
      conto = c;
   }

   public synchronized void esegui(Executor e) {
      if (e == null || eseguita == true)
         return;
      eseguita = true;
      num = conto.getNumeroOperazioni();
   }

   public int getNum() {
```

```
      if (!eseguita) throw new RuntimeException("Risultato non pronto");
      return num;
   }

   public synchronized boolean estEseguita() {
      return eseguita;
   }

}
```

Attività atomica (task): Saldo

```
package attivita_atomiche;

   import _framework.*;
   import conto.*;

   public class Saldo implements Task {

      private boolean eseguita = false;

      private Conto conto;
      private int saldo;

      public Saldo(Conto c) {
         conto = c;
      }

      public synchronized void esegui(Executor e) {
         if (e == null || eseguita == true)
            return;
         eseguita = true;
         saldo = conto.getSaldo();
      }

      public int getSaldo() {
```



```
         if (!eseguita) throw new RuntimeException("Risultato non pronto");
         return saldo;
      }

      public synchronized boolean estEseguita() {
         return eseguita;
      }

   }
```