IS 2002-03 mod. B
Corso di Ingegneria del Software
UML

Analisi e progettazione concorrente

(esempi con Java)

Bibliografia e riferimenti

- Hassan Gomaa: "Designing concurrent distributed and real time applications with UML",
Addison Wesley 2000

- Reference manual UML 1.4

- JDK 1.4 DOC (Sun Microsystems)

- http:// www.real-time.org

Analisi e progettazione concorrente

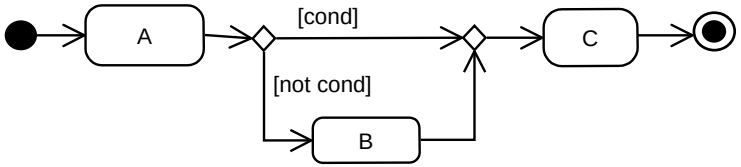
elementi di programmazione concorrente

Task e thread

- **heavyweight process:** (task)
  - scambio di contesto oneroso
  - possiede e controlla il proprio spazio di memoria
- **lightweight process:** (thread)
  - scambio di contesto essenziale
  - condivide spazi di memoria con heavyweight process

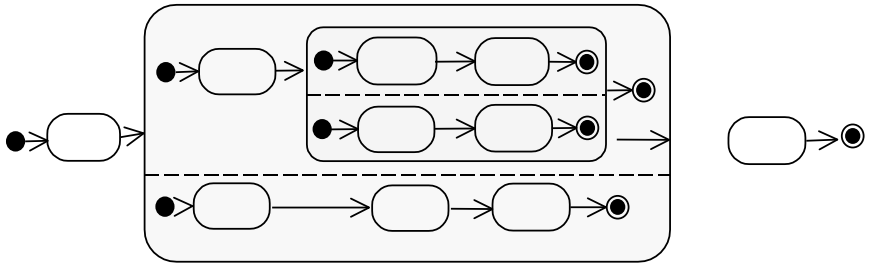
In questo modulo, le considerazioni di analisi fatte con thread di applicano anche ai task e viceversa

Thread



• Un thread rappresenta l'esecuzione **sequenziale** di istruzioni (o di attività), molte volte in un contesto concorrente.

Multithreading / multitasking



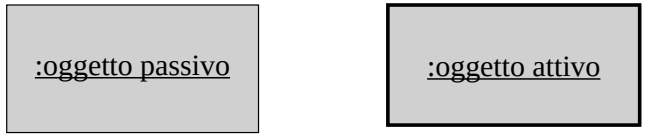
Oggetti attivi e passivi

• **Oggetti passivi:** aspettano un messaggio per invocare una operazione, non fanno partire mai delle attività. Non hanno thread di controllo.

• **Oggetti attivi:** sono task concorrenti. Hanno il loro thread di controllo (hanno la propria "vita"). Possono far partire delle azioni che coinvolgono altri oggetti.

• Una operazione di un oggetto passivo, una volta invocato da un oggetto attivo, va in esecuzione in un thread di controllo dell'oggetto attivo.

Oggetti attivi e passivi



Oggetti attivi e passivi (2)

· Oggetti contemporaneamente attivi e passivi : in Java, e' possibile per un oggetto incapsulare un thread, ma anche avere operazioni (metodi) che possono essere invocate da altri thread.

Problemi in thread concorrenti

- **Mutua esclusione** (su risorse condivise)
  (es. uso di semafori)
- **Sincronizzazione**
  (es.con segnali asincroni)
- **Produttore consumatore**
  (es. con monitor)

Mutua esclusione

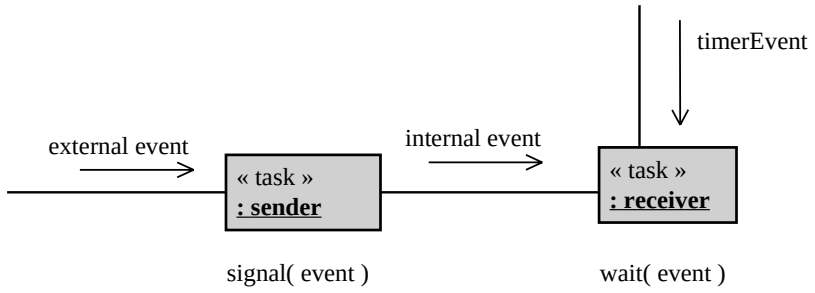
aquire (Semaforo); *// oppure wait (S)*

X.usoDellaRisorsaCondivisa( ); *// sezione critica*

release (Semaforo); *// oppure signal(S)*

Dijkstra:
P(semaforo)
SezioneCritica
V(semaforo)

Sincronizzazione con messaggi
con o senza comunicazione dati



send
receive

Sincronizzazione con messaggi

send (**in** dest, **in** msg)
- il processo inviante e' bloccato fino a che il msg è ricevuto
- il processo inviante invia il msg e continua

dest/sorg:
- proc.
- mailbox

receive(**in** sorg, **out** msg)
- se il msg è già stato inviato, viene acquisito e l'esecuzione continua
- se il msg non è ancora stato inviato:
  - il proc. è bloccato in attesa;
  - il proc. continua perdendo il msg.

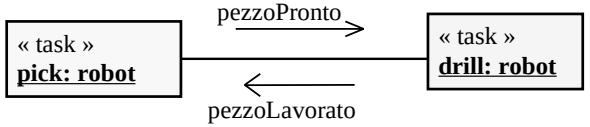
Bloking send - bloking receive: tight synchronization o rendezvous

Nonbloking send - bloking receive: la più comune

Nonbloking send - Nonbloking receive

Es. sincronizzazione *event signal* senza comunicazione dati

(senza conferma => canale asincrono)



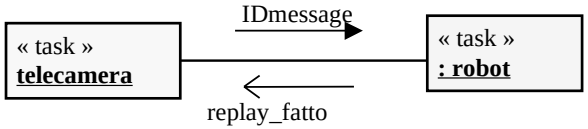

```
while( lavoroDaFare )
        preparaPezzo( )
        posizionaPezzo( )
        signal( pezzoPronto )
        wait( pezzoLavorato )
        prelevaPezzo( )
endwhile
```

```
while( lavoroDaFare )
        wait( pezzoPronto )
        lavoraPezzo( )
        signal( pezzoLavorato)
endwhile
```

... quando la situazione è più complessa pericolo di **deadlok**

Tightly coupled message communication whith replay

(con conferma => canale sincrono)

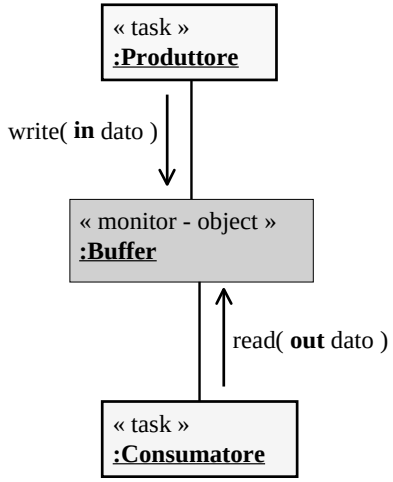



```
while( lavoroDaFare )

        identificaPezzo( )
        send ( IDmessage )
        // attesa conferma
        wait ( replay_fatto )

endwhile
```

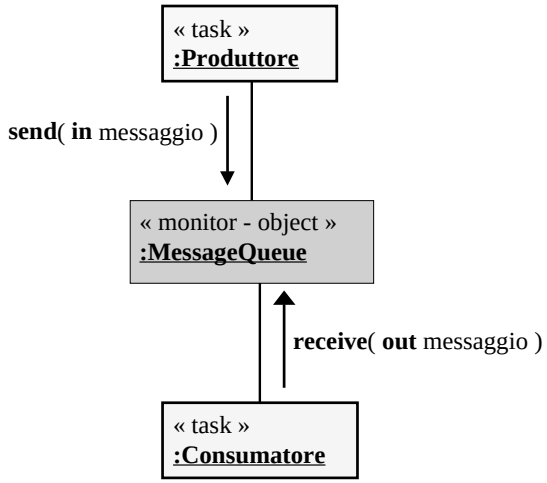
```
while( lavoroDaFare )

        receive( IDmessage ) *
        lavoraPezzo( ... )
        send( replay_fatto )

endwhile
```

* puo' essere la receive stessa che invia conferma di ricevuto

Produttore consumatore con monitor



Inter task communication con monitor “queue connector”

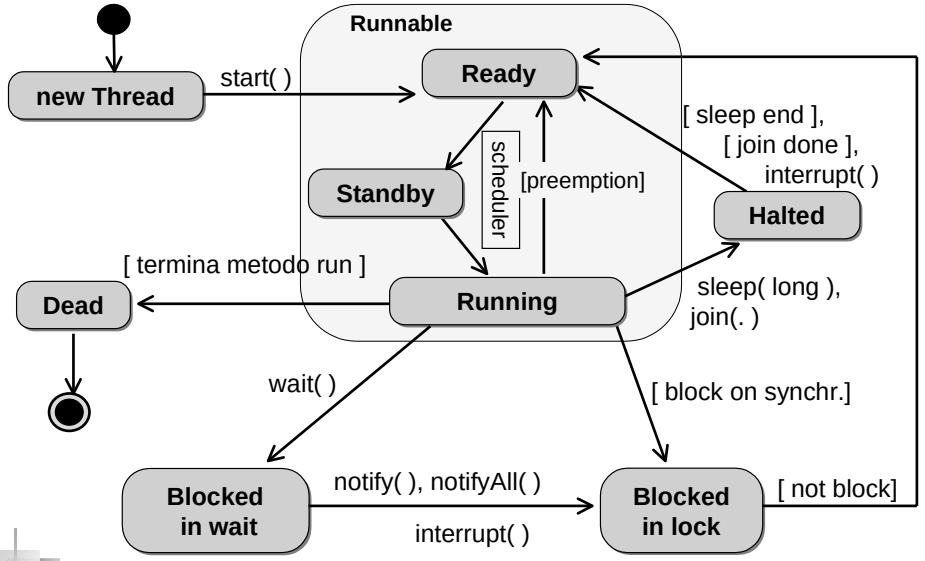


Inter task communication con monitor “queue connector”

```
class   MessageQueue:  Monitor
            coda: Tcoda

 public send( in messaggio )
            while ( coda.codaPiena( ) ) wait
            deposita messaggio in coda
            signal
  end send

 public receive( out messaggio )
            while ( coda.codaVuota( )  ) wait
            preleva messaggio dalla coda
            signal
 end send

endclass MessageQueue
```


Thread e sincronizzazione in Java

Thread java: stati

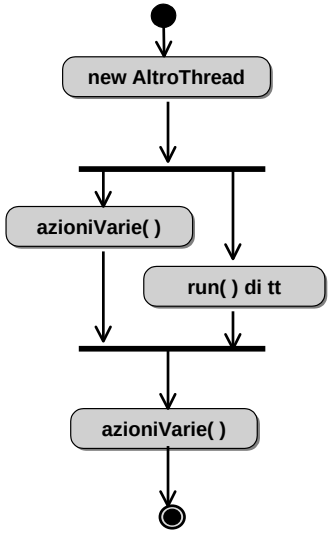


Thread java: metodi *(1)*

join( ) il thread corrente aspetta che il thread, sul quale la join è invocata, termini

join(long t) aspetta per **t** millisecondi che il thread, sul quale la join è invocata, termini

Thread java: metodi *(1)*




```
// questo è un metodo di un  thread xx
public void provaJoin( )
{
     AltroThread tt =  new AltroThread( );
     tt.start( ); // fork

     azioniVarie( ); // thread xx   in parallelo con  tt

     try {   tt.join( ); // thread xx     aspetta  tt
     }    catch( interruptException e) {   }

     azioniVarie( );  // thread xx    continua senza tt

} // provaJoin
```


Thread java: metodi *(2)*

static **sleep**(long t) ; *// aspetta **t** millisecondi*

Thread java: metodi *(3)*

wait( )

aspetta finche' un altro thread invochi una

notify( ) , od una **notifyAll**( )

sono metodi invocati su un oggetto rendezvous

da metodi synchronized

wait(long t)

taxista cliente e taxi

Thread java: metodi *(4)*

interrupt( ); // *interrompe il thread*

(Java)

```
class SimpleThread extends Thread
{
        public SimpleThread( String str )
        {       super(str);             } // nome del thread

         public void run( )  // override il metodo vuoto
         {      for (int i = 0; i < 10; i++)
                 {  System.out.println(i + " " + getName( ));
                    try { sleep((long)(Math.random( ) * 100));
                     } catch (InterruptedException e) { }
                 } //for
            System.out.println("DONE! " + getName());
         } //run
} //SimpleThread
```


(Java)

```
public class TwoThreadsDemo
{
       public static void main( String[ ] args )
       {
          new SimpleThread(”Pippo").start( );
          new SimpleThread(”Pluto").start( );
        } //main
}
```

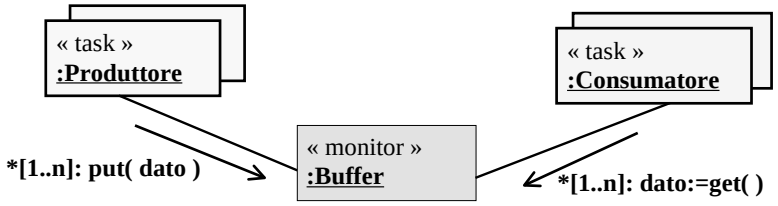
```
0 Pippo
0 Pluto
1 Pippo
2 Pippo
1 Pluto
3 Pippo
2 Pluto
3 Pluto
...
```


(Java) thread e interface runable

```
public class InterfThread
          extends   ImportantClass
          implements   Runnable
{
        public void run( )
          {
                . . .   // azioni varie
          }
}
```

```
new Thread( new InterfThread ( ) ).start( );
```

Produttore consumatore in Java



Produttore consumatore in Java (1)

```
public class Produttore extends Thread
{       private Buffer buffer;   // reference
        private int pin;

   public Produttore ( Buffer c, int number )
    {       buffer = c;   // reference
            this. pin = pin;
     }

   public void run( )
    {     for (int i = 0; i < 10; i++)
          {  buffer.put(i);
             System.out.println("Prod. #" + pin + " put: " + i);
             /* ...   sleep statement */
          } //for
    }
} // Produttore
```

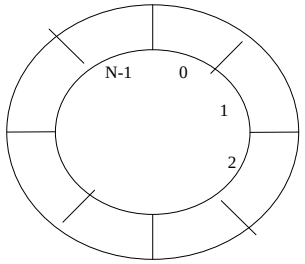

Produttore consumatore in Java (2)

```
public class Consumatore   extends   Thread
{       private Buffer buffer; // reference
        private int  pin;

   public Consumatore ( Buffer c, int pin)
    {     buffer = c; // reference
          this. pin = pin;
     }

   public void run( )
    {     int value;
          for ( int i = 0; i < 20; i++ )  // non elegante: si prevedono due produttori da 10
          {    value = buffer.get();
               System.out.println("Consumer #" + pin + " got: " + value);
          } //for
    } //run
} // Consumatore
```


Produttore consumatore in Java (3)

```
public class Buffer
{
        static final  int  N = 10;
        private int g = 0,  p= 0;
        private int  vettC [ ]= new int[N];

        public synchronized int get( )
          { ... }

        public synchronized void put( int value )
          { ... }

 } // Buffer
```

accesso mutuamente esclusivo sull'oggetto Buffer ai metodi synchronized

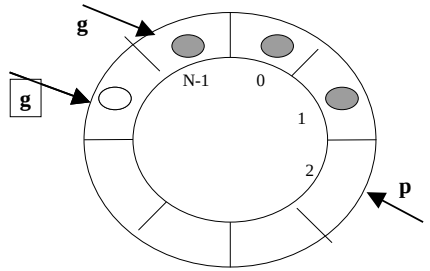


Produttore consumatore in Java (4)

```
public class ProduttoreConsumatoreTest
{
        public static void   main( String[] args )
        {       Buffer c =  new Buffer( );
                Produttore   p1 = new Produttore (c, 1);
                Produttore   p2 = new Produttore (c, 2);
               Consumatore  c1 = new Consumatore (c, 1);

                   p1.start( );    p2.start(  );

                   c1.start();

        } //main

 }  // ProducerConsumerTest
```

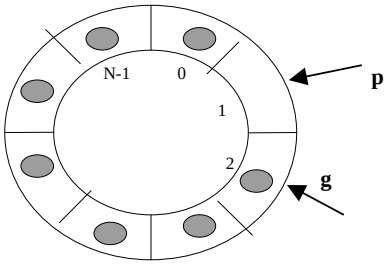

Produttore consumatore in Java (5)

```
public synchronized int get( )
{       while  (g==p ) // vuoto
        {       try {   wait( ); // aspetta che un Produttore inserisca un valore
                } catch (InterruptedException e) {  }
        }

       notifyAll(); // sveglia tutti il thread che attendono sul buffer
        int x= vettC[g];
       g=(g+1)%N;
       return x;

} //get
```

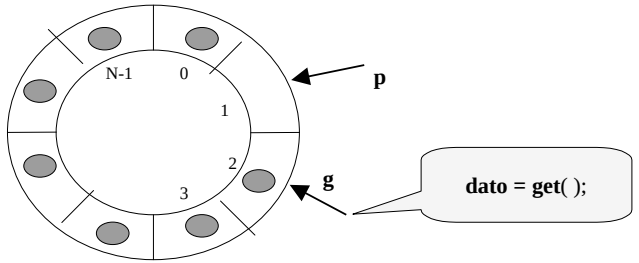


Produttore consumatore in Java (6)

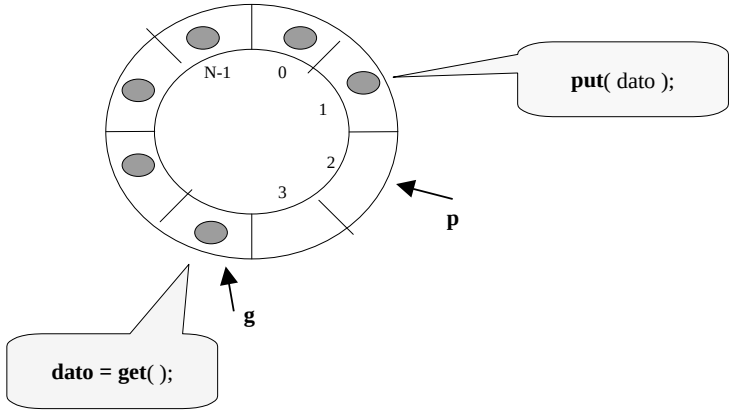
```
public synchronized void put( int value )
{       while ( (p+1) % N== g ) // pieno
        {       try {   wait( ); // aspetta che un Consum. prelevi
                } catch (InterruptedException e) { }
        }
        vettC[p] = value;
        p= (p+1) % N;
        notifyAll(); // notifica che c'è un valore disponibile
} //put
```



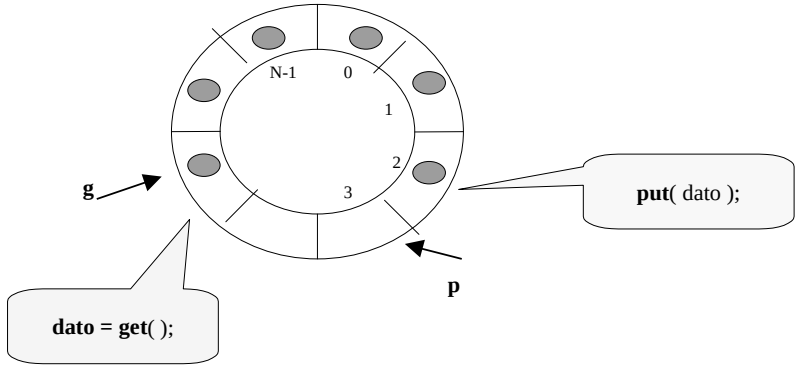
Produttore consumatore in Java (7)



Produttore consumatore in Java (9x)



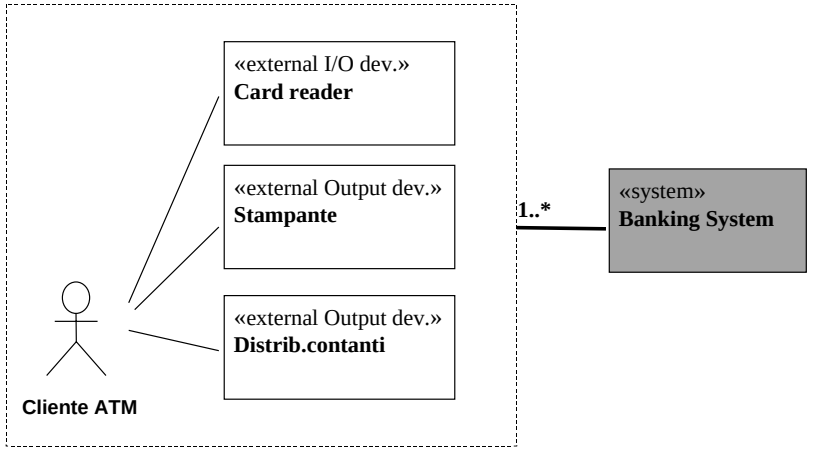
Produttore consumatore in Java (10x)



Studio di caso: ATM

(alcuni cenni)

Modello statico del contesto: classi external, system



ATM: Use case sottosistemi Client / Server

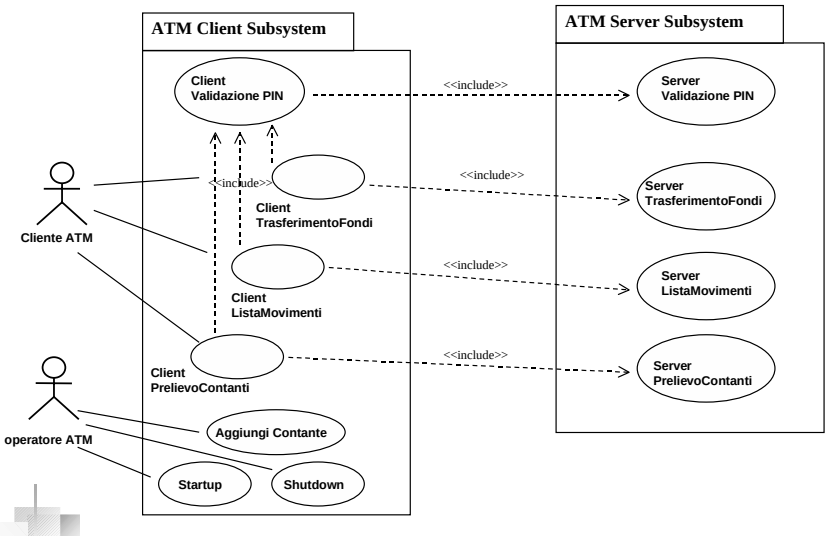


Diagramma di collaborazione Banking system

diagramma concorrente ed ad alto livello con oggetti attivi

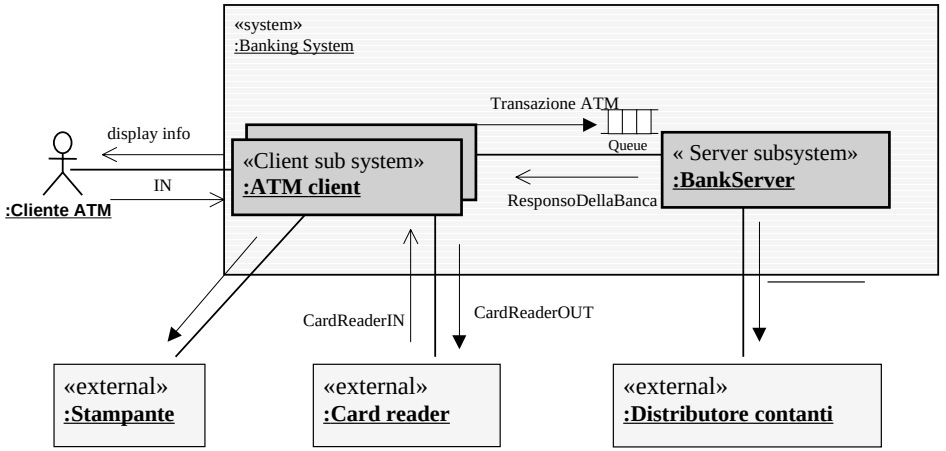


Diagramma di collaborazione BankServer subsystem