

# **Programmazione concorrente in ADA**

# Linguaggio ADA

- Sviluppato per conto del DOD (Department Of Defense) degli Stati Uniti.
- Applicazioni tradizionali, distribuite ed in tempo reale.
- Unità di concorrenza: **task**
- Adotta come metodo di interazione tra i processi (task) il rendezvous esteso.

## Il linguaggio ADA: origini

- Ada è un linguaggio creato inizialmente per scopi militari
- 1980 : primo standard in ambito militare, MILSTD1815
- All compilers must be validated against the standard
- 1983 : primo US standard (ANSI)
- 1987 : primo standard in ambito internazionale, ISO 8652
- 1995 : Ada 95 is the first standard object-oriented language
- 2005 : definizione di Ada 2005

## Chi usa ADA?

- Aeronautica e Spazio:
  - Eurofighter: 2 million lines of Ada code
  - Boeing (in progress: 7E7)
  - Airbus (in progress: A380, A400M)
  - Ariane Satellites Eurocontrol (air traffic control)
- Industria ferroviaria:
  - French highspeed train (TGV), Metro lines in New York, Paris (line 14), Delhi, Calcutta, etc.
- Nuclear industry Electricité de France (EDF): emergency reactor shutdown
- Applicazioni Finanziarie: BNP (France), Paranon (Switzerland), PostFinance
- Healthcare and medical : JEOL (USA), ReadySoft (France)
- Automotive : BMW
- TLC: Canal+ (France)
- Numerose applicazioni in ambito free software!

## ADA: obiettivi

- Ridurre i costi di sviluppo e manutenzione.
- Prevenire bugs
- Rilevare bugs il prima possibile (compilazione)
- Favorire riutilizzo e sviluppo in team:
  - Packages with separate interface and implementation
  - Generics (a.k.a. templates)
- Semplificare la manutenzione: leggibilità e autodocumentazione.
- Adatto per lo sviluppo:
  - In the small: embedded, risorse limitate, real time.
  - In the large: millions of lines of code, networking, GUIs, etc.

# Certificazioni: adacore.com

«Sin dalla fondazione dell'azienda a metà degli anni '90, AdaCore ha servito i clienti nei settori più esigenti in termini di sicurezza. Con questa esperienza è maturata una profonda esperienza in una serie di importanti **standard di certificazione di sicurezza del software**, tra cui:

- DO-178B/C (avionics)
- EN 50128 (railway)
- ECSS-E-ST-40C / ECSS-Q-ST-80C (space)
- IEC 61508 (industrial automation)
- ISO 26262 (automotive).

I nostri strumenti e la nostra tecnologia soddisfano i requisiti di certificazione più esigenti, dal punto di vista della funzionalità delle caratteristiche e della garanzia di sicurezza.» [Dal sito web di adacore, società che mantiene e distribuisce il linguaggio.]

# ADA: tipi di dato

Ada è un linguaggio **fortemente e staticamente tipato**:

- La maggior parte degli errori possono essere rilevati in fase di compilazione.
- Il programmatore può definire nuovi tipi per esprimere pienamente le caratteristiche del dominio applicativo:
  - Due tipi diversi non possono essere confrontati (errori rilevati a compile time)

# Tipi scalari

```
package Apples_And_Oranges is
type Number_Of_Apples is range 1 .. 20;  --integer
type Number_Of_Oranges is range 1 .. 40;  --integer
type Mass is digits 4 range 0.0 .. 4000.0;  --real
type Colour is (Red, Green, Blue);  --enumeration
end Apples_And_Oranges;
```

Number\_Of\_Apples, Number\_Of\_Oranges: particolari Integer che riflettono i vincoli del problema.

- I tipi scalari sono caratterizzati anche da **attributi**:
  - **T'First**, **T'Last** : costanti che individuano estremi inferiore e superiore di un intervallo.
  - **T'Range = T'First .. T'Last** (intervallo di valori permessi)
  - **T'Pred(X)**, **T'Succ(X)** : funzioni che ritornano il valore precedente o successivo nel dominio
  - **T'Image(X)** : la rappresentazione di X come stringa (utile per la stampa).



# Array

**A:** **array** (4 .. 8) of Integer;

Gli array hanno **attributi**: **Range**, **Length**

Ad esempio:

**A' Range** è l'intervallo dell'indice

**A' Range** = 4 .. 8;

**A' Length** è la dimensione dell'array (5)

# Record

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Record1 is
  type DATE is
    record
      Month : INTEGER range 1..12;
      Day   : INTEGER range 1..31;
      Year  : INTEGER range 1776..2018;
    end record;

  Independence_Day : DATE;

begin
  Independence_Day.Month := 7;
  Independence_Day.Day := 4;
  Independence_Day.Year := 1776;
end Record1;
```

# Puntatori: access

I puntatori si definiscono tramite il costrutto **access**.

Non è possibile combinare in espressioni puntatori a tipi diversi (tipaggio forte):

```
procedure Access1 is
  type POINT_SOMEWHERE is access INTEGER;
  Index: POINT_SOMEWHERE;

begin
  Index := new INTEGER;
  Index.all := 13; --dereferencing
  Put("The value is");
  Put(Index.all, 6); New_Line;
end Access1;
```

# Istruzioni di controllo

## Alternativa: if

```
if Index < 15  
    then Put_Line(" and is less than 15.");  
    else Put_Line(" and is 15 or greater.");  
end if;
```

# Istruzioni di controllo

## Ripetizione : loop

```
with Ada.Text_IO, Ada.Integer_Text_IO;  
use Ada.Text_IO, Ada.Integer_Text_IO;
```

```
procedure LoopDemo is  
    Index, Count : INTEGER;  
  
begin  
    Index := 1;  
    loop  
        Put("Index =") ;  
        Put(Index, 5); New_Line;  
        Index := Index + 1;  
        exit when Index = 5;  
    end loop;  
end LoopDemo;
```

# Istruzioni di controllo:

## Ripetizione - while:

```
while  Index < 5
loop
    Put ("Index =") ;
    Put (Index, 5) ; New_Line;
    Index := Index + 1;
end loop;
```

# Istruzioni di controllo

## Ripetizione: for

```
with Ada.Text_IO, Ada.Integer_Text_IO;  
use Ada.Text_IO, Ada.Integer_Text_IO;
```

```
procedure ForDemo is
```

```
    Index: INTEGER;
```

```
begin
```

```
    Index := 1;
```

```
    for Index in 1..4
```

```
        Put("Doubled index =");
```

```
        Put(2 * Index, 5); New_Line;
```

```
    end loop;
```

```
end ForDemo;
```

# Concorrenza

L'esecuzione concorrente è basata sul concetto di **task**:

- Processo  $\Rightarrow$  **task**
- Programma concorrente  $\Rightarrow$  **procedure** (Insieme di)

Un Task descrive un'attività che può essere eseguita in concorrenza con altre:

```
task <nome_task> is ... end <nome_task>; --dichiarazione
```

```
task body <nome_task> is ... --definizione  
end <nome_task>;
```



# Task: esempio con 3 processi

```
with Ada.Text_IO, Ada.Integer_Text_IO; --importaz.package
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Task1 is -- procedure "main"
  task First_Task; --def primo processo
  task body First_Task is
  begin --corpo
    for Index in 1..4 loop
      Put("This is in First_Task, pass number ");
      Put(Index, 3);
      New_Line;
    end loop;
  end First_Task;
-- continua..
```

```
task Second_Task;  -- def. secondo processo
  task body Second_Task is
  begin  -- corpo
    for Index in 1..7 loop
      Put("This is in Second_Task, pass number");
      Put(Index, 3);
      New_Line;
    end loop;
  end Second_Task;

begin  -- def main
  Put_Line("Questo è il main task..");
end Task1;
```

# Interazione tra task

- Comunicazione di tipo **asimmetrico** a **rendez vous esteso**.
- Ogni task può definire delle operazioni pubbliche (**entry**) visibili da altri task.
- Una entry definita in un task P e resa visibile all'esterno di P, può essere chiamata da un altro task Q.
- L'interazione tra Task avviene attraverso il meccanismo del **rendez-vous**: ad es, il Task Q chiama una entry E di del task P(**entry call**); Q attende che l'esecuzione di E da parte di P sia completata

# Entry

Una entry è un'operazione che un task rende disponibile agli altri task:

```
task S is      -- dichiarazione
    entry E (<lista_parametri>) ;
end S;

task body S is  --definizione
begin
<definizione di S e delle sue entry>
end S;
```

I parametri possono essere di tipo IN, OUT.

# Rendez-vous: call e accept

Una entry dichiarata in un task server S e resa visibile all'esterno di S, può essere chiamata da un task cliente C mediante call:

```
S.entryname (<parametri effettivi>);
```

La comunicazione tra C e S avviene quando S esprime la volontà di eseguire la entryname mediante **accept**:

```
accept entryname (in <par-in>,out <par-out>);  
do I1; I2;..In;  
end entryname;
```

Durante l'esecuzione delle istruzioni I1,..In i task C e S rimangono sincronizzati: C attende la terminazione della entry.

## Accept

```
accept entryname (in <par-in>,out <par-out>) ;  
do I1; I2;..In;  
end entryname;
```

- L'esecuzione di **accept entryname** da parte di S sospende il task fino a quando non avviene una chiamata di entryname.
- In quel momento i **parametri effettivi** sono **copiati** nei **parametri formali di ingresso** ed S esegue la lista di istruzioni I1, .. In.
- Al loro completamento i **risultati** sono **copiati** nei **parametri di uscita** e termina la sincronizzazione tra S e C.

**Coda associata ad ogni entry** (gestita FIFO): una stessa entry può essere chiamata da più task prima che il task che la definisce esegua la corrispondente accept; ogni richiesta non ancora servita viene accodata.

Ad una stessa entry possono essere associate più accept. Ad esse possono corrispondere azioni diverse a seconda della fase di esecuzione del task.

# Interazione esempio

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure HotDog is
    task Gourmet is    --dichiarazione
        entry Make_A_Hot_Dog;
    end Gourmet;

    task body Gourmet is    --definizione
    begin
        for Index in 1..4 loop
            accept Make_A_Hot_Dog do --def.entry
                delay 0.8; --cfr. sleep
                Put("Metto hot dog nel pane..");
                Put_Line("Aggiungo senape");
            end Make_A_Hot_Dog;
        end loop;
    end Gourmet;
```

```
begin  --task main
    for Index in 1..4 loop
        Gourmet.Make_A_Hot_Dog; --entry call
        delay 0.1;
        Put_Line("Mangio l'hot dog");
        New_Line;
    end loop;

end HotDog; -- fine procedure
```



# Interazione: select

Un Task (server) può esporre più operazioni e accettare le richieste attraverso il comando con guardia alternativo **select**:

```
select  
    accept E(...) do  
    ..  
end  
  
or  
  
    accept A(...) do  
    ..  
end  
  
end select;
```

**Ad ogni accept è associata una coda:**

- Se entrambe le code sono vuote, select sospende il task
- Se almeno una contiene una richiesta, viene fatta una selezione non deterministica su quella che verrà servita.

## Usi della select:

- **Selezione non deterministica: or**

select

<accept di una entry E1>;

or

<accept di una entry E2>;

or..

..

end select;

- **Conditional entry call: else**

select

    <chiamata ad una entry E>;

    else <istruzioni>; -- eseguite, se non ci  
                          --sono richieste per E

end select;

## Selezione con deadline:

```
select
    <accept di una entry>
    or
    ...
    or
    delay <intervallo di tempo T>;
end select;
```

Il task chiamante attende, per l'esecuzione del comando, al più un tempo pari all'intervallo T.

# Select esempio

```
with ada.text_io;                -- include libreria text_io
with ada.integer_text_io;        -- include libreria
integer_text_io
procedure task_demo is
    task type intro_task is      -- dichiarazione tipo di task
        entry start;             -- dichiarazioni entry
        entry turn_left;
        entry turn_right;
        entry stop;
    end intro_task;
```

```

task body intro_task is
begin
    accept start; --def. entry
    loop
        select
        accept turn_left; --def. entry
            Put_line ("turning left");
        or
        accept turn_right; --def. entry
            Put_line ("turning right");
        or
        accept stop;      --def. entry
            Put_line ("stop received");
            exit;          -- exit the loop
        else
            Put_line ("moving straight");
        end select;
        delay 0.5;
    end loop;
end intro_task;

```

```
--continua: main:
```

```
task_1 : intro_task; -- creazione task
```

```
begin
```

```
    task_1.start;
```

```
    delay 2.0;
```

```
    task_1.turn_left;
```

```
    delay 2.0;
```

```
    task_1.turn_right;
```

```
    delay 1.0;
```

```
    task_1.turn_right;
```

```
    delay 2.0;
```

```
    task_1.stop;
```

```
end task_demo;
```

# Select con guardie logiche

```
select
    when condizionale1 => accept E1(...)
        do ...
        end E1;

    or

    when condizionale2 => accept E2(...)
        do ...
        end E2;

end select;
```

## Comando con guarda ripetitivo

```
loop
  select
    when condizione1 => accept E1(...)
      do ...
      end E1;

    or

    when condizione2 => accept E2(...)
      do ...
      end E2;

    or ...
  end select;
end loop;
```



## Esempio produttore e consumatore

```
loop
  select
    when pieno < Bufsize =>
      accept Inserisci( v : IN Integer) do
        ...
      end Inserisci;
    or
      when pieno > 0 =>
        accept Estrai( v : out Integer) do
          ...
        end Estrai;
    end select;
end loop;
```

## Esempio completo\*

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure Retail2 is
  Number_Of_Dogs : INTEGER := 0;
  task Five_Dogs;           -- aggiunge 5 hot dog
  task Remove_Five_Dogs; -- rimuove 5 hot dog

  task Retail_Hot_Dogs is
    entry Stock_With_A_Hot_Dog; -- aggiunge 1 HD
    entry Deliver_A_Hot_Dog;    -- rimuove 1 HD
  end Retail_Hot_Dogs;
```

----

\*v. [https://perso.telecom-paristech.fr/pautet/Ada95/e\\_c27\\_p1.ada](https://perso.telecom-paristech.fr/pautet/Ada95/e_c27_p1.ada)

```

task body Retail_Hot_Dogs is
begin
    for Index in 1..1000 loop
        select
            when Number_Of_Dogs < 8 =>
            accept Stock_With_A_Hot_Dog do
                Number_Of_Dogs := Number_Of_Dogs + 1;
                Put_line("aggiunto 1 HD");
            end Stock_With_A_Hot_Dog;
        or
            when Number_Of_Dogs > 0 =>
            accept Deliver_A_Hot_Dog do
                Put_Line("tolto 1 HD");
                Number_Of_Dogs := Number_Of_Dogs - 1;
            end Deliver_A_Hot_Dog;
        end select;
    end loop;
end Retail_Hot_Dogs;

```

```
task body Five_Dogs is
begin
    for Index in 1..5 loop
        delay 0.1;
        Retail_Hot_Dogs.Stock_With_A_Hot_Dog;
    end loop;
end Five_Dogs;
```

```
task body Remove_Five_Dogs is
begin
    for Index in 1..5 loop
        delay 0.6;
        Retail_Hot_Dogs.Deliver_A_Hot_Dog;
    end loop;
end Remove_Five_Dogs;
```

## Esempio: pool di risorse equivalenti

- 3 tipi di task:
  - **Server**: gestore del pool di risorse
  - **Risorsa**: gestore di una risorsa del pool
  - **Cliente**(id): task che usa le risorse del pool

## DICHIARAZIONI:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure pool1 is
  type cliente_ID is range 1..10;           -- max 10 clienti
  type ris_ID is range 1..5;                -- 5 risorse nel pool

  task type cliente (ID: cliente_ID); -- dichiaraz. task cliente
  type ac is access cliente;           -- riferimento ad un cliente

  task type risorsa (ID: ris_ID) is -- dichiaraz. gestore di 1 risorsa
    entry operazione(ID: in cliente_ID; OK: out Boolean);
  end risorsa;

  type ar is access all risorsa; -- rif. ad un task gestore di risorsa

  type pool is array(ris_ID'Range) of ar;

  task type server is -- processo gestore del pool
    entry Richiesta (ID: in cliente_ID; RIS: out ar; IND: out ris_ID);
    entry Rilascio(ID: in cliente_ID; IND: in ris_ID);
  end server;
```

## Definizione Cliente:

P: pool; --creazione pool

S: server; -- creazione server

task body cliente is

GR: ar;

INDEX: ris\_ID;

OK: Boolean;

begin

Put\_Line ("cliente" & cliente\_ID'Image (ID) & " !");

S. Richiesta(ID, GR, INDEX);

Put\_Line ("cliente ha ottenuto una risorsa!");

GR.operazione(ID, OK); ---Accesso alla risorsa

S.Rilascio(ID, INDEX);

Put\_Line ("cliente terminato!");

end;

## Definizione server:

```
task body server is -- dichiarazione server
    disp: Integer;
    k: ris_ID;
    libere: array(ris_ID'Range) of Boolean;
begin
    k:=1;
    Put_Line ("SERVER iniziato!");
    -- inizializzazione stato server:
    disp:=5;
    for i in ris_ID'Range loop
        libere(i):=True;
    end loop;
    -- continua
```



```

loop
select
  when disp > 0 =>  -- c'è almeno una risorsa libera
  accept Richiesta (ID: in cliente_ID; RIS: out ar; IND: out ris_ID)
  do
    Put_Line ("server ha ricevuto richiesta dal cliente
    "& cliente_ID'Image(ID) &" !");
    for k in ris_ID'Range loop
      if libere(k)=True
      then
        libere(k) :=False;
        disp:=disp-1;
        IND:=k;
        RIS :=P(k) ;
        exit;
      end if;
    end loop;
  end Richiesta;          -- fine servizio
or

```

```
accept Rilascio(ID: in cliente_ID; IND: in ris_ID)
do
    libere(IND) := True;
    disp := disp + 1;
end Rilascio; -- fine servizio

end select;
end loop;
end; -- fine server
```

## Def. risorsa

```
task body risorsa is
  mioid: ris_ID;
begin
  Put_Line ("risorsa " & ris_ID'Image (ID) & " ! ");
  mioid:=ID;
  loop
    select
      accept operazione (ID: in cliente_ID; OK: out
        Boolean) do
        Put_Line ("risorsa " & ris_ID'Image (mioid) & " in uso da " &
          cliente_ID'Image (ID) & " ! ");
        delay 1.0;
        OK:=True;
      end operazione;
    end select;
  end loop;
end risorsa;
```

## Def. “main”

```
New_client: ac;
```

```
begin -- equivale al main
```

```
    for I in ris_ID'Range loop
```

```
        P(I):=new risorsa(I);
```

```
    end loop;
```

```
for I in cliente_ID'Range loop -- creaz. task
```

```
    New_client := new cliente (I);
```

```
end loop;
```

```
end pool1; -- fine programma
```

## Risorse utili

- Compilatore linux: **gnat**
- Comando per compilazione:

```
gnat make programma.adb
```

- Per download pugin ADA per eclipse, xcode, etc:

<http://www.adacore.com>

Ide gratuito: **Gnat Programming Studio (gps)**

- Tutorial ADA on line:

<http://www.infres.enst.fr/~pautet/Ada95/a95list.htm>

## Esempi

## Esempio 1: ponte a senso unico alternato

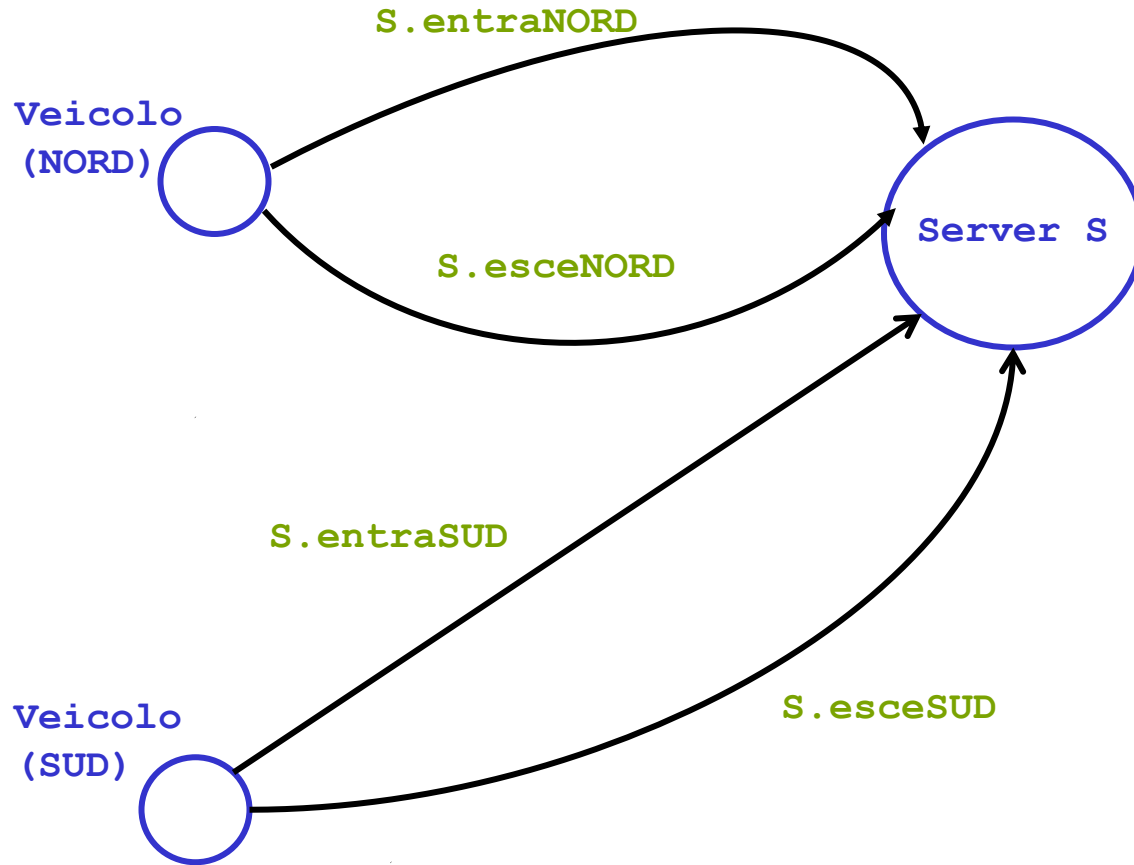
Si consideri un ponte a senso unico con capacità limitata a MAX veicoli.

Ogni veicolo che vuole entrare dalla direzione X è autorizzato se:

- c'è posto sul ponte (il numero di veicoli è minore di MAX)
- non ci sono veicoli in direzione opposta a X.

Realizzare un'applicazione distribuita ADA in cui i veicoli siano rappresentati da task concorrenti (clienti) e la gestione del ponte sia affidata ad un task (servitore).

## Schema soluzione





# Impostazione

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure ponte is
    type cliente_ID is range 1..10;  -- 10 clienti
    type dir_ID is (NORD, SUD);      -- direzioni

    -- processo gestore del pool:
    task type server is
        entry entraNORD (ID: in cliente_ID );
        entry esceNORD (ID: in cliente_ID );
        entry entraSUD (ID: in cliente_ID );
        entry esceSUD (ID: in cliente_ID );
    end server;
    S: server;      -- creazione server
```

- Task client: rappresenta l'utente del ponte.

```
task type cliente (ID: cliente_ID; DIR: dir_ID);

task body cliente is
begin
    Put_Line ("gruppo" & cliente_ID'Image (ID) & " di "&
              dir_ID'Image (DIR) & "iniziato!");

    loop
        if DIR=NORD
        then
            S. entraNORD(ID);
            delay 1.0;
            S. esceNORD(ID);
            delay 1.0;
        end if;
        if DIR=SUD
        then
            S. entraSUD(ID);
            delay 1.0;
            S. esceSUD(ID);
            delay 1.0;
        end if;
    end loop;
end;
```

- Definizione task server:

```
task body server is
    MAX : constant INTEGER := 5; --capacità ponte
    sulponte: Integer;
    utenti: array(dir_ID'Range) of Integer;

begin
    Put_Line ("SERVER iniziato!");
    --INIZIALIZZAZIONI:
    sulponte:=0;
    for i in dir_ID'Range loop
        utenti(i):=0;
    end loop;
    -- continua..
```

```

-- .. Gestione richieste
loop
    select
        when sulponte < MAX and utenti(SUD)=0 =>
        accept entraNORD (ID: in cliente_ID ) do
            utenti(NORD):=utenti(NORD)+1;
            sulponte:=sulponte+1;
            end entraNORD;    -- fine sincron.

    or

        when sulponte < MAX and utenti(NORD)=0 =>
        accept entraSUD (ID: in cliente_ID ) do
            utenti(SUD):=utenti(SUD)+1;
            sulponte:=sulponte+1;
            end entraSUD;

    -- continua..

```

```

-- .. continua
or
    accept esceNORD (ID: in cliente_ID ) do
        utenti(NORD):=utenti(NORD)-1;
        sulponte:=sulponte-1;
    end esceNORD;

or
    accept esceSUD (ID: in cliente_ID ) do
        utenti(SUD):=utenti(SUD)-1;
        sulponte:=sulponte-1;
    end esceSUD;

end select;
end loop;
end;

```

## Struttura programma e definizione main:

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure ponte is

  -- dichiarazioni e definizioni task ecc.
  ...

  type ac is access cliente; -- riferimento ad un task cliente
  New_client: ac;

  begin -- equivale al main
    for I in cliente_ID'Range loop -- ciclo creazione task
      New_client := new cliente (I); -- creazione cliente I-simo
    end loop;
  end ponte;
```

## In alternativa: selezione entry in base a parametri

### Vettore delle operazioni di servizio:

Per consentire la realizzazione di politiche dipendenti dal tipo di richiesta, dai parametri associati al task si può usare il concetto di *famiglie di entry*:

```
entry entryname (first..last) (in..out);
```

- Soluzione con 2 entries:

```
task type server is
```

```
    entry entra(dir_ID) (ID: in cliente_ID );
```

```
    entry esce(dir_ID) (ID: in cliente_ID );
```

```
end server;
```

- Definizione task server: struttura.

```
task body server is
    MAX : constant INTEGER := 5; --capacità ponte
    -- <variabili di stato del ponte>
begin
    --<inizializzaz. variabili di stato del ponte>
    loop--Gestione richieste:
        select
            ... accept entra(NORD) (ID: in cliente_ID ) do ...
        or
            ... accept entra(SUD) (ID: in cliente_ID ) do ...
        or
            ... accept esce(NORD) (ID: in cliente_ID ) do ...
        or
            ... accept esce(SUD) (ID: in cliente_ID ) do ...
        end select;
    end loop;
end;
```



## Task client (con family of entries).

```
task type cliente (ID: cliente_ID; DIR: dir_ID);

task body cliente is
begin
    Put_Line ("gruppo" & cliente_ID'Image (ID) & " di " &
              dir_ID'Image (DIR) & "iniziato!");

    S. entra(DIR) (ID);
    delay 1.0;
    S. esce(DIR) (ID);
end;
```

**Politiche basate su priorità**

## Politiche basate su priorità: necessità di selezionare la richieste di entrata da servire

- Vettore delle operazioni di servizio: **Family of entries**
- Es. 3 livelli di priorità: P1, P2 e P3
- Il gestore di una risorsa deve servire prima le richieste di priorità P1, poi P2, e per ultima, P3.

```
type prio is (P1, P2, P3);
```

```
task type server is  
    entry richiesta(prio) (<parametri formali>);  
    ..  
end server;
```

## Struttura server

- Primo metodo: usare il connettore **else** all'interno di select (possibilità di innestare select):

```
select
    accept richiesta(P1) ...
else
    select
        accept richiesta(P2) ...
    else
        select
            accept richiesta(P3) ...
            or
            delay 1.0;
        end select;
    end select;
end select;
```

## Struttura server

Secondo metodo (consigliato): usare l'attributo **'COUNT** (applicabile alle entries). Ad esempio:

**Richiesta (P1) ' COUNT**

Restituisce il numero di richieste in coda per la entry Richiesta(P1).

•SCHEMA da seguire:

```
select
    accept richiesta(P1) do.. end;
or
    when richiesta(P1)'COUNT=0 =>
        accept richiesta(P2) do.. end;
or
    when richiesta(P1)'COUNT=0 and
        richiesta(P2)'COUNT=0 =>
        accept richiesta(P3) do.. end;
end select;
```

## Esempio 2: la fabbrica di torte

Si consideri il laboratorio di un'azienda artigianale che produce dolci. L'azienda è specializzata nella produzione di torte; in particolare, i tipi di torte prodotti sono 2:

- Torta al cioccolato,
- Crostata alla marmellata.

Le torte vengono vendute in scatole pre-confezionate. L'azienda commercializza 3 tipi di confezioni:

- confezione semplice “Cioccolato”, contenente 1 torta al cioccolato;
- confezione semplice “Marmellata”, contenente 1 crostata.
- Confezione multipla “Famiglia”, contenente 1 torta al cioccolato e 1 crostata.

Nel laboratorio dell'azienda vi è un tavolo per il deposito delle torte in attesa di essere confezionate al quale accedono:

- gli operai dedicati alla produzione (OP), che accedono ciclicamente al tavolo per depositarvi ogni torta appena sfornata; ogni OP deposita sul tavolo 1 torta alla volta.
- gli operai dedicati alle confezioni (OC), ognuno dedicato alla confezione di scatole di un tipo predefinito dato (Cioccolato, Marmellata o Famiglia); essi accedono ciclicamente al tavolo per prelevare la/le torte necessaria/e a realizzare la confezione del tipo assegnato.

Il tavolo ha una capacità massima pari a  $MaxC$ , costante che esprime il massimo numero di torte che possono stare contemporaneamente su di esso.

Si sviluppi un'applicazione distribuita ADA, che rappresenti operai (clienti) e gestore del tavolo(server) con task concorrenti.

L'applicazione deve realizzare una politica di gestione del tavolo che soddisfi i vincoli dati e che, inoltre, soddisfi i seguenti vincoli di priorità:

- tra gli operai OP: i portatori di crostate siano favoriti rispetto ai portatori di torte al cioccolato;
- tra gli operai OC: gli operai dedicati alla confezione di scatole Famiglia siano favoriti rispetto a quelli dedicati alle scatole semplici (Cioccolato, Marmellata); inoltre, tra gli OC dedicati alle confezioni semplici, venga data priorità alle confezioni “Marmellata”.



## Impostazione server: famiglie di entries

```
type torta is (cioccolata, marmellata);  
type confezione is (cioc, marm, family);  
  
task type server is  
  entry deposito(torta) (<par. formali>);  
  entry prelievo(confezione) (<par. formali>);  
end server;  
  
S: server;
```

## Impostazione clienti OP/OC

```
type cliente_ID is range 1..10;
```

```
task type clienteOP (ID: cliente_ID; T:torta);  
  task body clienteOP is  
    begin  
      S. deposito(T) (ID);  
    end;
```

```
task type clienteOC (ID: cliente_ID; C:confezione);  
  task body clienteOC is  
    begin  
      S. prelievo(C) (ID);  
    end;
```

## Politica

- E' realizzata all'interno del server:

```
task body server is
    <variabili locali per rappr. Stato risorsa>
begin
    <INIZIALIZZAZIONI>
    loop
    select
        <accettazione/definizione entries>
    end select;
    end loop;
end server;
```

## Soluzione Completa

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;

procedure torte is
  type clienteOP_ID is range 1..10;
  type clienteOC_ID is range 1..4;
  type torta is (cioccolato, marmellata);
  type confezione is (cioc, marm, family);
  task type clienteOP (ID: clienteOP_ID; T:torta);
  task type clienteOC (ID: clienteOC_ID; C:confezione);
  type acOP is access clienteOP;
  type acOC is access clienteOC;

  task type server is
    entry deposito(torta) (ID:clienteOP_ID);
    entry prelievo(confezione) (ID:clienteOC_ID);
  end server;
```

```
S: server;
```

```
task body server is
```

```
    MAX : constant INTEGER := 18; -- capacita' tavolo
```

```
    sultavolo: array(torta'Range) of Integer;
```

```
begin
```

```
    Put_Line ("SERVER iniziato!");
```

```
    --INIZIALIZZAZIONI:
```

```
    for i in torta'Range loop
```

```
        sultavolo(i):=0;
```

```
    end loop;
```

```
    delay 2.0;
```

```
        -- continua..
```

```

loop
  select  -- deposito crostata:
    when  sultavolo(marmellata)+sultavolo(cioccolato)<MAX and
    sultavolo(marmellata) < MAX-1 =>
      accept deposito(marmellata) (ID: in clienteOP_ID ) do
        sultavolo(marmellata):=sultavolo(marmellata)+1;
      end;
  or      -- deposito cioccolato:
    when  sultavolo(marmellata)+sultavolo(cioccolato)<MAX and
    sultavolo(cioccolato) < MAX-1 and
    deposito(marmellata)'COUNT=0 =>
      accept deposito(cioccolato) (ID: in clienteOP_ID ) do
        sultavolo(cioccolato):=sultavolo(cioccolato)+1;
      end;
  -- CONTINUA..

```

```

or  -- prelievo family:
    when sultavolo(marmellata) >=1
    and sultavolo(cioccolato) >=1 =>
    accept prelievo(family) (ID: in clienteOC_ID ) do
    sultavolo(marmellata):=sultavolo(marmellata)-1;
    sultavolo(cioccolato):=sultavolo(cioccolato)-1;
    end;

or  -- prelievo marmellata:
    when sultavolo(marmellata) >=1 and prelievo(family) 'COUNT=0 =>
    accept prelievo(marm) (ID: in clienteOC_ID ) do
    sultavolo(marmellata):=sultavolo(marmellata)-1;
    end;

or  -- prelievo cioccolato
    when sultavolo(cioccolato) >=1 and prelievo(family) 'COUNT=0
    and prelievo(marm) 'COUNT=0=>
    accept prelievo(cioc) (ID: in clienteOC_ID ) do
    sultavolo(cioccolato):=sultavolo(cioccolato)-1;
    end;

    end select;
end loop;
end; -- fine task server

```

```
-- definizione task clienti:
```

```
task body clienteOP is
```

```
begin
```

```
    S. deposito(T) (ID) ;
```

```
end;
```

```
task body clienteOC is
```

```
begin
```

```
    S. prelievo(C) (ID) ;
```

```
end;
```



```

-- "main":
NewOP: acOP;
NewOC: acOC;

begin -- equivale al main
  for I in clienteOP_ID'Range
  loop -- ciclo creazione task OP
    NewOP := new clienteOP (I, cioccolato);
    NewOP := new clienteOP (I, marmellata);
  end loop;

  for I in clienteOC_ID'Range
  loop -- ciclo creazione task OC
    NewOC := new clienteOC (I, cioc);
    NewOC := new clienteOC (I, marm);
    NewOC := new clienteOC (I, family);
  end loop;
end torte; -- fine programma

```