# Esercizi sul linguaggio Java

**Mario G. Cimino, G. Dini**

**Pisa, 2005**

**Il presente materiale è stato prodotto durante l'attività didattica per il Corso di Tecnologie Informatiche per la Gestione Aziendale (CdL in Ing. Informatica per la Gestione d'Azienda) degli a.a. 2003-04, e 2004-05.**

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

7 GIUGNO 2004

Scrivere un programma Java che implementi il servizio di *sincronizzazione a barriera* per un insieme di thread (Fig.1).

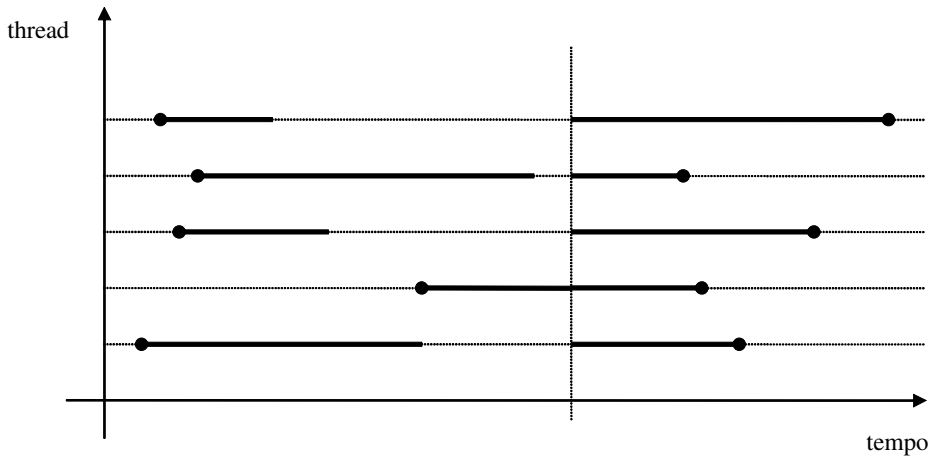



**Fig.1 –** Sincronizzazione a barriera

Ciascun thread attende per un tempo random e poi esegue sulla barriera un metodo `waitAll()` che risulta bloccante sino a che è l'ultimo thread ad invocarlo, ed in tal caso esso risveglia tutti i thread.

---

**ESEMPIO DI FUNZIONAMENTO:**

```
> java Simulazione 5
 Thread  n.0 partito...
 Thread  n.0 sospeso...
 Thread  n.1 partito...
 Thread  n.2 partito...
 Thread  n.3 partito...
 Thread  n.1 sospeso...
 Thread  n.4 partito...
 Thread  n.3 sospeso...
 Thread  n.2 sospeso...
 Thread  n.4 sospeso...
 --------------------
 Thread  n.4 risvegliato...
 Thread  n.1 risvegliato...
 Thread  n.2 risvegliato...
 Thread  n.0 risvegliato...
 Thread  n.3 risvegliato...
 Thread  n.1 terminato
 Thread  n.4 terminato
 Thread  n.2 terminato
 Thread  n.0 terminato
 Thread  n.3 terminato
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

7 GIUGNO 2004

**SOLUZIONE:**

```
// RandLifeThread.java
public class RandLifeThread extends Thread {
    private Barriera barriera;
    public RandLifeThread(String name, Barriera barriera)   {
        super(name);
        this.barriera = barriera;
    }
    public void run()   {
        System.out.println(" Thread "+ getName() + " partito...");
        Simulazione.attesaRandom();
        System.out.println(" Thread "+ getName() + " sospeso...");
        barriera.waitAll();
        System.out.println(" Thread "+ getName() + " risvegliato...");
        Simulazione.attesaRandom();
        System.out.println(" Thread "+ getName() + " terminato");
    }
}

// Barriera.java
public class Barriera {
    int contaThread;
    public Barriera(int numThread) {
        contaThread = numThread;
    }
    public synchronized void waitAll() {
        if (contaThread > 1) {
            contaThread--;
            try {
                wait();
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        else {
            System.out.println(" ----------------------");
            notifyAll();
        }
    }
}

// Simulazione.java
public class Simulazione {
    public static void attesaRandom() {
        try {
            Thread.sleep(500 + (long)(Math.random()*2500));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        final int N = Integer.parseInt(args[0]);
        Barriera barriera = new Barriera(N);
        for (int i=0; i<N; i++) {
            new RandLifeThread(" n." + i, barriera).start();
            attesaRandom();
        }
    }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

25 GIUGNO 2004

Un insieme di merci è costituito da diversi tipi di merci e per ogni merce da un certo numero di articoli. La classe Java StockSet realizza un insieme ordinato di merci che ha i seguenti costruttori e metodi

- **public** StockSet(**int** numberOfStocks) che definisce un insieme costituito da un numero di merci pari a numberOfStocks. Il costruttore imposta a zero il numero di articoli di ciascun tipo di merce.
- **public** String toString() che converte uno StockSet in uno String.
- **public boolean** geq(StockSet other) **throws** ArrayIndexOutOfBoundsException che ritorna **true** se questo insieme di merci è maggiore o uguale dell'insieme di merci other specificato come argomento. Un insieme di merci è maggiore o uguale di un altro se, per ogni tipo di merce, il numero di articoli nel primo insieme è maggiore o uguale del numero di articoli nel secondo insieme. I due insiemi devono essere costituiti dallo stesso numero di merci; altrimenti viene sollevata l'eccezione ArrayIndexOutOfBoundsException.
- **public void** order(StockSet ordered) **throws** ArrayIndexOutOfBoundsException che sottrae l'insieme di merci ordered da questo insieme di merci. La sottrazione di un insieme di merci da un altro consiste nel sottrarre, per ciascuna merce, il numero degli articoli del primo da quello del secondo. I due insiemi devono essere costituiti dallo stesso numero di merci; altrimenti viene sollevata l'eccezione ArrayIndexOutOfBoundsException.
- **public void** supply(StockSet supplied) **throws** ArrayIndexOutOfBoundsException che somma l'insieme di merci supplied a questo insieme di merci. La somma di un insieme di merci ad un altro consiste nel sommare, per ciascuna merce, il numero degli articoli del primo a quello del secondo. I due insiemi devono essere costituiti dallo stesso numero di merci; altrimenti viene sollevata l'eccezione ArrayIndexOutOfBoundsException.
- **public void** initRandom(int maxItems) che imposta il numero di articoli di ciascuna merce di questo insieme ad un valore casuale compreso tra 0 e maxItems.
- **public void** numberOfStocks() che ritorna il numero di tipi di merci di questo insieme.

Un *magazzino* è uno StockSet in cui le operazioni di order e supply possono essere eseguite in modo concorrente in accordo alla seguente condizione di sincronizzazione: l'esecuzione di un ordine può essere completata se la merce in magazzino è maggiore di quella ordinata; altrimenti, l'ordine deve essere sospeso fino a quando il magazzino non sarà approvvigionato.

**Esercizio n. I.** Il candidato realizzi la classe Stockhouse che modella un magazzino

**Esercizio n. II.** Il candidato realizzi inoltre un timer task di nome Report che va in esecuzione periodicamente ogni 3 secondi e stampa lo stato del magazzino per mezzo del metodo print.

```
>java Simulazione 3
 Content:       [       0       0       0       ]
 Order  of t0: -[       7       8       3       ]
 Supply of t1: +[       7       2       0       ]
 Supply of t2: +[       4       7       2       ]
 Content:       [       11      9       2       ]
 Order  of t3: -[       3       1       1       ]
 Order  of t3   dispatched
 Content:       [       8       8       1       ]
 Supply of t4: +[       3       5       7       ]
 Order  of t0   dispatched
 Content:       [       4       5       5       ]
```

**Fig.1 –** Esempio di funzionamento

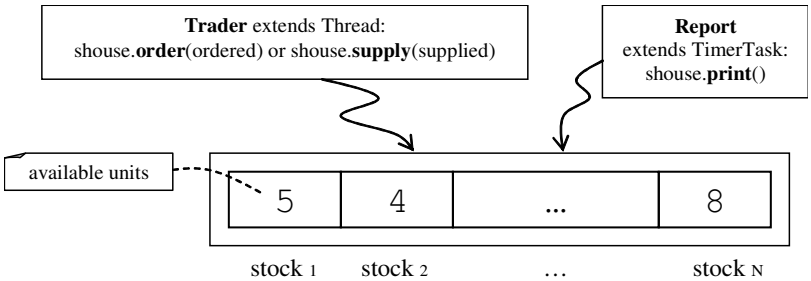



**Fig.2 –** Struttura dell'oggetto `shouse` di classe `Stockhouse`

## TRACCIA PROPOSTA AL CANDIDATO:

```
// StockSet.java
public class StockSet {

    private int[]               stock;

    public StockSet(int numberOfStocks) {
        stock = new int[numberOfStocks];
        for (int i = 0; i < stock.length; i++)
            stock[i] = 0;
    }

    public String toString() {
        String result = "[\t";
        for (int i = 0; i < stock.length; i++)
            result += stock[i] + "\t";
        return result + "]";
    }

    public boolean geq(StockSet other) throws ArrayIndexOutOfBoundsException {
        if (stock.length!=other.numberOfStocks())
            throw new ArrayIndexOutOfBoundsException();

        for (int i = 0; i < stock.length; i++)
            if (stock[i] < other.stock[i])
                return false;
        return true;
    }

    public void order(StockSet ordered) throws ArrayIndexOutOfBoundsException {
        if (stock.length!=ordered.numberOfStocks())
            throw new ArrayIndexOutOfBoundsException();

        for (int i = 0; i < stock.length; i++)
            stock[i] -= ordered.stock[i];
    }

    public void supply(StockSet supplied) throws ArrayIndexOutOfBoundsException {
        if (stock.length!=supplied.numberOfStocks())
            throw new ArrayIndexOutOfBoundsException();

        for (int i = 0; i < stock.length; i++)
            stock[i] += supplied.stock[i];
    }

    public void initRandom(int maxItems) {
        for (int i = 0; i < stock.length; i++)
            stock[i] = (int)(Math.random()*maxItems);
    }

    public int numberOfStocks() {
        return stock.length;
    }
}

// Stockhouse.java
public class Stockhouse extends StockSet {

    // private part
    private static final int    DEFAULT_NUM_STOCKS = 5;

    public Stockhouse() {
        // …
    }

    public Stockhouse(int numberOfStocks) {
        // …
    }

    public synchronized void supply(StockSet supplied) {
        // …
    }

    public synchronized void order(StockSet ordered) {
        // …
    }

    public synchronized void print() {
        // …
    }

    public synchronized int numberOfStocks() {
```

```
        // …
    }
}

// Trader.java
public class Trader extends Thread {

    private Stockhouse house;

    public Trader(String name, Stockhouse house) {
        super(name);
        this.house = house;
    }

    public void run() {
        StockSet stock = new StockSet(house.numberOfStocks());
        stock.initRandom(10);

        if (Math.random()>= 0.5) {
            System.out.println (" Order  of " + getName() + ": -" + stock);
            house.order(stock);
            System.out.println (" Order  of " +  getName() + "   dispatched" );
        }
        else {
            System.out.println (" Supply of " + getName() + ": +" + stock);
            house.supply(stock);
        }
    }
}

// Report.java
import java.util.TimerTask;

public class Report extends TimerTask {
  // …
}

// Simulazione.java
import java.util.Timer;

public class Simulazione {

    public static void attesaRandom() {
        try {
            Thread.sleep(500 + (long)(Math.random()*2500));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        final int N = Integer.parseInt(args[0]);

        Stockhouse house = new Stockhouse(N);

        // qui generare il TimerTask  Report e schedularlo

        for (int i=0; i<5; i++) {
            new Trader("t" + i, house).start();
            attesaRandom();
        }
   }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

25 GIUGNO 2004

**SOLUZIONE DELLE PARTI DA SVOLGERE:**

## Esercizio n. I:

```
// Stockhouse.java
public class Stockhouse extends StockSet {

    private static final int    DEFAULT_NUM_STOCKS = 5;

    public Stockhouse() {
        super(DEFAULT_NUM_STOCKS);
    }

    public Stockhouse(int numberOfStocks) {
        super(numberOfStocks);
    }

    public synchronized void supply(StockSet supplied) {
        super.supply(supplied);
        notifyAll();
    }

    public synchronized void order(StockSet ordered) {
        while (!geq(ordered))
            try {
                wait();
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        super.order(ordered);
   }

   public synchronized void print() {
        System.out.println(" Content:       " + this);
   }

   public synchronized int numberOfStocks() {
        return super.numberOfStocks();
   }
}
```

## Esercizio n. II:

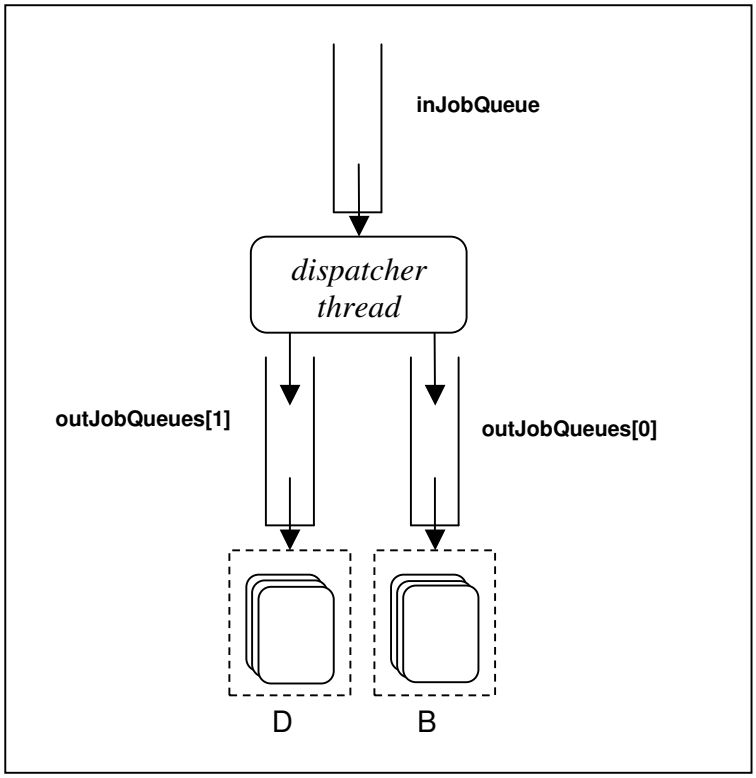
```
// Report.java
import java.util.TimerTask;

public class Report extends TimerTask {

    private Stockhouse house;

    public Report(Stockhouse house) {
        super();
        this.house = house;
    }

    public void run() {
        house.print();
    }
}

// Simulazione.java
// parte che genera e schedula il TimerTask  Report

Timer timer = new Timer();
timer.schedule(new Report(house),0,3000);
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

16 LUGLIO 2004

Si consideri la versione semplificata del kernel SAP® riportata in figura. Il *dispatcher thread* riceve un *job* dalla coda di ingresso inJobQueue e lo accoda sulla coda outJobQueues[0] se è un *batch job*, ovvero sulla coda di uscita outJobQueues[1] se è un *dialog job*. Ogni coda di uscita è servita da un diverso pool di *working thread*. Ciascun working thread è un ciclo senza fine costituito dalle seguenti azioni: estrazione di un job dalla coda di pertinenza; esecuzione del job estratto.

La coda di uscita outJobQueues[0] è servita da un pool di thread aventi tutti la minima priorità di esecuzione, mentre la coda di uscita outJobQueues[1] è servita da un pool di thread aventi tutti la massima priorità di esecuzione.




Assumendo che un job sia un oggetto dotato di un metodo Object execute(Object arg) e che le classi Job, DialogJob e BatchJob siano date, il candidato realizzi le seguenti classi

- JobQueue
- DispatcherThread
- WorkingThread;

sotto le seguenti ipotesi:

- una JobQueue ha una lunghezza statica e definita al momento della creazione della coda stessa;
- una JobQueue ha un metodo pubblico void put(Job j) per inserire un job in coda ed il metodo pubblico Job get() per estrarre un job dalla coda; il metodo put è bloccante se la coda è piena; il metodo get è bloccante se la coda è vuota.

Il candidato definisca anche il codice che crea i pool di thread come gruppi di thread di nome batchGroup e dialogGroup rispettivamente.

**ESEMPIO DI ESECUZIONE:**

Nell'esempio che segue la coda di ingresso ha lunghezza 3, le altre due code sono lunghe la metà; il metodo execute riceve per argomento un numero reale random.

| *OUTPUT* | *COMMENTI* |
|---|---|
| >java Simulazione 3 | |
| put BatchJob 0 into inputQueue | *- Inserimenti in coda di ingresso da parte del* ***main*** |
| put DialogJob 1 into inputQueue | " |
| put BatchJob 2 into inputQueue | " |
| get BatchJob 0 from inputQueue | *- Il* ***Dispatcher****(pr.NORMAL) preleva il 1° Job (batch) e lo colloca nella rispettiva coda* |
| put BatchJob 0 into batchQueue | |
| get BatchJob 0 from batchQueue | *- Un* ***Working*** *in attesa (pr.MIN) lo preleva…* |
| get DialogJob 1 from inputQueue | *- Ma il* ***Dispatcher****(pr.NORMAL) riparte e preleva il 2° Job (dialog) e lo colloca nella rispettiva coda* |
| put DialogJob 1 into dialogQueue | |
| get DialogJob 1 from dialogQueue | *- Un* ***Working*** *in attesa (pr.MAX) lo preleva subito… …e lo esegue indisturbato* |
| exe DialogJob 1 with arg 0.016035349585280412 | |
| get BatchJob 2 from inputQueue | *- Il* ***Dispatcher****(pr.NORMAL) riprende, preleva il 3° Job (batch) e lo colloca nella rispettiva coda* |
| put BatchJob 2 into batchQueue | |
| get BatchJob 2 from batchQueue | *- Un* ***Working*** *in attesa (pr.MIN) lo preleva e lo esegue* |
| exe BatchJob 2 with arg 0.14529240370504637 | |
| exe BatchJob 0 with arg 0.5431222373190386 | *- Finalmente il primo* ***Working*** *(pr.MIN) esegue il Job* |

**TRACCIA PROPOSTA AL CANDIDATO:**

```
// Job.java
public abstract class Job {
```

```
    private int id;

    public Job(int id) {
        this.id = id;
    }

    public Object execute(Object arg) {
        return  " exe " + getClass().getName() + " " + id + " with \targ " + arg;
    }

    public int getId() {
        return id;
    }
}
// DialogJob.java
public class DialogJob extends Job {
    public DialogJob(int id) {
        super(id);
    }

    public Object execute(Object arg) {
        Simulazione.elaborazioneRandom(1.0);
        return super.execute(arg);
    }
}

// BatchJob.java
public class BatchJob extends Job {
    public BatchJob(int id) {
        super(id);
    }

    public Object execute(Object arg) {
        Simulazione.elaborazioneRandom(10.0);
        return super.execute(arg);
    }
}
// Simulazione.java
public class Simulazione {

    public static void attesaRandom(double maxSec) {
        try {
            Thread.sleep(500 + (long)(Math.random()* 1000 * maxSec));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void elaborazioneRandom(double maxSec) {
        long period = (long)(Math.random()* 1000 * maxSec);
        long start  = new java.util.Date().getTime();
        long now;
        do { now    = new java.util.Date().getTime();   }
        while((now-start) < period );
    }
    public static void main(String[] args) {
        final int MAX_SIZE = Integer.parseInt(args[0]);
        JobQueue    inJobQueue   =   new JobQueue(MAX_SIZE, "inputQueue");
        for (int i=0; i<MAX_SIZE; i++)
            if (Math.random() >= 0.5)
                inJobQueue.put(new BatchJob(i));
            else
                inJobQueue.put(new DialogJob(i));

        JobQueue[]  outJobQueues = { new JobQueue(MAX_SIZE/2, "batchQueue"),
                                     new JobQueue(MAX_SIZE/2, "dialogQueue")
        };

        new DispatcherThread(inJobQueue,outJobQueues).start();
           // QUI GENERARE I POOL DI WORKING THREAD DI DIALOGO/BATCH COME THREADGROUP

    }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

16 LUGLIO 2004

## SOLUZIONE DELLE PARTI DA SVOLGERE:

```
// Simulazione.java
[…]
        // QUI GENERARE I POOL DI WORKING THREAD DI DIALOGO E BATCH

        ThreadGroup batchGroup = new ThreadGroup("");
        for (int i=0; i<5; i++)
            new WorkingThread(batchGroup, outJobQueues[0], Thread.MIN_PRIORITY).start();

        ThreadGroup dialogGroup = new ThreadGroup("");
        for (int i=0; i<5; i++)
            new WorkingThread(dialogGroup, outJobQueues[1], Thread.MAX_PRIORITY).start();
[…]

// JobQueue.java
import java.util.*;

public class JobQueue {

    private LinkedList      queue;
    private final int       MAX_SIZE;
    private String          name;

    public JobQueue(int maxSize, String name) {
        queue = new LinkedList();
        MAX_SIZE = maxSize;
        this.name = name;
    }

    public synchronized void put(Job job) {
        while (queue.size() == MAX_SIZE)
            try {
                wait();
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        queue.addLast(job);
        System.out.println(" put " + job.getClass().getName() + " " + job.getId() + " into\t" + name);
        notify();
    }

    public synchronized Job get() {
        while(queue.size() == 0)
            try {
                wait();
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        Job job = (Job)queue.removeFirst();
        System.out.println(" get " + job.getClass().getName() + " " + job.getId() + " from\t" + name);
        notify();
        return job;
    }
}

// DispatcherThread.java
public class DispatcherThread extends Thread {
    private JobQueue     inJobQueue;
    private JobQueue[]   outJobQueues;

    public DispatcherThread(JobQueue inJobQueue, JobQueue[] outJobQueues) {
        this.inJobQueue = inJobQueue;
        this.outJobQueues = outJobQueues;
    }

    public void run() {
        while(true) {
            Simulazione.attesaRandom(1.0);
            Job j = inJobQueue.get();
            if (j instanceof BatchJob)
                outJobQueues[0].put(j);
```

```
            else
                outJobQueues[1].put(j);
        }
    }
}



// WorkingThread.java
public class WorkingThread extends Thread {
    private JobQueue jobQueue;

    public WorkingThread(ThreadGroup group, JobQueue jobQueue, int priority) {
        super(group,"");
        this.jobQueue = jobQueue;
        this.setPriority(priority);
    }

    public void run() {
        while(true) {
            Simulazione.attesaRandom(1.0);
            System.out.println(jobQueue.get().execute(new Double(Math.random())));
        }
    }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

20 SETTEMBRE 2004

Si consideri la classe `Transaction` che realizza il servizio di *transazione distribuita*. Per semplicità una transazione distribuita sia composta da un numero predefinito *N* di statement SQL di tipo *insert/update/delete*, ciascuno dei quali è eseguito su di un diverso database. Si assuma inoltre che i database siano tutti del medesimo DBMS (driver).

La classe `Transaction` è caratterizzata dai seguenti costruttori e metodi:

- **public** `Transaction(String dbDriver,` **int** `nStmt)` **throws** `ClassNotFoundException` che definisce una transazione composta da `nStmt` statement, caricando il driver `dbDriver` e sollevando una eccezione in caso di mancato caricamento.
- **public void** `getConnections(String[] dbUrls)` **throws** `SQLException` che si connette a tutti i database di questa transazione, localizzati mediante `dbUrls`, rilasciando tutte le risorse e sollevando una eccezione in caso di impossibilità di connessione ad (almeno) uno dei database.
- **public void** `prepareTransaction(String[] sqlStmt, Vector[] pars)` **throws** `SQLException` che crea tutti i `PreparedStatement` di questa transazione definiti dalle stringhe `sqlStmt` e dai parametri `pars`[1]. Il metodo rilascia tutte le risorse e genera una eccezione in caso di impossibilità a creare almeno uno degli statement.
- **public void** `executeTransaction()` **throws** `SQLException` che esegue tutti gli statement di questa transazione. In caso di impossibilità ad eseguire (almeno) uno degli statement, il metodo provvede a ripristinare lo stato di tutti i database, rilasciare tutte le risorse e sollevare una eccezione[2].

La classe `AccountReference` realizza un riferimento ad un conto corrente, composto dall'indirizzo IP dell'host di residenza del database, il nome del database e l'*id* del conto corrente. La classe `Purchase` implementa infine un servizio di `reliablePayment(…)` adoperando le classi `AccountReference` e `Transaction` per eseguire in modo affidabile il pagamento on line di un bene come transazione, costituita da un prelievo ed un versamento su due diversi database rispettivamente.

**Esercizio n. I.** Implementare la classe `Transaction` in modo *thread-unsafe*, avendo a disposizione tutte le altre classi e due database locali[3] contenenti entrambi la seguente tabella:

**Fig.1** - Tabella ACCOUNT

| ID | BALANCE |
|---|---|
| 1 | 100.00 |

**Esercizio n. II.** Aggiungere, al messaggio contenuto nella `SQLException` rilanciata dai metodi di `Transaction`, una stringa che specifichi quale tra le `N` transazioni elementari ha causato l'eccezione. Inoltre nel metodo `executeTransaction()` generare una `SQLException` di tipo `SQLWarning("\n Record not found.")`, nel caso in cui (almeno) una esecuzione non abbia coinvolto alcun record[4]. Anche in tal caso rilasciare tutte le risorse e ripristinare lo stato dei database.

---

[1] Ciascun *`Vector pars[i]`* può contenere un numero arbitrario di parametri, dai tipi non noti a priori. A tale scopo, si consiglia di adoperare il metodo *`setObject(int, Object)`* per predisporre i *PreparedStatement*. In tal modo è il driver ad occuparsi del riconoscimento del tipo dell'oggetto Java e della relativa conversione in tipo SQL, secondo un mapping standard definito dalle specifiche JDBC.

[2] A tale scopo, si consiglia di adoperare il supporto al salvataggio/ripristino dello stato delle esecuzioni fornito dall'interfaccia *`java.sql.Connection`* tramite il meccanismo del *rollback*.

[3] Registrare i due database forniti, *bank1.mdb* e *bank2.mdb*, come sorgente ODBC dai nomi *bank1* e *bank2* rispettivamente.

[4] Il metodo *`executeUpdate`* ritorna un intero che rappresenta il numero di record coinvolti nella esecuzione. Ad esempio, ritorna *`0`* se la clausola *`WHERE`* non ha identificato alcun record nella tabella, senza sollevare alcuna eccezione.

## PROVE DI ESECUZIONE[5]:

a) Se ogni operazione va a buon fine, una singola *Simulazione* produce `BALANCE 90.00` in *bank1* e `110.00` in *bank2*.
b) Se il database *bank2* (o *bank1*) viene rinominato, una *Simulazione* visualizza dei messaggi di errore sulla mancata connessione, senza produrre alcuna modifica sulle tabelle.
c) Se la tabella `BALANCE` di *bank2* (o *bank1*) viene rinominata, una *Simulazione* visualizza dei messaggi di errore sulla mancata esecuzione, senza produrre alcuna modifica sulle tabelle.
d) Se il record della tabella `BALANCE` di *bank2* (o *bank1*) viene modificato, dando al campo `ID` un valore diverso da `1`, una *Simulazione* visualizza il messaggio della `SQLWarning`, senza produrre alcuna modifica sulle tabelle.

## TRACCIA PROPOSTA AL CANDIDATO:

```
// AccountReference.java

import java.net.*;

public class AccountReference {
    private InetAddress ipAddr;
    private String      dbName;
    private int         accountId;

    public AccountReference(InetAddress ipAddr, String dbName, int accountId) {
        this.ipAddr     = ipAddr;
        this.dbName     = dbName;
        this.accountId  = accountId;
    }

    public String getDbUrl() {
        return "jdbc:odbc:" + dbName;
    }

    public int getAccountId() {
        return accountId;
    }
}


// Purchase.java

import java.sql.*;
import java.util.*;

public class Purchase {

    public void reliablePayment( double price, AccountReference customerAccount,
                                               AccountReference supplierAccount )
                                               throws SQLException, ClassNotFoundException {
        String[] dbUrls = {
            customerAccount.getDbUrl(),
            supplierAccount.getDbUrl()
        };

        Vector[] pars = {new Vector(2), new Vector(2)};
            pars[0].add(new Double(price));
            pars[0].add(new Integer(customerAccount.getAccountId()));
            pars[1].add(new Double(price));
            pars[1].add(new Integer(supplierAccount.getAccountId()));

            Transaction transaction = new Transaction(dbDriver,2);
            transaction.getConnections(dbUrls);
            transaction.prepareTransaction(sqlStats, pars);
            transaction.executeTransaction();
    }

  private static final String    dbDriver = "sun.jdbc.odbc.JdbcOdbcDriver";

  private static final String[]  sqlStats = {
        "UPDATE ACCOUNT SET BALANCE = BALANCE - ? WHERE ID = ?",
        "UPDATE ACCOUNT SET BALANCE = BALANCE + ? WHERE ID = ?"
  };
}
```

[5] È possibile che, chiudendo e riaprendo una tabella dall'interfaccia utente di *MicrosoftAccess*®, dopo averla modificata mediante l'esecuzione di una *Simulazione*, riappaiano i vecchi valori in quanto la cache non è stata automaticamente aggiornata. In caso di dubbio basta chiudere e riaprire una seconda volta la tabella per indurre un aggiornamento della cache.

```
// Simulazione.java

import java.sql.*;
import java.net.*;

public class Simulazione {

    public static void main(String[] args) {
        try {
        Purchase purchase = new Purchase();
        purchase.reliablePayment( 10.0, new AccountReference(InetAddress.getLocalHost(), "bank1", 1),
                                        new AccountReference(InetAddress.getLocalHost(), "bank2", 1)
                                );
        }
        catch (SQLException e) {
            gestoreEccezioneSQL(e);
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }

    private static void gestoreEccezioneSQL(SQLException e) {
        System.err.println("Message: " + e.getMessage() );
        System.err.println("SQLState: " + e.getSQLState() );
        System.err.println("ErrorCode: " + e.getErrorCode() );
        e.printStackTrace();
    }

}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

20 SETTEMBRE 2004

## SOLUZIONE DELLE PARTI DA SVOLGERE:

```
// Transaction.java

import java.sql.*;
import java.util.*;

public class Transaction {
    private Connection[]        conns;
    private PreparedStatement[] stmt;

    public Transaction(String dbDriver, int nStmt) throws ClassNotFoundException {
        Class.forName(dbDriver);
        conns = new Connection[nStmt];
        stmt = new PreparedStatement[nStmt];
    }

    public void getConnections(String[] dbUrls) throws SQLException {
        int i = 0;
        try {
            for (i=0; i<conns.length; i++)
                conns[i] = DriverManager.getConnection(dbUrls[i]);
        }
        catch (SQLException e) {
            for (int t=i; t>=0; t--)
                if (conns[t]!=null)
                    conns[t].close();
            throw new SQLException(e.getMessage() + "\n On connection n. " + (i+1),
                                   e.getSQLState(), e.getErrorCode()) ;
        }
    }

    public void prepareTransaction(String[] sqlStmt, Vector[] pars) throws SQLException {
        int i = 0;
        try {
            for (i=0; i<stmt.length; i++) {
                stmt[i] = conns[i].prepareStatement(sqlStmt[i]);
                for (int p=0; p<pars[i].size(); p++)
                    stmt[i].setObject(p+1,pars[i].get(p));
            }
        }
        catch (SQLException e) {
            for (int t=i; t>=0; t--) {
                if (stmt[t]!=null)
                    stmt[t].close();
                if (conns[t]!=null)
                    conns[t].close();
            }
            throw new SQLException(e.getMessage() + "\n On statement n. " + (i+1),
                                   e.getSQLState(), e.getErrorCode()) ;
        }
    }

    public void executeTransaction() throws SQLException {
        int i = 0;
        try {
            for (i=0; i<conns.length; i++) {
                conns[i].setAutoCommit(false);
                if ( stmt[i].executeUpdate()==0 )
                    throw new SQLWarning("\n Record not found. ");
            }
            for (i=0; i<stmt.length; i++) {
                stmt[i].close();
                conns[i].commit();
                conns[i].setAutoCommit(true);
                conns[i].close();
            }

        }
        catch (SQLException e) {
            for (int t=i; t>=0; t--) {
                if (stmt[t]!=null)
                    stmt[t].close();
                if (conns[t]!=null) {
```

```
                    conns[t].rollback();
                    conns[t].close();
                }
            }
            throw new SQLException(e.getMessage() + "\n On execution n. " + (i+1),
                                   e.getSQLState(), e.getErrorCode());
        }
    }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

1 FEBBRAIO 2005

Si abbiano dei processi consumatore $C_i$ che vogliono determinare quando un certo evento *e* si verifica presso un processo produttore *P*. A tale scopo si adopera un paradigma di tipo *callback* ("richiamata"), in cui il consumatore che intende essere avvisato si registra preliminarmente presso il produttore, e si utilizza il supporto fornito da RMI, progettando la comunicazione tra i processi come un insieme di invocazioni remote di metodi di opportune classi.

In particolare, la classe `Producer` crea e registra un unico oggetto `e` di classe `Event`[6] sul quale la classe `Consumer` può invocare remotamente il metodo `e.subscribe(c)` per registrarsi, dopo aver acceduto al riferimento remoto `e` tramite l'RMIRegistry. `c` è il riferimento[7] (remoto) di un oggetto di tipo `CallbackEvent` (locale a ciascun consumatore) che il produttore inserirà in una lista, al fine di invocare successivamente il metodo di notifica `c.notifyEvent(String description)`.

Per semplicità:

*i)* Si suppongano tutti i processi residenti sul medesimo PC, *localhost*;
*ii)* Per istruzioni che coinvolgono eccezioni da gestire necessariamente, si adoperi semplicemente

```
try { … } catch (Exception e) { e.printStackTrace(); }
```

*iii)* Si implementino gli oggetti remoti ignorando la possibilità di processi concorrenti.
*iv)* Non si adoperi alcun *RMISecurityManager* né alcun caricamento dinamico delle classi.
*v)* Si crei l'RMIRegistry nel `Producer` con `LocateRegistry.createRegistry(1099);`

Tenendo conto della traccia fornita, e delle Figg.1-2, si completino:

- Le classi `Consumer` e `Producer`. Quest' ultima classe, oltre a quanto sopra specificato, genera una stringa diversa ogni secondo, la stampa e ne informa i consumatori tramite il metodo `happen`.
- Le classi `EventImpl` e `CallbackEventImpl`, in particolare il metodo:
- **`void`** `happen(String description)` della classe `EventImpl` invoca la `notifyEvent` su tutti gli oggetti `CallbackEvent` registrati nella lista
- **`void`** `subscribe(CallbackEvent aCallbackEvent)` della classe `EventImpl` inserisce il riferimento `aCallbackEvent` nella lista.
- **`void`** `notifyEvent(String description)` della classe `CallbackEventImpl`, che stampa `description`.

**TRACCIA PROPOSTA AL CANDIDATO:**

```
// Event.java
import java.rmi.*;

public interface Event extends Remote {
    public void happen(String description)              throws RemoteException;
    public void subscribe(CallbackEvent aCallbackEvent) throws RemoteException;
}

// CallbackEvent.java
import java.rmi.*;

public interface CallbackEvent extends Remote {
    public void notifyEvent(String description) throws RemoteException;
}

// Consumer.java
import java.rmi.*;
import java.rmi.server.*;
```

[6] Questa classe, definita di seguito, non ha nulla a che vedere con l'omonima `java.awt.Event`.
[7] Si noti che tale riferimento viene fornito al produttore come parametro, senza adoperare RMIRegistry.

```
public class Consumer {
    public static void main(String[] args) {
        //...
    }
}

// Producer.java
import java.rmi.*;
import java.rmi.registry.*;

public class Producer {
    public static void main(String[] args) {
         //...
    }
}

// CallbackEventImpl.java
import java.rmi.*;
import java.rmi.server.*;

public class CallbackEventImpl  extends UnicastRemoteObject implements CallbackEvent {
    public CallbackEventImpl() throws RemoteException {
        //...
    }

    public void notifyEvent(String description) throws RemoteException {
        //...
    }
}
// EventImpl.java
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
public class EventImpl extends UnicastRemoteObject implements Event {
    //...
    public EventImpl() throws RemoteException {
        //...
    }
    public void happen(String description) throws RemoteException {
        //...
    }
    public void subscribe(CallbackEvent aCallbackEvent) throws RemoteException {
        //...
    }
}
```

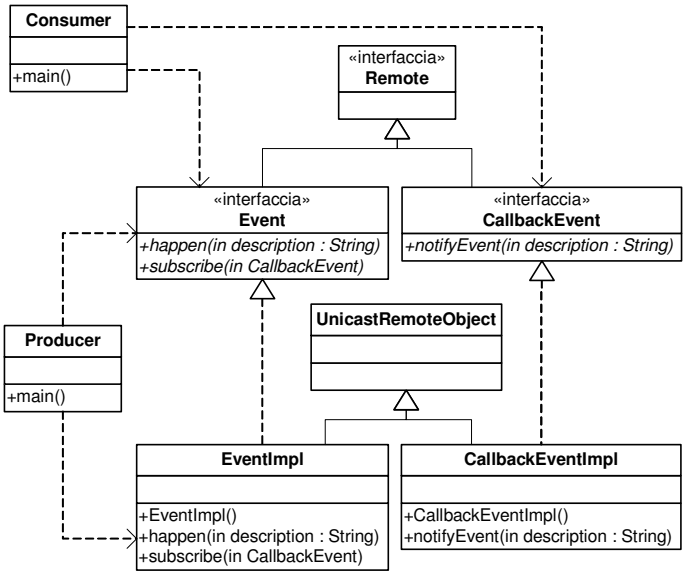



**Fig.1 Diag. delle Classi**

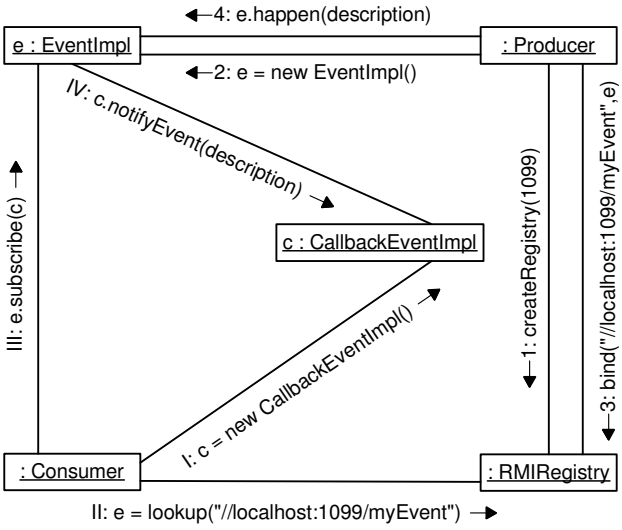



**Fig.2 Diag. Collaborazione degli oggetti**

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

1 FEBBRAIO 2005

**SOLUZIONE DELLE PARTI DA SVOLGERE:**

```
// Consumer.java
...
    public static void main(String[] args) {
       try {
            CallbackEvent c = new CallbackEventImpl();
           Event e = (Event)Naming.lookup("//localhost:1098/myEvent");
           e.subscribe(c);
       }
       catch (Exception e) {
           e.printStackTrace();
       }
       System.out.println("Ho chiesto a 'myEvent' di avere la notifica degli eventi");
          // Questo processo non termina finche` vi sono oggetti locali (remotamente) riferiti.
          // Ogni invocazione remota 'c.notifyEvent(desc)' produrra` una corrispondente
          // esecuzione in questo scope, quindi la stampa di 'desc' nella console di questo processo
    }
// CallbackEventImpl.java
...
    public CallbackEventImpl() throws RemoteException {
        super();
    }

    public void notifyEvent(String description) throws RemoteException {
        System.out.println(description);
    }

// EventImpl.java
...
public class EventImpl extends UnicastRemoteObject implements Event {

    private LinkedList list;

    public EventImpl() throws RemoteException {
        super();
        list = new LinkedList();
    }

    public void happen(String description) throws RemoteException {
        for (int i=0; i<list.size(); i++)
            ((CallbackEvent)list.get(i)).notifyEvent(description);
    }

    public void subscribe(CallbackEvent aCallbackEvent) {
        list.add(aCallbackEvent);
    }
}

// Producer.java
...
    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1098);
            Event e = new EventImpl();
            Naming.bind("//localhost:1098/myEvent",e);
            System.out.println("Oggetto 'myEvent' registrato ed in attera di invocazioni remote...");

            for (int i=0; ;i++) {
                System.out.println("Ho prodotto l'evento n. " + i);
                e.happen("Evento n." + i);
                Thread.sleep(1000);
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

## 18 FEBBRAIO 2005

Una unità aziendale di decision making opera, per le decisioni di tipo consueto e ripetitivo, secondo la seguente consolidata procedura. Un qualsiasi membro del gruppo formula una linea d'azione; un secondo membro ne prende visione approvandola in pieno – formulando quindi una proposta identica – oppure proponendo una variante; un terzo collega consulta le proposte già formulate e quindi formula la propria; e così via.

Quando esiste una proposta identicamente formulata per la "metà più uno" dei componenti, la decisione viene presa e comunicata a tutti i membri del gruppo. Altrimenti i partecipanti continuano iterativamente nel passaggio dell'elenco di proposte riformulando la propria e possibilmente tenendo conto della proposta più accreditata allo scopo di far giungere il gruppo ad una convergenza di opinioni.

Viene convocato un ingegnere informatico per la gestione d'azienda, allo scopo di progettare un modello per tale processo aziendale. Egli introduce i concetti di *lavagna mobile* e *minimale di fiducia* (Fig.1) ed inoltre sviluppa un prototipo in linguaggio Java per informatizzare la procedura. Siano N i componenti del gruppo, ciascuno dei quali identificato da un intero tra 0 ed N-1; la lavagna mobile (`MobileBlackboard`) rappresenta una tabella ad N righe, collocata in un posto "ben noto". Ciascun componente i-esimo può prelevare la lavagna, leggerla valutando se sia stato ottenuto il minimale di fiducia, altrimenti scrivere la propria proposta in corrispondenza della i-esima riga, ricollocando la lavagna nella posizione di partenza. Il minimale di fiducia è un particolare stato della lavagna in cui esiste una proposta identica per almeno N/2 +1 partecipanti, anche se vi sono righe ancora vuote.

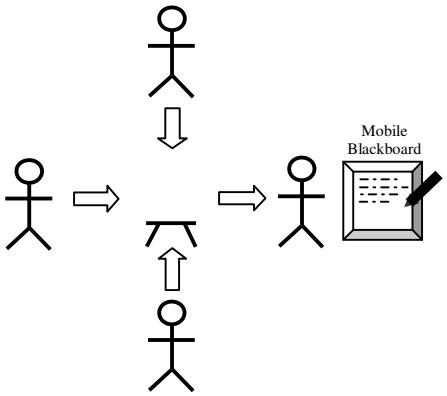



Fig. 1 – Mobile Blackboard, scenario con decision making unit di 4 elementi.

Su un oggetto `MobileBlackboard` possono compiersi le seguenti operazioni:

**`public`** `MobileBlackboard(`**`int`** `dim)`, il costruttore;

**`public void`** `print()`, stampa l'elenco di proposte contenute;

**`public void`** `put(String proposal,` **`int`** `clientID)`, inserisce una proposta alla posizione `clientID`;

**`public void`** `closing(`**`int`** `clientID)` un partecipante annota sulla lavagna di aver preso atto del conseguimento del minimale di fiducia, e quindi di considerare chiuso il processo decisionale.

**`public boolean`** `allClosed()` ritorna true se tutti i partecipanti considerano chiuso il processo decisionale.

**`public`** `String makeDecision()`, ritorna la proposta più accreditata se questa ha conseguito il *minimale di fiducia*; altrimenti ritorna `null`.

Dopodichè tutti i concetti legati alla comunicazione sono incapsulati nelle classi `TCPServer` e `TCPClient`, con due rispettivi metodi `main` da completare a cura del candidato. In particolare, il `TCPServer` istanzia un oggetto `MobileBlackboard` di dimensione data ed esegue il seguente ciclo sino a che non venga verificata la condizione `allClosed()`: (i) accetta richieste di connessione su una data porta (con una coda di backlog di dimensione 5), (i) invia la lavagna mobile, (ii) la riceve dal medesimo client, (iii) chiude la connessione. Al termine del ciclo stampa la decisione formulata. Il `TCPClient` esegue il seguente ciclo sino a che non venga ricevuta una lavagna mobile con conseguimento del minimale di fiducia: (i) chiede al `TCPServer` una connessione su una determinata porta, (ii) riceve la lavagna, (iii) se è stato conseguito il minimale di fiducia annota la chiusura del proprio processo decisionale, altrimenti stampa la lavagna ed inserisce la propria proposta (iv) rinvia la lavagna al server. Al termine del ciclo stampa la decisione formulata.

### TRACCIA PROPOSTA AL CANDIDATO:

```
// MobileBlackboard.java
import java.io.*;
public class MobileBlackboard implements Serializable {
    private String[]  proposals;
    private int[]     trust;
    private boolean[] closed;
    private int       totalClosed;
    private String    decision;

    public MobileBlackboard(int dim) {
        proposals = new String[dim];
        trust = new int[dim];
        closed = new boolean[dim];
        for (int i=0; i<proposals.length; i++) {
```

Scenario: il TCPServer e 3 TCPClient risiedono su localhost, in ascolto sulla porta 1000. Ecco alcune foto delle 4 finestre di shell dei comandi.

| Windows Shell |
| --- |
| **java TCPServer 3 1000** |

Il TCPServer parte per prima e rimane in ascolto.

```
            proposals[i] = null;
            closed[i] = false;
            trust[i] = 0;
        }
        decision = null;
        totalClosed = 0;
    }
    public void print() {
        System.out.println("---------------");
        for (int i=0; i<proposals.length; i++)
            if (proposals[i] != null)
                System.out.println(i + "> " + proposals[i]);
            else
                System.out.println(i + "> ");
        System.out.println("---------------");
    }
    public void put(String proposal, int clientID) {
        this.proposals[clientID] = proposal;
    }
    public void closing(int clientID) {
        if (!closed[clientID])
            totalClosed++;
        closed[clientID]=true;
    }
    public boolean allClosed() {
        return (totalClosed==proposals.length);
    }
    public String makeDecision() {
        if (decision!=null)
             return decision;
        for (int i=0; i<proposals.length; i++) {
            if (proposals[i]==null)
                continue;
            trust[i] = 1;
            for (int j=i+1; j<proposals.length; j++)
                if ( (proposals[j]!=null) &&
                     (proposals[j].equalsIgnoreCase(proposals[i])) )
                    trust[i]++;
        }
        for (int i=0; i<proposals.length; i++)
            if ( trust[i] > (proposals.length/2))
                return (decision = proposals[i]);
        return null;
   }
}
```

```
Windows Shell 0
java TCPClient 0 127.0.0.1 1000
---------------
0>
1>
2>
---------------
- Inserisci una proposta in posizione 0:
Ci vediamo alle 19
```

Il TCPClient 0 propone un incontro alle 19

```
Windows Shell 1
java TCPClient 1 127.0.0.1 1000
---------------
0> Ci vediamo alle 19
1>
2>
---------------
- Inserisci una proposta in posizione 1:
Ci vediamo alle 20
```

Il TCPClient 1, presa visione, lo propone alle 20

```
Windows Shell 2
java TCPClient 2 127.0.0.1 1000
---------------
0> Ci vediamo alle 19
1> Ci vediamo alle 20
2>
---------------
- Inserisci una proposta in posizione 2:
Ci vediamo alle 20
Decisione formulata: Ci vediamo alle 20
```

Il TCPClient 2, presa visione, lo propone alle 20. Quindi si raggiunge il minimale di fiducia.

```
Windows Shell 0
java TCPClient 0 127.0.0.1 1000
---------------
0>
1>
2>
---------------
- Inserisci una proposta in posizione 0:
Ci vediamo alle 19
Decisione formulata: Ci vediamo alle 20
```

```
Windows Shell 1
java TCPClient 1 127.0.0.1 1000
---------------
0> Ci vediamo alle 19
1>
2>
---------------
- Inserisci una proposta in posizione 1:
Ci vediamo alle 20
Decisione formulata: Ci vediamo alle 20
```

Tutti i TCP Client stampano il messaggio di decisione formulata e terminano.

```
Windows Shell
java TCPServer 3 1000
Decisione formulata: Ci vediamo alle 20
```

Da ultimo, il TCP Server chiude allo stesso modo.

```
// TCPServer.java
import java.io.*;
import java.net.*;

public class TCPServer {
    public static void main(String args[]) {

    final int   NUM_CLIENT  = Integer.parseInt(args[0]);
    final int   LISTEN_PORT = Integer.parseInt(args[1]);
    final int   BACKLOG     = 5;

    //...
}

// TCPClient.java
import java.io.*;
import java.net.*;

public class TCPClient {
    public static void main(String args[]) {

        final int       CLIENT_ID       = Integer.parseInt(args[0]);
        final String    SERVER_IP       = args[1];
        final int       SERVER_PORT     = Integer.parseInt(args[2]);
       //...
       BufferedReader in      = new BufferedReader(new InputStreamReader(System.in));
       //...
                    System.out.print("- Inserisci una proposta in posizione " + CLIENT_ID + ": ");
```

```
                    try {
                        proposal = in.readLine();
                    }
                    catch (IOException e) {
                        e.printStackTrace();
                    }
        //...
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

18 FEBBRAIO 2005

**SOLUZIONE DELLE PARTI DA SVOLGERE:**

```
// TCPServer.java
//...
    final int   BACKLOG     = 5;
    ServerSocket        servs    = null;
    Socket              sock     = null;
    ObjectInputStream   oin      = null;
    ObjectOutputStream  oout     = null;
    Integer             clientID = null;

    MobileBlackboard board = new MobileBlackboard(NUM_CLIENT);
    try {
        servs = new ServerSocket(LISTEN_PORT, BACKLOG);
        do {
            sock = servs.accept();
            oout = new ObjectOutputStream( sock.getOutputStream() );
            oin  = new ObjectInputStream( sock.getInputStream() );

            oout.writeObject(board);
            board = (MobileBlackboard)oin.readObject();

            oin.close();
            oout.close();
             sock.close();
        }
        while (!board.allClosed());
        servs.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    finally {
        if (!servs.isClosed())
        try {
            oin.close();
            oout.close();
            sock.close();
            servs.close();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
    System.out.println("Decisione formulata: " + board.makeDecision());
   }
}

// TCPClient.java
//...
        final int       SERVER_PORT     = Integer.parseInt(args[2]);

        Socket              sock     = null;
        ObjectInputStream   oin      = null;
        ObjectOutputStream  oout     = null;

        BufferedReader in      = new BufferedReader(new InputStreamReader(System.in));
        String proposal        = null;
        MobileBlackboard board = null;
        boolean  decisionMaked = false;

        try {
            do {
                sock = new Socket(SERVER_IP, SERVER_PORT);
                oin  = new ObjectInputStream( sock.getInputStream() );
                oout = new ObjectOutputStream( sock.getOutputStream() );

                board = (MobileBlackboard)oin.readObject();
                proposal = board.makeDecision();

                if (proposal!=null) {
                    decisionMaked = true;
                    board.closing(CLIENT_ID);
```

```
                }
                else {
                    board.print();
                    System.out.print("- Inserisci una proposta in posizione " + CLIENT_ID + ": ");
                    try {
                        proposal = in.readLine();
                    }
                    catch (IOException e) {
                        e.printStackTrace();
                    }
                    board.put(proposal, CLIENT_ID);
                }
                oout.writeObject(board);

                oout.close();
                oin.close();
                sock.close();
            } while(!decisionMaked);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            if (!sock.isClosed())
            try {
                oout.close();
                oin.close();
                sock.close();
            }
            catch (IOException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Decisione formulata: " + proposal);
    }
}
```

Esempio di script per il testing, che apre automaticamente 4 finestre di Shell in locale, a diversi colori di sfondo:

```
rem make.bat
set CLASSPATH=
javac *.java
start cmd /k "color 1F && java TCPServer 3 1000"
start cmd /k "color 2F && java TCPClient 0 127.0.0.1 1000"
start cmd /k "color 3F && java TCPClient 1 127.0.0.1 1000"
start cmd /k "color 4F && java TCPClient 2 127.0.0.1 1000"
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

## 10 GIUGNO 2005

Un *sequencer* è un oggetto di classe `Sequencer` che ha due metodi pubblici `sWait` e `sSignal`. Questi metodi permettono ad un insieme di thread di sincronizzarsi come segue. L'esecuzione del metodo `sWait` è sempre sospensiva. L'esecuzione del metodo `sSignal` permette ad un thread sospeso sul sequencer di riprendere l'esecuzione in accordo alla seguente politica:

1. se al momento dell'esecuzione di `sSignal` nessun thread risulta sospeso sul sequencer, l'esecuzione non ha alcun effetto ed il metodo ritorna `false`;
2. se, invece, al momento dell'esecuzione di `sSignal` due o più thread risultano sospesi sul sequencer, viene risvegliato quello che si è sospeso per primo ed il metodo ritorna `true`;

Il candidato progetti e realizzi la classe `Sequencer` seguendo la traccia fornita.

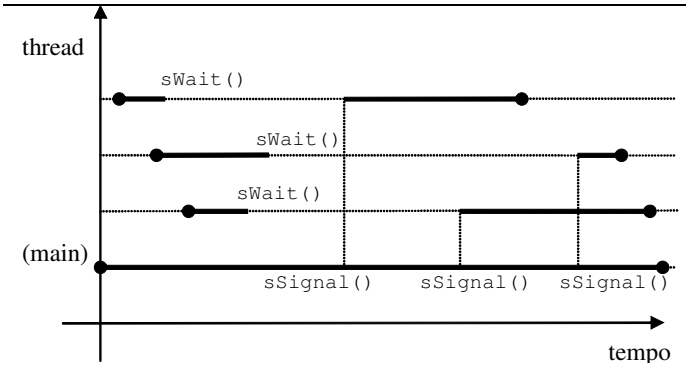



**Fig.1 –** Rappresentazione del funzionamento del Sequencer

```
>java Simulazione 5
 Thread n.0 sospeso...
 Thread n.2 sospeso...
 Thread n.1 sospeso...
 Thread n.0 risvegliato...
 Thread n.3 sospeso...
 Thread n.4 sospeso...
 Thread n.2 risvegliato...
 Thread n.1 risvegliato...
 Thread n.3 risvegliato...
 Thread n.4 risvegliato...
```

**Fig.2 –** Simulazione con 5 thread

### TRACCIA PROPOSTA AL CANDIDATO:

```
// RandLifeThread.java
public class RandLifeThread extends Thread {
    private Sequencer sequencer;
    public RandLifeThread(String name, Sequencer sequencer) {
        super(name);
        this.sequencer = sequencer;
    }
    public void run()   {
        Simulazione.attesaRandom();
        sequencer.sWait();
    }
}
// Simulazione.java
public class Simulazione {
    public static void attesaRandom() {
        try {
            Thread.sleep((long)(Math.random()*500));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        final int N = Integer.parseInt(args[0]);
        Sequencer sequencer = new Sequencer();
        for (int i=0; i<N; i++)
            new RandLifeThread(" Thread n." + i, sequencer).start();
        for (int i=0; i<N; i++) {
            attesaRandom();
            if ( !sequencer.sSignal() )
               i--;
        }
    }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

10 GIUGNO 2005

## SOLUZIONE DELLE PARTI DA SVOLGERE:

```
01: // Sequencer.java
02:
03: import java.util.*;
04:
05: public class Sequencer {
06:     private class PrivateSemaphore {
07:         public synchronized void pwait() throws InterruptedException {
08:             System.out.println( Thread.currentThread().getName() + " sospeso...");
09:             wait();
10:             System.out.println( Thread.currentThread().getName() + " risvegliato...");
11:         }
12:         public synchronized void psignal() {
13:             notify();
14:         }
15:     }
16:
17:     private LinkedList queue;
18:
19:     public Sequencer() {
20:         queue = new LinkedList();
21:     }
22:
23:     private synchronized PrivateSemaphore addLast() {
24:         queue.addLast( new PrivateSemaphore() );
25:         return (PrivateSemaphore)queue.getLast();
26:     }
27:
28:     private synchronized PrivateSemaphore removeFirst() {
29:         if ( !queue.isEmpty() )
30:             return (PrivateSemaphore)queue.removeFirst();
31:         return null;
32:     }
33:
34:     public void sWait() {
35:         try {
36:             addLast().pwait();
37:         }
38:         catch (InterruptedException e) {
39:             e.printStackTrace();
40:         }
41:     }
42:
43:     public boolean sSignal() {
44:         PrivateSemaphore pm = removeFirst();
45:         if ( pm == null)
46:            return false;
47:          pm.psignal();
48:          return true;
49:     }
50: }
```

## OSSERVAZIONI ED APPROFONDIMENTI:

*Ai fini dell'esame consideriamo una soluzione semplificata in cui vengono gestite le sincronizzazioni degli accessi a LinkedList (la cui implementazione Java* ***non*** *è thread-safe, vedere API) ed a PrivateSemaphore (il synchronized è imposto a run-time dai costrutti wait e notify, anche per un solo thread), ma non l'atomicita' dei metodi sWait ed sSignal.*
*Tale semplificazione non è in grado di impedire alcuni scenari di* ***corsa critica****. Ad esempio se la lista è vuota, e se una sSignal si infrappone in una sWait, tra addLast e pwait (riga 36), allora essa rimuove l'oggetto PrivateSemaphore prima che il thread corrispondente si sospenda, per cui questo non potra' piu' essere risvegliato. Se invece e' un'altra sWait ad infrapporsi, allora gli inserimenti in coda (e quindi il risveglio) dei due thread corrispondenti sono invertiti rispetto alle sospensioni: $addLast_1$ $addLast_2$ $pwait_2$ $pwait_1$. Anche due sSignal possono sovrapporsi, per cui le rimozioni dalla coda (rigo 44) sono invertite rispetto ai risvegli (rigo 47): $removeFirst_1$ $removeFirst_2$ $psignal_2$ $psignal_1$. Quest'ultimo caso di corsa critica si puo' risolvere modificando sSignal come segue:*

```
public synchronized boolean sSignal() {
    if ( queue.isEmpty() )
        return false;
    ((PrivateSemaphore)queue.removeFirst()).psignal();
    return true
}
```

*Sebbene vi siano invocazioni innestate di metodi sincronizzati, il nested monitor lockout non puo' avvenire in quanto i metodi sSignal e psignal non sono bloccanti, a differenza della sWait in cui la sospensione di pwait non permetterebbe il rilascio del monitor associato all'oggetto Sequencer. Per risolvere anche questo problema occorre fornire alla sSignal un modo per sapere se il thread relativo all'oggetto che si sta per rimuovere è giã sospeso, ed in caso contrario eseguire un* ***return false****. A tale scopo si può adoperare il metodo Thread.getState(), a patto di memorizzare anche la lista degli*

*oggetti Thread corrispondenti agli oggetti PrivateSemaphore. Ma allora, perché non sostituire la lista di oggetti PrivateSemaphore con la lista degli oggetti Thread, che sono comunque degli oggetti Java con un proprio monitor, e* ***sospendere ciascuna attività thread sull'oggetto Thread che lo rappresenta****? Con tale accorgimento il codice mantiene la compattezza originaria.*

```
// Sequencer.java - codice compatibile con la piattaforma Java 1.5
import java.util.*;

public class Sequencer {

    private LinkedList<Thread> queue;

    public Sequencer() {
        queue  = new LinkedList<Thread>();
    }

    public void sWait() {
        Thread t = Thread.currentThread();
        synchronized (this) {
            queue.addLast(t);
        }
        synchronized(t) {
            System.out.println(t.getName() + " sospeso...");
            try {
                t.wait();
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println( t.getName() + " risvegliato...");
        }
    }

    public synchronized boolean sSignal() {
        if ( queue.isEmpty() )
                return false;
        Thread t = (Thread)queue.getFirst();
        if (!((t.getState()).equals(Thread.State.WAITING)))
            return false;
        queue.removeFirst();
        synchronized(t) {
                t.notify();
        }
        return true;
    }
}
```

*Rimane un ultimo caso di corsa critica non risolto. La sovrapposizione di due sWait, in cui la seconda sWait si inserisce tra i due blocchi sincronizzati della prima (relativi agli oggetti this e t). Come visto questo provoca un fenomeno di ribaltamento dell'ordine dei thread. Per aumentare la probabilita` di questo evento inserire la seguente istruzione tra i due blocchi sincronizzati della sWait:*

```
  for (long i = 0; i<50000000; i++) ;
```

*e notare come l'ordine di risveglio dei thread (ossia l'ordine di inserimento in lista) risulta scorrelato dall'ordine di sospensione (stabilito dal tempo occorrente ad eseguire il for per ciascun thread). Per risolvere questa corsa critica occorre dotare il Sequencer di un semaforo a mutua esclusione (così come inteso a livello di sistema operativo) in modo da creare una sezione critica contenente i due blocchi sincronizzati. In tal modo una seconda sWait si sospende sul semaforo ed attende la conclusione della prima.*

```
// Sequencer.java - soluzione completa - codice compatibile con la piattaforma Java 1.5
import java.util.*;

public class Sequencer {
        private class Semaphore {
        private int counter;

        public Semaphore(int c) {
                counter = c;
        }

        public synchronized void _wait() {
                while (counter<=0)
                try {
                        wait();
                }
                catch (InterruptedException e) {
                        e.printStackTrace();
                }
                counter--;
        }
        public synchronized void _signal() {
            counter++;
            notify();
        }
    }
```

```
    private LinkedList<Thread> queue;
    private Semaphore mutex;

    public Sequencer() {
        queue  = new LinkedList<Thread>();
        mutex  = new Semaphore(1);
    }

    public void sWait() {
        mutex._wait();                                          // inizio  sez. critica
        Thread t = Thread.currentThread();
        synchronized (this) {
            queue.addLast(t);
        }
        synchronized(t) {
            System.out.println(t.getName() + " sospeso...");
            try {
                mutex._signal();                                // rilascio sez. critica
                t.wait();
                mutex._wait();                                  // ripresa  sez. critica
            }
            catch (InterruptedException e) {
               e.printStackTrace();
            }
            System.out.println( t.getName() + " risvegliato...");
        }
        mutex._signal();                                        // fine     sez. critica
    }

    public synchronized boolean sSignal() {
        if ( queue.isEmpty() )
               return false;
        Thread t = (Thread)queue.getFirst();
        if (!((t.getState()).equals(Thread.State.WAITING)))
            return false;
        queue.removeFirst();
        synchronized(t) {
               t.notify();
        }
        return true;
    }
}
```

*Si osservi che è sempre possibile che, all'interno della sWait, tra l'istruzione* mutex._signal() *e l'istruzione* t.wait *si inserisca un secondo thread con una sWait, che pertanto inserirebbe il nuovo oggetto in lista prima che il thread precedente si sia sospeso. Ma questa volta il secondo thread non potrebbe eseguire il secondo blocco sincronizzato, per cui l'ordine di esecuzione risulta corretto ($addLast_1$ $addLast_2$ $pwait_1$ $pwait_2$) grazie all'interlacciamento delle sezioni critiche (dell'oggetto t e del semaforo).*

*Si osservi come il costrutto semaforico risulti meno leggibile e maggiormente deadlock-prone rispetto al costrutto monitor: prima della t.wait il programmatore deve occuparsi di rilasciare la sezione critica e riacquisirla dopo. A questo punto si potrebbe eliminare la sincronizzazione dei due blocchi, usufruendo della mutua esclusione offerta dal semaforo? Per il secondo blocco questo non è possibile, in quanto le istruzioni t.wait e t.notify di Java richiedono l'acquisizione del monitor dell'oggetto t. La sincronizzazione sul primo blocco (this) è legata all'atomicita` nella gestione dell'oggetto LinkedList, per cui eliminandola si permetterebbe a sWait si accedere alla lista in parallelo ad sSignal. A meno di non togliere la sincronizzazione dal metodo sSignal e gestire la mutua esclusione con il medesimo semaforo. L'uso estensivo del costrutto semaforico renderebbe meno leggibile anche il codice della sSignal (es. prima di ogni return occorre rilasciare la sezione critica). In conclusione, si preferisce lasciare inalterata la sSignal e quindi anche il primo blocco di sicnronizzazione della sWait, adoperando per quanto possibile il monitor di Java invece del semaforo.*

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

## 1 LUGLIO 2005

Con riferimento all'architettura proposta di seguito, si realizzino le implementazioni dei servizi (classi "…`Impl.java`" evidenziate in grassetto in Fig.1) e le invocazioni del client di un sistema basato su RMI che permette l'interazione *asincrona tra i processi*, strutturato come di seguito.

Il processo client invoca un metodo remoto **`executeTask`** per far produrre un risultato ad un processo server, e si sospende in attesa di tale risultato, da prelevare localmente su un oggetto di classe **`ResultImpl`**. Il metodo **`executeTask`** passa (per copia) un parametro di ingresso **`TaskImpl`**, e (per riferimento remoto) l'oggetto **`ResultImpl`** sul quale il processo server inserirà il risultato tramite un metodo remoto. Tali oggetti sono riferiti mediante corrispondenti riferimenti interfaccia, rispettivamente locali o remoti.

In particolare, la classe **`ResultImpl`** (Fig.1) realizza il metodo locale **`waitForResult()`**, che nella fase iniziale provoca una sospensione in attesa del risultato **`Object`** e nella fase finale resituisce tale oggetto, ed il metodo remoto **`returnResult(Object o)`**, che serve a passare (per copia) l'oggetto **`o`** dal server al client.

La classe **`TaskImpl`** realizza il metodo locale **`execute()`** che prende un **`Object`** (generato dal client) lo converte in una stringa, ne converte i caratteri in maiuscolo, e ritorna tale stringa[8].

La classe **`ComputeImpl`** realizza il metodo remoto **`executeTask(Task t, Result r)`** di cui sopra, che provvede ad elaborare l'oggetto **`t`** e restituire il risultato nell'oggetto **`r`**.

La dinamica delle invocazioni si può riassumere come in Fig.2, in cui ciascun tratto tra due oggetti rappresenta un'azione (invocazione di metodo, creazione) di un oggetto sull'altro, nel verso della freccia.

Si osservi che il metodo **`returnResult(Object o')`**, oltre a passare l'oggetto **`o'`**, provvede a risvegliare il processo client in attesa del risultato. Questi, appena risvegliato, stampa l'oggetto **`o'`**.

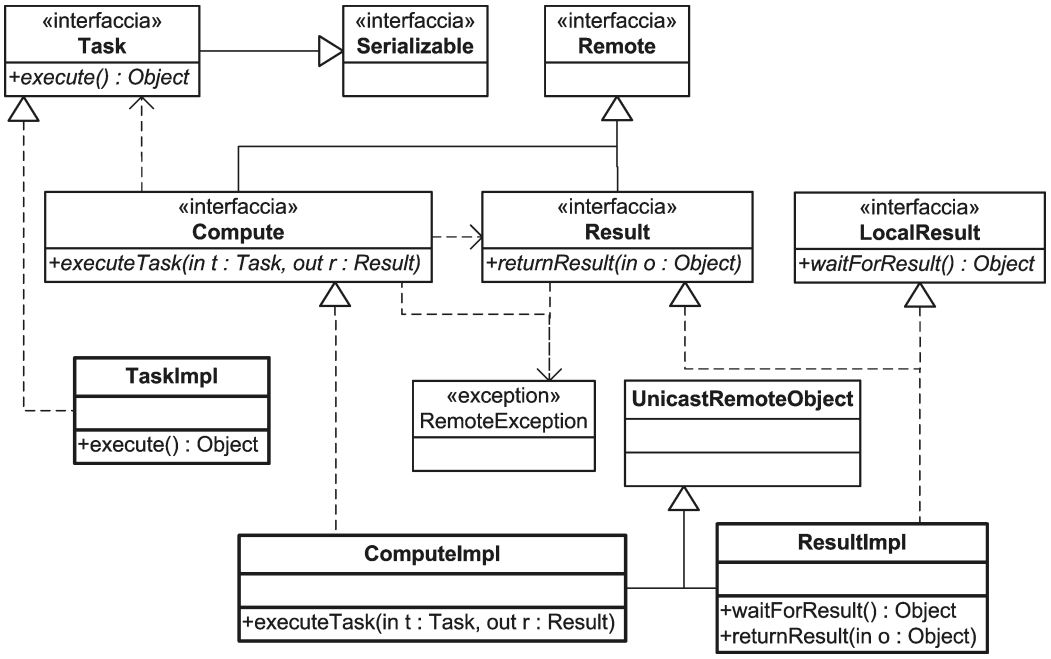



Fig.1 Diagramma delle Classi

La Fig.3 mostra la distribuzione dei componenti a run-time, automaticamente gestita da alcuni script della cartella *traccia*. In Fig.4 viene mostrata la distribuzione dei file sorgente (cartella *java*) e dei file di bytecode (cartella *class*). Lo script `make.bat` compila tutti i file sorgente e li distribuisce nelle cartelle `class/client` e `class/server`. I file `goc.lnk` e `gos.lnk` sono collegamenti (nel senso di Windows® ) agli script `class/client/goc.bat` e `class/server/gos.bat` che provvedono ad eseguire le applicazioni client e server rispettivamente.

[8] Nella simulazione proposta, il processo client genera una stringa "ciao" come oggetto Object, e quindi come risultato verrà restituita la medesima stringa "CIAO" in maiuscolo.

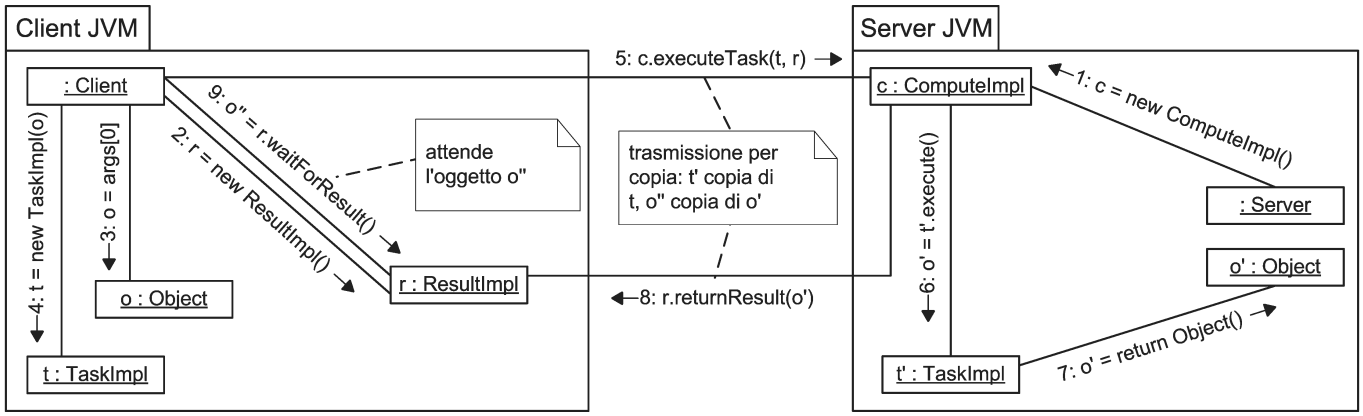



Fig.2 Diagramma di collaborazione degli oggetti

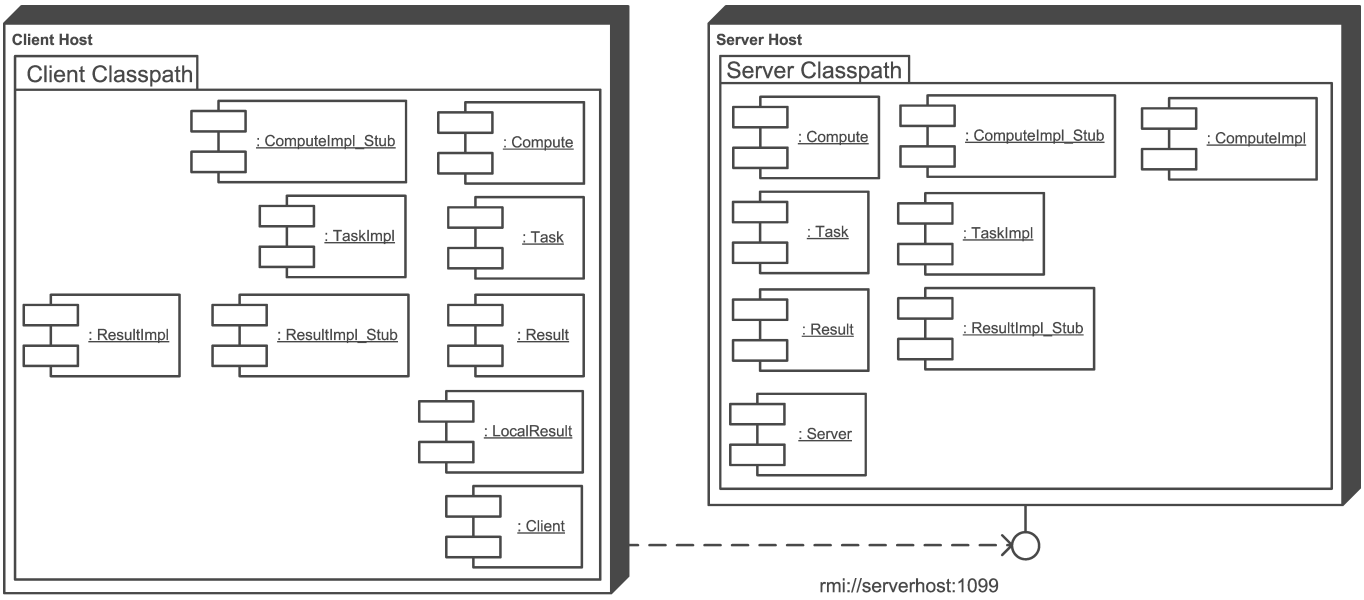



Fig.3 Diagramma di distribuzione dei componenti

Ai fini dello svolgimento del compito, è sufficiente:

1) editare i file: *java/both/TaskImpl.java*
   *java/client/ResultImpl.java*
   *java/client/Client.java*
   *java/server/ComputeImpl.java* ;
2) lanciare lo script *make.bat*, leggendo eventuali errori dalla consolle;
3) lanciare il server mediante il link *gos.lnk* (nessun messaggio viene stampato)
4) lanciare il client mediante il link *goc.lnk* (viene stampato "CIAO")
5) uccidere il processo server[9]

e così via, riprendendo dal punto 2) ad ogni modifica dei file sorgente.

[9] Potrebbe essere necessario uccidere anche il processo client, che tarda a terminare nell'attesa che vengano liberate le risorse occupate dall'oggetto remoto ResultImpl.

## TRACCIA PROPOSTA AL CANDIDATO:

```
rem make.bat ------------------------------------------------------
@echo off
set   CLASSPATH=
del   class\server\*.class /F /S /Q
del   class\client\*.class /F /S /Q
javac java\both\*.java -d class\server
copy  class\server\*.class class\client
javac -classpath class\server java\server\*.java -d class\server
javac -classpath class\client java\client\*.java -d class\client
rmic  -classpath .\class\server -v1.2 ComputeImpl -d .\class\server
copy  class\server\ComputeImpl_Stub.class class\client
rmic  -classpath .\class\client -v1.2 ResultImpl -d  .\class\client
copy  class\client\ResultImpl_Stub.class class\server
pause

rem gos.bat ------------------------------------------------------
@echo off
color 4F
set CLASSPATH=
java Server
pause

rem goc.bat ------------------------------------------------------
@echo off
color 2F
set CLASSPATH=
java Client ciao
pause
```

```
// Compute.java
import java.rmi.*;
public interface Compute extends Remote {
        void executeTask(Task t, Result r)    throws RemoteException;
}
```

```
// Result.java
import java.rmi.*;
public interface Result extends Remote {
        void returnResult(Object o) throws RemoteException;
}
```

```
// Task.java
import java.io.*;
public interface Task extends Serializable {
        Object execute();
}
```

```
// LocalResult.java
public interface LocalResult {
        Object waitForResult();
}
```

```
// Server.java
import java.rmi.*;
import java.rmi.registry.*;
public class Server {
        public static void main(String[] args) {
                try {
                        LocateRegistry.createRegistry(1099);
                        Naming.rebind("//localhost:1099/Compute", new ComputeImpl());
                }
                catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

```
// Client.java
import java.rmi.*;
public class Client {
        public static void main(String[] args) {
                try {
                        String name = "//localhost:1099/Compute";
                        Compute c = (Compute)Naming.lookup(name);
                        //...
                }
                catch (Exception e) {
                        e.printStackTrace();
                }
        }
```

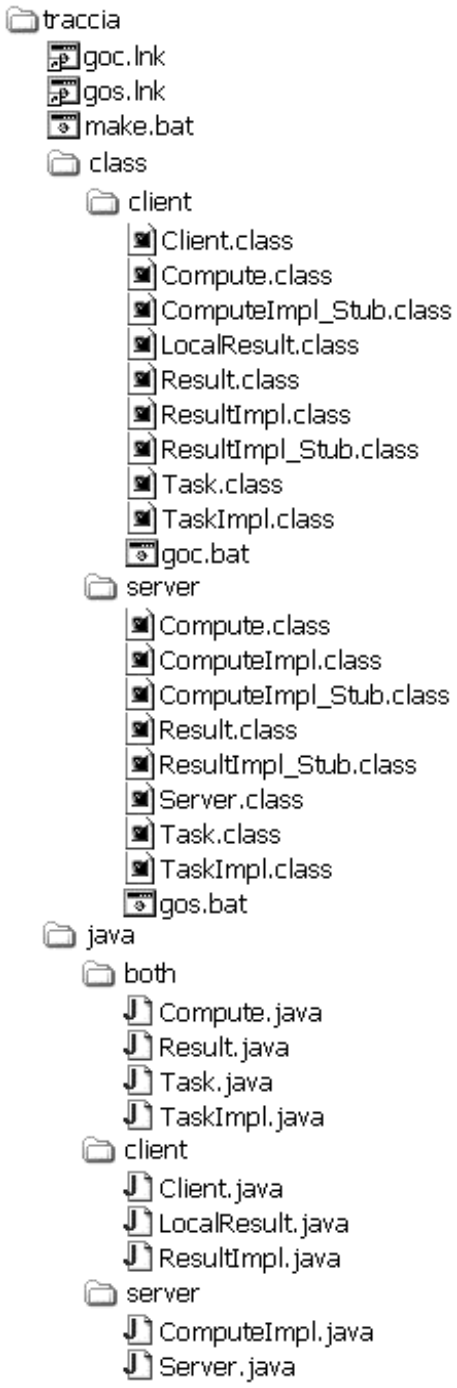



Fig.4 - distribuzione dei files

```
}

// ResultImpl.java
import java.rmi.*;
import java.rmi.server.*;

public class ResultImpl extends UnicastRemoteObject implements Result, LocalResult {
        //...

        public ResultImpl()    throws RemoteException {
                //...
        }

        public synchronized Object waitForResult() {
                //...
        }

        public synchronized void returnResult(Object o) throws RemoteException {
                //...
        }
}

// ComputeImpl.java
import java.rmi.*;
import java.rmi.server.*;

public class ComputeImpl extends UnicastRemoteObject implements Compute {

        public ComputeImpl() throws RemoteException {
                //...
        }

        public void executeTask(Task t, Result r) throws RemoteException {
                //...
        }
}

// TaskImpl.java
import java.util.*;

public class TaskImpl implements Task {
        //...

        public TaskImpl(Object o) {
                //...
        }

        public Object execute() {
                //...
        }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

1 LUGLIO 2005

**SOLUZIONE DELLE PARTI DA SVOLGERE:**

```
// Client.java
import java.rmi.*;
public class Client {
	public static void main(String[] args) {
		try {
			String name = "//localhost:1099/Compute";
			Compute c = (Compute)Naming.lookup(name);
			ResultImpl r = new ResultImpl();
			c.executeTask(new TaskImpl(args[0]), r);
			System.out.println(r.waitForResult());
		}
		catch (Exception e) {
			e.printStackTrace();
		}
	}
}
// ResultImpl.java
import java.rmi.*;
import java.rmi.server.*;

public class ResultImpl extends UnicastRemoteObject implements Result, LocalResult {
	private Object o;
	public ResultImpl()    throws RemoteException {
		super();
		o = null;
	}
	public synchronized Object waitForResult() {
		while (o==null) {
			try {
				wait();
			}
			catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		Object res = o;
		o = null;
		return res;
	}
	public synchronized void returnResult(Object o) throws RemoteException {
		this.o = o;
		notify();
	}
}
// ComputeImpl.java
import java.rmi.*;
import java.rmi.server.*;
public class ComputeImpl extends UnicastRemoteObject implements Compute {
	public ComputeImpl() throws RemoteException {
		super();
	}
	public void executeTask(Task t, Result r) throws RemoteException {
		r.returnResult(t.execute());
	}
}
// TaskImpl.java
import java.util.*;

public class TaskImpl implements Task {
	private Object o;
	public TaskImpl(Object o) {
		this.o = o;
	}
	public Object execute() {
		return ( o.toString().toUpperCase() );
	}
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

## 22 LUGLIO 2005

Si consideri un servizio di *storage server*, che permette di memorizzare in modo persistente degli oggetti Java qualsiasi. I servizi disponibili per client e server sono definiti dalle seguenti interfacce[10]:

```
public interface Storage extends Remote {
    long    save(Object o)          throws RemoteException, IOException;
    Object  load(long id)           throws RemoteException, IOException;
}

public interface LocalStorage {
    void    open(String archiveName) throws FileNotFoundException;
    void    close()                  throws IOException;
}
```

- Il metodo `save` passa (per valore[11]) un `Object` e lo inserisce in archivio, restituendo un intero che identifica univocamente tale oggetto all'interno dell'archivio.
- Il metodo `load` restituisce l'oggetto identificato dall'intero `id`.
- Il metodo `open` apre l'archivio di nome `archiveName`, creando un archivio nuovo con tale nome se non già esistente, consentendo ai client il salvataggio ed i recupero di oggetti.
- Il metodo `close` chiude l'archivio precedentemente aperto, impedendo le suddette operazioni remote.

L'archivio viene implementato direttamente su filesystem e costituito da un unico file binario di nome "`archive.dat`". A tale scopo, si adoperi la classe `java.io.RandomAccessFile` che consente di gestire i dati su file come in un lungo array di byte (Fig.1). É possibile accedere direttamente all'i-esimo byte mediante una variabile intera, il *file pointer*.

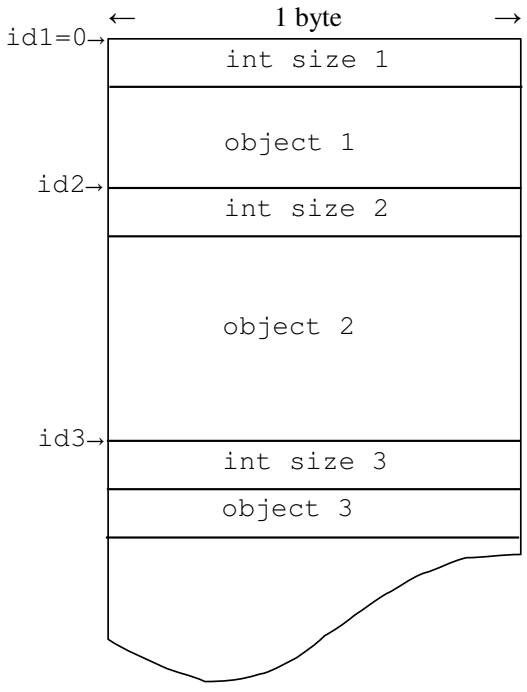


Fig.1 Implementazione dell'archivio su un file ad accesso casuale

Nell'implementare il metodo `save`, ogni nuovo inserimento viene effettuato in fondo al file. In particolare, l'oggetto viene prima convertito in un'array di byte, quindi si inserisce nel file un intero (le dimensioni di tale array) e l'array medesimo. L'identificatore restituito dal metodo è la posizione del file pointer prima dell'inserimento. Dualmente, il metodo `load` provvede a collocare il file pointer nella posizione indicata dall'identificatore, prelevare l'intero contenente le dimensioni dell'array di byte, caricare tale array e convertirlo nell'oggetto originario.

I servizi di serializzazione e conversione in array di byte sono svolti dalla classe `ByteConverter`, e così composti:

- Il metodo `object2byte` prende come parametro di ingresso un `Object`, lo serializza mediante il metodo `writeObject()` di `ObjectOutputStream`, che a sua volta avvolge un `ByteArrayOutputStream` da cui i dati possono essere letti come array di byte mediante il metodo `toByteArray()`.
- Il metodo `byte2object` prende come parametro di ingresso un array di byte, costruisce un `ByteArrayInputStream` con tale array, lo avvolge mediante un `ObjectInputStream` da cui si può deserializzare l'oggetto mediante `readObject()`.

Il candidato realizzi le classi `ByteConverter` e `StorageImpl`, gestendo (per quest'ultima) la possibilità di accessi multipli all'archivio mediante dei blocchi `synchronized` di estensione minima, ed adoperando il codice proposto di seguito.

[10] Per semplicità, non è prevista la possibilità di rimozione di oggetti.
[11] In accordo al meccanismo di passaggio dei parametri in RMI, l'oggetto Java passato a run-time dovrà essere serializzabile.

In particolare, si considerino le applicazioni di testing[12] `Server` e `Client`. Il server apre un archivio, lo registra presso il registry, e lo chiude dopo dieci secondi[13]. Il client, dopo aver ottenuto il riferimento remoto dell'archivio, genera 4 diversi tipi di oggetto ciascuno dei quali viene aggiunto in archivio, caricato e quindi stampato.

La compilazione, la distribuzione e l'esecuzione dei componenti viene automaticamente gestita da alcuni script della cartella *traccia*, definiti in Fig.2. La Fig.3 mostra la distribuzione dei file sorgente (cartella *java*) e dei file di bytecode (cartella *class*). Lo script `make.bat` compila tutti i file sorgente e li distribuisce nelle cartelle `class/client` e `class/server`. I file `goc.lnk` e `gos.lnk` sono collegamenti (nel senso di Windows® ) agli script `class/client/goc.bat` e `class/server/gos.bat` che provvedono ad eseguire le applicazioni client e server rispettivamente.

```
rem make.bat ------------------------------------------------------
@echo off
set   CLASSPATH=
del   class\server\*.class /F /S /Q
del   class\client\*.class /F /S /Q
javac java\both\*.java -d class\server
copy  class\server\*.class class\client
javac -classpath class\server java\server\*.java -d class\server
javac -classpath class\client java\client\*.java -d class\client
rmic  -classpath .\class\server -v1.2 StorageImpl -d .\class\server
copy  class\server\StorageImpl_Stub.class class\client
pause

rem gos.bat ------------------------------------------------------
@echo off
color 4F
set CLASSPATH=
java Server
pause

rem goc.bat ------------------------------------------------------
@echo off
color 2F
set CLASSPATH=
java Client
pause
```

Fig.2 – script di compilazione e distribuzione


Fig.3 - distribuzione dei files

**goc.lnk**

```
0) java.lang.StringBuffer: Universita' di Pisa
176) java.math.BigDecimal: 3.1415926535897932384
478) java.lang.Exception: java.lang.Exception: Confermo la regola
896) java.util.EventObject: java.util.EventObject[source=null]
Premere un tasto per continuare . . .
```

Fig.4 – esempio di esecuzione del client

Ai fini dello svolgimento del compito, il candidato deve limitarsi a:

6) editare i file *java/server/ByteConverter.java* e *java/server/StorageImpl.java* nei punti indicati da "`//...`"
7) lanciare lo script *make.bat*, leggendo eventuali errori dalla consolle;
8) lanciare il server mediante il link *gos.lnk* (nessun messaggio viene stampato)
9) lanciare il client mediante il link *goc.lnk* (La Fig.4 rappresenta un esempio di output)
10) uccidere il processo server[14]

[12] Il testing effettuato da tali classi realizza uno scenario standard di invocazione dei servizi. Non verifica ad esempio il corretto funzionamento in caso di accessi multipli. Il candidato dovrà realizzare comunque tutti i requisiti richiesti a prescindere dalle applicazioni client e server, basandosi sulla propria esperienza e sul materiale di consultazione per gli aspetti non verificati.

[13] Provare almeno una volta ad attendere l'invocazione della close(), per verificarne il corretto funzionamento.

[14] Dopo dieci secondi il server chiude l'archivio, per cui successive invocazioni del client non potranno essere compiute.

e così via, riprendendo dal punto 1).

**TRACCIA PROPOSTA AL CANDIDATO:**

```
// Storage.java
import java.rmi.*;
import java.io.*;
public interface Storage extends Remote {
    long    save(Object o)      throws RemoteException, IOException;
    Object  load(long id)       throws RemoteException, IOException;
}
// LocalStorage.java
import java.io.*;
public interface LocalStorage {
    void    open(String archiveName) throws FileNotFoundException;
    void    close()                  throws IOException;
}

// Server.java
import java.rmi.*;
import java.rmi.registry.*;
public class Server {
    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1099);
            StorageImpl storage = new StorageImpl();
            storage.open("archive.dat");
            Naming.rebind("//localhost:1099/Storage",storage);
            Thread.sleep(10000);
            storage.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
// Client.java
import java.rmi.*;
public class Client {
    public static void main(String[] args) {
        try {
            String name = "//localhost:1099/Storage";
            Storage s = (Storage)Naming.lookup(name);

            Object obj1 = null, obj2 = null;
            long   id;

            for (int i = 0; i<4; i++) {
                switch(i) {
                    case 0: obj1 = new java.lang.StringBuffer("Universita' di Pisa"); break;
                    case 1: obj1 = new java.math.BigDecimal("3.14159265358979323384"); break;
                    case 2: obj1 = new java.lang.Exception("Confermo la regola");     break;
                    case 3: obj1 = new java.util.EventObject(new Object());
                }
                id   = s.save(obj1);
                obj2 = s.load(id);
                System.out.println(id + ") " + obj2.getClass().getName() + ": " + obj2.toString());
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
// ByteConverter.java
import java.io.*;

public class ByteConverter {
    public static byte[] object2byte(Object obj) {
        //...
        try {
           //...
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        //...
    }

    public static Object byte2object(byte[] byteArr) {
        //...
        try {
           //...
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        //...
    }
}

// StorageImpl.java

import java.rmi.*;
import java.rmi.server.*;
import java.io.*;

public class StorageImpl extends UnicastRemoteObject
                         implements Storage, LocalStorage {

    //...

    public StorageImpl() throws RemoteException {
        //...
    }

    public void open(String filename) throws FileNotFoundException {
        //...
    }

    public long save(Object obj) throws RemoteException, IOException {
        //...
    }

    public Object load(long position) throws RemoteException, IOException {
        //...
    }

    public void close() throws IOException {
        //...
    }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

22 LUGLIO 2005

## SOLUZIONE DELLE PARTI DA SVOLGERE:

```
// ByteConverter.java
import java.io.*;

public class ByteConverter {
    public static byte[] object2byte(Object obj) {
        ByteArrayOutputStream bos = null;
        ObjectOutputStream    oos = null;
        byte[]                res = null;
        try {
            bos = new ByteArrayOutputStream();
              oos = new ObjectOutputStream(bos);
                oos.writeObject(obj);
                res = bos.toByteArray();
              oos.close();
            bos.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return res;
    }
    public static Object byte2object(byte[] byteArr) {
        ByteArrayInputStream bis = null;
        ObjectInputStream    ois = null;
        Object               res = null;
        try {
            bis = new ByteArrayInputStream(byteArr);
              ois = new ObjectInputStream(bis);
                res = ois.readObject();
              ois.close();
            bis.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return res;
    }
}

// StorageImpl.java
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;

public class StorageImpl extends UnicastRemoteObject implements Storage, LocalStorage {
    private RandomAccessFile archive = null;

    public StorageImpl() throws RemoteException {
        super();
    }
    public void open(String filename) throws FileNotFoundException {
        synchronized(this) {
          if (archive == null)
            archive = new RandomAccessFile(filename,"rwd");
        }
    }
    public long save(Object obj) throws RemoteException, IOException {
        byte [] bstream = ByteConverter.object2byte(obj);
        long res;
        synchronized(this) {
            archive.seek(archive.length());
            res = archive.getFilePointer();
            archive.writeInt(bstream.length);
            archive.write(bstream);
        }
        return res;
    }
```

```
    public Object load(long position) throws RemoteException, IOException {
        byte[] bstream;
        synchronized(this) {
            archive.seek(position);
            bstream = new byte[archive.readInt()];
            archive.read(bstream);
        }
        return ByteConverter.byte2object(bstream);
    }

    public void close() throws IOException {
        synchronized(this){
          if (archive != null) {
            archive.close();
            archive = null;
          }
        }
    }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

19 SETTEMBRE 2005

Si consideri il seguente *problema dei filosofi ciechi* (Fig.1). Cinque[15] filosofi ciechi sono seduti attorno ad una tavola rotonda. Ogni filosofo passa il suo tempo a compiere ciclicamente le seguenti due azioni: mangia riso finché non è sazio, quindi riflette fino a quando per lo sforzo profuso gli viene di nuovo fame, dunque riprende a mangiare e così via. Per mangiare ha bisogno di due bastoncini. Sul tavolo esistono cinque bastoncini che si alternano ai filosofi, in modo che ogni filosofo ne abbia uno a destra ed uno a sinistra per mangiare, e quindi ogni bastoncino sia una risorsa condivisa da due filosofi.

Come conseguenza di ciò, mai più di due filosofi mangeranno contemporaneamente.

Ogni filosofo, una volta preso un bastoncino (nel caso peggiore dopo la durata massima di un pasto del suo compagno adiacente) lo trattiene (anche se ancora non può mangiare) ed attende l'altro bastoncino. Se il filosofo attendesse la disponibilità di entrambi i bastoncini per impossessarsene, potrebbe attendere un tempo indefinito (starvation) nel caso in cui i suoi due filosofi adiacenti si alternassero nei pasti.

Inoltre, i filosofi scelgono a caso se prelevare prima il bastoncino destro o il sinistro. Ciò non esclude due casi particolari di corsa critica in cui tutti i filosofi prendono il bastoncino alla propria destra, o tutti alla propria sinistra, e rimangono in attesa dell'altro bastoncino, morendo tutti di fame (deadlock).

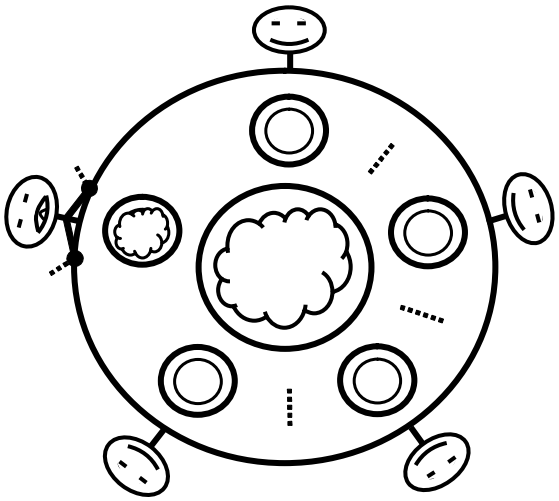


Fig.1 – Il problema dei cinque filosofi ciechi

Per risolvere questo problema si adotta una soluzione di *deadlock detection and recovery*, consistente nel lanciare un timer che periodicamente controlla che non ci sia stallo, ed in caso contrario prende dei provvedimenti[16].

Il candidato consideri la seguente traccia per l'implementazione di tale problema.

La classe `Stick` realizza il concetto di bastoncino ed offre i seguenti metodi:

- Il costruttore, che riceve come parametro la stringa `name`, il nome del bastoncino. Tale nome serve semplicemente a scopo di visualizzazione e non di controllo.
- Il metodo `grab()` che provoca la sospensione del thread `Philosopher` se il bastoncino non è disponibile.
- Il metodo `put()` che rende il bastocino disponibile per gli altri thread `Philosopher`.
- Il metodo statico `getStatus()` che restituisce lo *stato di mobilità* dell'insieme dei bastoncini, rappresentato da una variabile intera statica di tipo `long`, che viene inizializzata a zero ed incrementata nei metodi `grab()` e `put()`.

Il valore dello stato di mobilità indica il numero di volte che i bastoncini sono stati presi e rimessi a posto. Se lo stato di mobilità non cambia entro un tempo maggiore del massimo tra la massima durata di un pasto e la massima durata di una riflessione, significa che nessun bastoncino è stato rilasciato o preso, e quindi c'è un deadlock.

Infatti la classe `DeadlockDetector`, completamente fornita al candidato, è un timer che ad intervalli regolari confronta due valori successivi dello stato di mobilità.

La classe `Table` realizza una tavola, contenente un array di bastoncini ed uno di filosofi, ed i seguenti metodi:

- Il costruttore inizializza il riferimento all'array `philosophers` e costruisce l'array `sticks`.
- Il metodo `grabStick()` preleva il bastoncino a sinistra (`pos=0`) o a destra (`pos=1`) del filosofo `diner`.
- Il metodo `putStick()` posa il bastoncino a sinistra (`pos=0`) o a destra (`pos=1`) del filosofo `diner` .

[15] Supponiamo cinque per fissare le idee. In generale il problema vale per $N \geq 2$ filosofi.

[16] Un'altra metodologia, in generale non sempre facile da progettare unita ad un uso ottimale delle risorse, è quella di *deadlock prevention*. Nella fattispecie, si potrebbe fissare l'ordine di prelievo dei bastoncini (es. prima a destra) per tutti i filosofi tranne uno che seguirà l'ordine inverso (es. prima a sinistra). Oppure rendere la tavola in grado di rilevare lo stato immediatamente precedente al deadlock ed impedirne il raggiungimento condizionando la scelta dell'ultimo filosofo.

- Il metodo statico `getStatus()` restituisce lo stato di mobilità della classe dei bastoncini.
- Il metodo `removeDeadlock()` integralmente fornito, stampa un messaggio e termina l'applicazione.

La classe `Philosopher` realizza il thread filosofo, dai seguenti metodi:

- Il costruttore, che riceve come parametro la stringa `name` (il nome del filosofo), il riferimento alla tavola ed il numero di posto in cui il filosofo è seduto. Il nome è adoperato a scopo di visualizzazione e non di controllo.
- Il metodo `getPlace()` restituisce il numero di posto occupato dal filosofo.
- Il metodo `dine()` preleva i due bastoncini adiacenti, in ordine casuale, ed attende un intervallo random.
- Il metodo `think()` posa i due bastoncini, in ordine casuale, ed attende un intervallo random.
- Il metodo `run()` esegue due iterazioni della sequenza *pranza-pensa*.

Il candidato completi le classi `Stick`, `Table` e `Philosopher` adoperando il codice proposto di seguito.
In particolare, si consideri la classe di testing[17] `Simulazione` che costruisce un oggetto tavola, costruisce un array di thread filosofi e li avvia in sequenza, e schedula infine un thread periodico di deadlock detection.

La compilazione e la esecuzione dei componenti viene automaticamente gestita dallo script *make.bat*, definito in Fig.2. Le Fig.3-4 mostrano due esempi di esecuzione di cui il secondo con deadlock.

```
rem make.bat --------------
@echo off
set CLASSPATH=
javac *.java
java Simulazione 3
pause
```

Fig.2 – script di compilazione ed esecuzione

```
cmd.exe
f0 prende lo stick s1
f1 prende lo stick s2
f0 prende lo stick s0
f0 posa lo stick s0
f0 posa lo stick s1
f1 prende lo stick s1
f1 posa lo stick s2
f1 posa lo stick s1
f2 prende lo stick s2
f2 prende lo stick s0
f2 posa lo stick s0
f2 posa lo stick s2
f0 prende lo stick s0
f0 prende lo stick s1
f0 posa lo stick s0
f0 posa lo stick s1
f2 prende lo stick s0
f1 prende lo stick s1
f2 prende lo stick s2
f2 posa lo stick s2
f2 posa lo stick s0
f1 prende lo stick s2
f1 posa lo stick s1
f1 posa lo stick s2
Press any key to continue . . .
```

Fig.3 – esempio di esecuzione

```
cmd.exe
f0 prende lo stick s0
f1 prende lo stick s2
f0 prende lo stick s1
f0 posa lo stick s0
f0 posa lo stick s1
f2 prende lo stick s0
f0 prende lo stick s1
DEADLOCK!
Press any key to continue . . .
```

Fig.4 – esempio di esecuzione con deadlock

Ai fini dello svolgimento del compito, il candidato deve limitarsi a:

1) editare i file *traccia/Stick.java*, *traccia/Table.java* e *traccia/Philosopher.java* nei punti indicati da "`//...`"
2) lanciare lo script *make.bat*, leggendo eventuali errori di compilazione dalla consolle;
3) controllare i risultati e così via, riprendendo dal punto 1) sino a specifiche risolte.

## TRACCIA PROPOSTA AL CANDIDATO:

```
// Stick.java
public class Stick {
    //...
    public Stick(String name) {
        //...
    }
```

[17] Il testing effettuato da tale classe realizza uno scenario standard di esecuzione, ed il suo corretto funzionamento non garantisce la correttezza del programma in tutte le possibili condizioni. Il candidato dovrà realizzare comunque tutti i requisiti richiesti a prescindere da tale simulazione, basandosi sulla propria esperienza e sul materiale di consultazione per gestire le possibili corse critiche.

```
    public synchronized void grab() {
        //...
        System.out.println(Thread.currentThread().getName() + " prende lo stick " + name);
        //...
    }
    public synchronized void put() {
        //...
        System.out.println(Thread.currentThread().getName() + " posa lo stick " + name);
        //...
    }
    public static synchronized long getStatus() {
        //...
    }
}

// Table.java
public class Table {
    private Stick[]         sticks;
    private Philosopher[]   philosophers;

    public Table( Philosopher[] philosophers ) {
       //...
    }
    public void grabStick(Philosopher diner, int pos) {
       //...
    }
    public void putStick(Philosopher diner,  int pos) {
       //...
    }
    public static long getStatus() {
      //...
    }
    public void removeDeadlock() {
        System.err.println("DEADLOCK!");
        System.exit(1);
    }
}

// Philosopher.java
public class Philosopher extends Thread {
    //...
    public Philosopher(String name, Table table, int place) {
        //...
    }
    public int getPlace() {
        //...
    }
    private void dine() {
        //...
        Simulazione.attesaRandom();
    }
    private void think() {
        //...
        Simulazione.attesaRandom();
    }
    public void run() {
       //...
    }
}


// DeadlockDetector.java
import java.util.TimerTask;
public class DeadlockDetector extends TimerTask {

    private Table   table;
    private long[]  status  = {0, 0};
    private int     toggle;

    public DeadlockDetector(Table table) {
        super();
        this.table = table;
    }
```

```
    public boolean isDeadlock() {
        toggle = (toggle + 1) % 2;
        status[toggle] = table.getStatus();
        return (status[0] == status[1]);
    }
    public void run() {
        if ( isDeadlock() )
            table.removeDeadlock();
    }
}

// Simulazione.java
import java.util.Timer;

public class Simulazione {
    public static void attesaRandom() {
        try {
            Thread.sleep(100 + (long)(Math.random()*900));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        final int N = Integer.parseInt(args[0]);

        Philosopher[] philosophers = new Philosopher[N];
        Table table = new  Table(philosophers);

        for (int i=0; i<N; i++)
            philosophers[i] = new Philosopher("f" + i, table, i);

        for (int i=0; i<N; i++)
            philosophers[i].start();

        Timer timer = new Timer(true);
        timer.schedule(new DeadlockDetector(table),2000,2000);
   }
}
```

# TECNOLOGIE INFORMATICHE PER LA GESTIONE AZIENDALE

19 SETTEMBRE 2005

**SOLUZIONE DELLE PARTI DA SVOLGERE:**

```
// Stick.java
public class Stick {
    private boolean         available    = true;
    private String          name;
    private static long     status = 0;
    public Stick(String name) {
        this.name = name;
    }
    public synchronized void grab() {
        while (!available) {
            try {
                wait();
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        status++;
        System.out.println(Thread.currentThread().getName() + " prende lo stick " + name);
        available = false;
    }
    public synchronized void put() {
        status++;
        available = true;
        System.out.println(Thread.currentThread().getName() + " posa lo stick " + name);
        notify();
    }
    public static synchronized long getStatus() {
        return status;
    }
}

// Philosopher.java
public class Philosopher extends Thread {
    private Table table;
    private int   place;

    public Philosopher(String name, Table table, int place) {
        super(name);
        this.table  = table;
        this.place  = place;
    }
    public int getPlace() {
        return place;
    }
    private void dine() {
        int toggle =  (int)Math.round(Math.random());
        table.grabStick(this, toggle);
        table.grabStick(this, ++toggle % 2);
        Simulazione.attesaRandom();
    }
    private void think() {
        int toggle =  (int)Math.round(Math.random());
        table.putStick(this, toggle);
        table.putStick(this, ++toggle % 2);
        Simulazione.attesaRandom();
    }
    public void run() {
        for (int i=0; i<2; i++) {
            dine();
            think();
        }
    }
}

// Table.java
public class Table {
    private Stick[]         sticks;
    private Philosopher[]   philosophers;

    public Table( Philosopher[] philosophers ) {
        this.philosophers = philosophers;
```

```
        this.sticks = new Stick[philosophers.length];
        for (int i = 0; i < sticks.length; i++)
            sticks[i]   = new Stick("s" + i);
    }
    public void grabStick(Philosopher diner, int pos) {
        pos = (diner.getPlace() + pos) % sticks.length;
        sticks[pos].grab();
    }
    public void putStick(Philosopher diner,  int pos) {
        pos = (diner.getPlace() + pos) % sticks.length;
        sticks[pos].put();
    }
    public static long getStatus() {
        return Stick.getStatus();
    }
    public void removeDeadlock() {
        System.err.println("DEADLOCK!");
        System.exit(1);
    }
}
```