Università degli Studi di Bologna
Facoltà di Ingegneria

Corso di

Reti di Calcolatori M

CORBA - Implementazione

Invocazione statica: seconda parte

Luca Foschini

Anno accademico 2014/2015

Agenda

- Esempio completo di client statico
- Esempio completo di server statico

Esempio completo di servizio in Java con invocazione statica: lato Client

Definizione dell'interfaccia CORBA

In generale non si definiscono singole interfacce, ma moduli (→ mappati in package Java) che racchiudono la definizione di uno o più oggetti CORBA (interface), ed eventualmente anche eccezioni e costanti

```
module nome_modulo {
        <dichiarazioni di costanti, tipi, eccezioni>
        ...
  interface nome_interfaccia1 [:<clausole_di_derivazione>]{
    <dichiarazioni di costanti, tipi, attributi, eccezioni>
        ...
        [<tipo_di_ritorno>] nome_op1(<parametri>)
        [raises <eccezione>] [<contesto>]
        ...
        [<tipo_di_ritorno>] nome_opN(<parametri>)
        [raises <eccezione>] [<contesto>]
        ...
  };
    ...
  interface nome_interfacciaN [:<clausole_di_derivazione>]{  ...  };
};
```

Non solo di input, ma anche di output, e input/output

Il modulo MessageApp: interfaccia IDL

```
/* message.idl */
module MessageApp {
   /*  Eccezione applicativa  */
   exception ErroreApplicativo {
   string codice_errore;
   };

   interface Message {
   /*  Separa un messaggio in due parti, all'invocazione:
    * msg -> messaggio
    * separatore -> separatore
    *
    * al ritorno:
    * inizio -> prima parte del messaggio
    * msg -> parte restante del messaggio
    */
   void splitMessage( inout string msg, out string inizio, in string
   separatore)
   raises (ErroreApplicativo);
   };
};
```

Definiamo un servizio per dividere una stringa, che possa generare anche eccezioni, e abbia, oltre a parametri di input anche parametri di output e input/output

NOTA: in Java il passaggio dei parametri avviene per copia del riferimento e NON è possibile avere parametri di in/out → necessità di supporto (Holder, vedi dopo…)

Generazione automatica di stub e classi di supporto in Java - lato client (1)

1. Definizione delle interfacce IDL degli oggetti:

   Es. **<IDL_file_name>.idl**

2. Compilazione delle interfacce => file generati con compilatore di interfaccia (idl di JacORB):

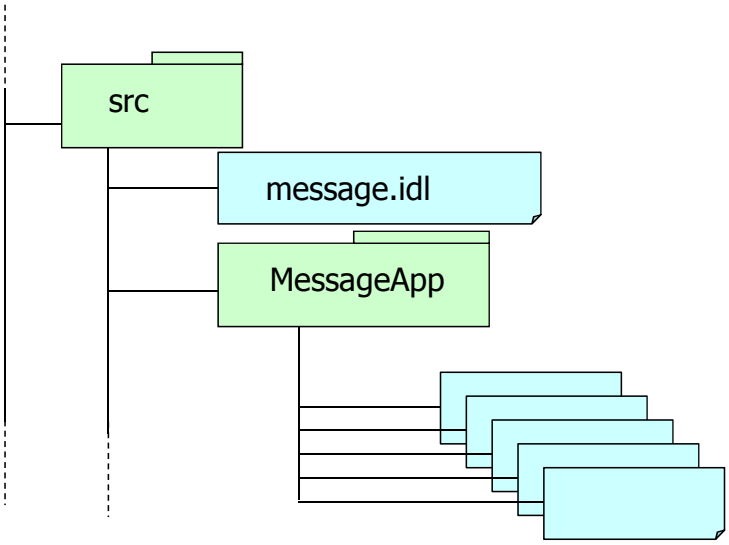
<interface_name>.java, <interface_name>Operations.java, <interface_name>Helper.java, <interface_name>Holder.java, _<interface_name>Stub.java

Traduzione dell' IDL CORBA in interfaccia Java

***Stub** dell'oggetto server*

Generazione automatica di stub e classi di supporto in Java - lato client (2)

```
>idl –noskel message.idl
```



Interface:
MessageOperations.java
Message.java
ErroreApplicativo.java

Helper:
ErroreApplicativoHelper.java
MessageHelper.java

Holder:
ErroreApplicativoHolder.java
MessageHolder.java

Stub: _MessageStub.java

Client: client.java

Generazione automatica di stub e classi di supporto in Java - lato client (3)

Signature Interface: <interface_name>.java
Estende l'interfaccia dell'Object CORBA (**org.omg.CORBA.Object**), e l'**Operations Interface** specifica dell'oggetto remoto

Operations Interface : <interface_name>Operations.java
Definisce l'interfaccia Java mappata dall'interfaccia IDL, con le **firme delle operazioni**

Helper : <interface_name>Helper.java
Fornisce le funzionalità ausiliarie necessarie per supportare la **conversione dei tipi** dal mondo CORBA allo specifico linguaggio (*narrow*, *read* e *write* di riferimenti/dati, …)

Generazione automatica di stub e classi di supporto in Java - lato client (4)

Holder: <interface_name>Holder.java
Contiene il riferimento all'oggetto che implementa la signature interface, e permette di realizzare parametri di in/out o out definite da IDL CORBA in Java. In pratica, realizzano dei **contenitori** di oggetti

Stub: _<interface_name>Stub.java
E' una classe che implementa la **signature interface** (Message), cioè l'object reference CORBA dell'oggetto remoto da invocare. In particolare, realizza tutti i metodi definiti dall'interfaccia CORBA facendo ***marshaling*** e ***unmarshaling*** degli argomenti e ***invocando i metodi remoti***

Signature Interface e Operations Interface

```
package MessageApp;

public interface Message
 extends MessageOperations,
 org.omg.CORBA.Object,
 org.omg.CORBA.portable.IDLEntity{}
```

Package generato dall'IDL compiler

Combina le operazioni del generico Object CORBA con i metodi specifici definiti nell'interfaccia MessageOperations

```
package MessageApp;

public interface MessageOperations
{
     void splitMessage(
        org.omg.CORBA.StringHolder msg,
        org.omg.CORBA.StringHolder inizio,
        java.lang.String separatore
    ) throws serverPack.MessageApp.ErroreApplicativo;
}
```

Nota: uso StringHolder

Signature Interface: CORBA Object

org.omg.CORBA.**Object**

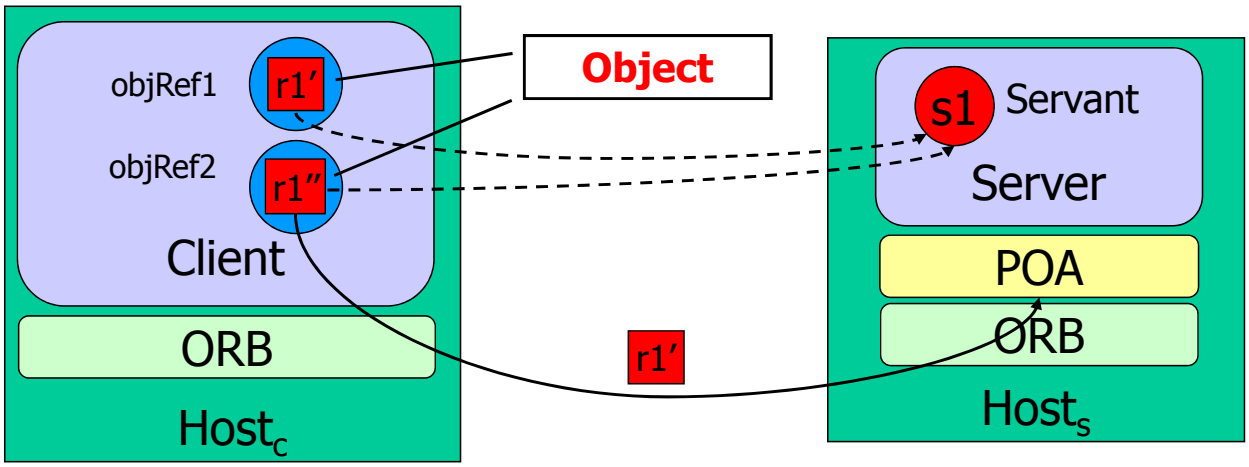
generico oggetto remoto CORBA, svolge un ruolo analogo all'interfaccia **`java.rmi.Remote`** in Java RMI

In particolare, definisce le ***operazioni di base***, ad esempio:

- **`is_a`:** verificare se l'oggetto remoto implementa l'interfaccia passata come stringa in ingresso
- **`is_equivalent`:** confronta l'equivalenza di due riferimenti

Attenzione!! Sono metodi remoti e in generale producono una invocazione remota

Signature Interface: is_equivalent



objRef2==objRef1 **Valutata localmente**

objRef2.is_equivalent(objRef1) **Chiamata remota**

Per **garantire interoperabilità**, si lascia comunque ampia libertà:

- Alcune implementazioni potrebbero ***NON*** implementare ***tutti i metodi***
- Semantica **`is_equivalent`**: se restituisce **true** i due riferimenti puntano allo stesso oggetto, se restituisce **false** significa che l'ORB non è stato capace di verificare l'uguaglianza (NON necessariamente oggetti diversi)

Le classi Holder in Java

- La classe Holder implementa un "contenitore di oggetti"; viene utilizzata per realizzare parametri di ingresso e di uscita in quei linguaggi ove **non** c'è **passaggio di in/out** (es. Java). Ad esempio nel metodo **splitMessage** appena visto:

  ```
  void splitMessage(inout string msg,…)
  ```

- Utilizziamo lo **`StringHolder`**, che non è altro che un **contenitore di una `String` di Java** → l'utilizzatore può accedere il valore contenuto nello StringHolder direttamente attraverso il campo pubblico **`value`**.

Le classi Helper in Java

- Punto di contatto fra mondo CORBA e mondo Java, generato automaticamente dal compilatore idl

- L'helper class viene utilizzata per "passare" dai tipi di dato definiti attraverso l'IDL CORBA ai tipi di dato presenti nello specifico linguaggio (es. Java)

- Offre diversi metodi, fra cui:
  - Gestione del tipo del riferimento CORBA, e.g., **`type`** e **`id`**
  - Narrowing dell'oggetto, cioè passare riferimento opaco CORBA di tipo **Object** allo **specifico oggetto** CORBA (definito dall'IDL), e usabile come oggetto tipato Java

Helper: TypeCode in Java

TypeCode è la **rappresentazione interna** mantenuta da Java di un **tipo di dato CORBA**; inoltre **`id()`** restituisce una stringa che identifica globalmente il tipo di dato

```
public static org.omg.CORBA.TypeCode type()
{
 return org.omg.CORBA.ORB.init().
create_interface_tc("IDL:MessageApp/Message:1.0",
          "Message");
}


public static String id()
{ return "IDL:MessageApp/Message:1.0"; }
```

Come in RPC è possibile distinguere versioni diverse dello stesso oggetto.

Helper: narrowing in JacORB (lato client)

```
public static MessageApp.Message narrow
                    (final org.omg.CORBA.Object obj)
{ if (obj == null) return null;
  try{ return (MessageApp.Message)obj; }
  catch (ClassCastException c){
    if (obj._is_a("IDL:MessageApp/Message:1.0"))
    { MessageApp._MessageStub stub;
      stub = new MessageApp._MessageStub();
      stub._set_delegate(
        ((org.omg.CORBA.portable.ObjectImpl)
      return stub;
    }
  }
  throw new org.omg.CORBA.BAD_PARAM("Narr
}
```

(1)

(2)

NOTA: uso di **_is_a**, metodo del generico Object CORBA

Siamo sul client → accedo a un oggetto remoto. In questo caso il **narrowing** include la ***creazione dello stub locale*** in due passi:
1. **controllo** del tipo atteso;
2. **creazione di uno stub** (***tipato Java***) per invocare i metodi dell'oggetto remoto.

Esempio: MessageHolder in Java

```
package serverPack.MessageApp;

public final class MessageHolder implements
    org.omg.CORBA.portable.Streamable {
  public MessageApp.Message value;
  public MessageHolder() {}

  public MessageHolder(serverPack.MessageApp.Message value)
  { this.value = value; }

  public void _read(org.omg.CORBA.portable.InputStream _stream)
  {value = MessageApp.MessageHelper.read(_stream);}

  public void _write(org.omg.CORBA.portable.OutputStream _stream)
  {MessageApp.MessageHelper.write(_stream, value);}

  public org.omg.CORBA.TypeCode _type()
  { return MessageApp.MessageHelper.type();}
}
```

Riferimento all'oggetto CORBA definito da noi

Lettura dell'oggetto remoto (riferimento) Message

Scrittura dell'oggetto remoto (riferimento) Message

Lo stub: _MessageStub

- Mantiene informazioni relative a tutte le interfacce implementate e viene generato automaticamente dal compilatore idl
- È una classe che implementa la **signature interface** (Message), cioè l'object reference CORBA dell'oggetto remoto da invocare. In particolare, realizza tutti i metodi definiti dall'interfaccia CORBA facendo ***marshaling*** e ***unmarshaling*** degli argomenti e ***invocando i metodi remoti***
- Se l'oggetto è presente nell'ambiente locale si tende a risolvere le invocazioni localmente (senza passare da tutta l'infrastruttura CORBA)

_MessageStub in JacORB: esecuzione operazione remota

```
public void splitMessage(org.omg.CORBA.StringHolder msg,
 org.omg.CORBA.StringHolder inizio, java.lang.String separatore)
   throws MessageApp.ErroreApplicativo
{
  while (true){
    if (!_is_local()) {
      org.omg.CORBA.portable.OutputStream _output = null;
      org.omg.CORBA.portable.InputStream _input = null;
      try {
        _output = _request("splitMessage", true);
        _output.write_string(msg.value);
        _output.write_string(separatore);
        _input = _invoke(_output);
        msg.value = _input.read_string();
        inizio.value = _input.read_string();
        return;
      }catch(
          org.omg.CORBA.portable.RemarshalException _exception)
      {continue;}
```

Marshaling

Unmarshaling

Oggetto remoto

Invocazione remota

_MessageStub in JacORB: esecuzione operazione locale

```
  catch(org.omg.CORBA.portable.ApplicationException _exception) {
    final java.lang.String _exception_id = _exception.getId();
    if (_exception_id.
      equals(MessageApp.ErroreApplicativoHelper.id())) {
       throw MessageApp.
       ErroreApplicativoHelper.read(_exception.getInputStream());
    }
    throw new org.omg.CORBA.UNKNOWN("Unexpected …" …);
  } finally { _releaseReply(_input); }
} else {
   org.omg.CORBA.portable.ServantObject _so = _servant_preinvoke
   ("splitMessage", _opsClass);
   if (_so == null) { continue; }
   MessageOperations _self = (MessageOperations) _so.servant;
   try {
     _self.splitMessage(msg, inizio, separatore);
     return;
   } finally { _servant_postinvoke(_so); }
} /*else*/ } /*while*/ } /* splitMessage*/
```

Oggetto locale

Riferimento alla classe che implementa il servant *(scelta specifica di JacORB)*

Invocazione locale

Client: narrowing e invocazione servizio

Il client, **scritto da noi** è identico a quello già visto: inizializza l'ORB, ottiene il riferimento all'oggetto CORBA, quindi effettua il narrowing e richiede i servizi all'oggetto remoto. Uso di **holder** per ottenere argomenti ***out***

```
   …
   // Narrowing
   Message message = MessageHelper.narrow(obj_mio);

   // Invocazione operazione remota
   // NOTA: uso di StringHoler per parametri di in/out
   StringHolder holder_corpo =new StringHolder("testa|corpo");
   StringHolder holder_testa = new StringHolder();
   message.splitMessage(holder_corpo, holder_testa, "|");
   System.out.println("Risultato: "+holder_testa.value+" "+
     holder_corpo.value);
  // Gestione eccezioni
  } catch (Exception ex) {
    System.out.println("Unexpected CORBA exception: " +ex);
    ex.printStackTrace(System.out);  }
 }    //main
} //client
```


Esempio completo di servizio in Java con invocazione statica: lato Server

Generazione automatica di skeleton e classi di supporto in Java - lato server (1)

1. Definizione delle interfacce IDL degli oggetti:

   Es. **<IDL_file_name>.idl**

2. Compilazione delle interfacce => file generati con compilatore di interfaccia (idl di JacORB):

<interface_name>.java, <interface_name>Operations.java, <interface_name>Helper.java, <interface_name>Holder.java, <interface_name>POA.java, <interface_name>POATie.java

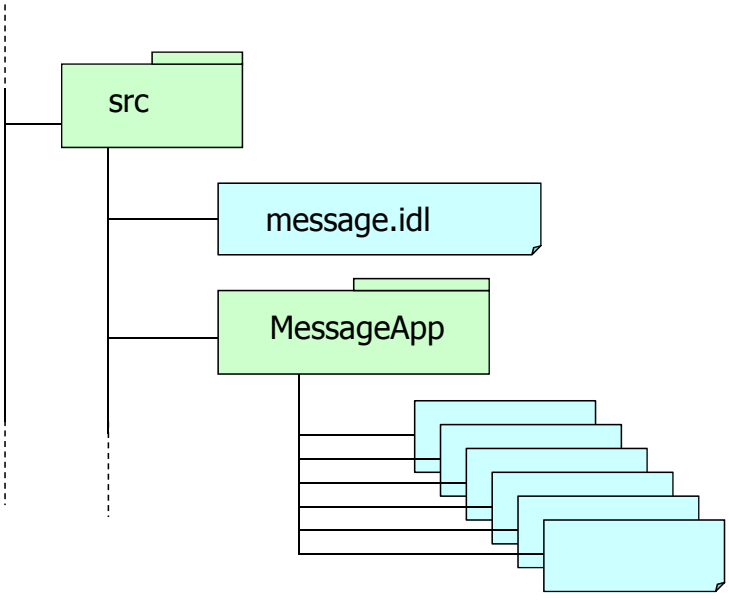
Traduzione dell' IDL CORBA in interfaccia Java

***Skeleton** per l'oggetto server*

Generazione automatica di skeleton e classi di supporto in Java - lato server (2)

Utilizziamo i tool di sviluppo di JacORB (idl), ecco l'invocazione dell'idl:

```
idl -nostub message.idl
```



Interface:
MessageOperations.java
Message.java
ErroreApplicativo.java

Helper:
ErroreApplicativoHelper.java
MessageHelper.java

Holder:
ErroreApplicativoHolder.java
MessageHolder.java

POA:
MessagePOA.java
MessagePOATie.java

Generazione automatica di skeleton e classi di supporto in Java - lato server (3)

Signature Interface: <interface_name>.java
Estende l'interfaccia dell'Object CORBA (**org.omg.CORBA.Object**), e l'**Operations Interface** specifica dell'oggetto remoto

Operations Interface : <interface_name>Operations.java
Definisce l'interfaccia Java mappata dall'interfaccia IDL, con le **firme delle operazioni**

Helper : <interface_name>Helper.java
Fornisce le funzionalità ausiliarie necessarie per supportare la **conversione dei tipi** nel mondo CORBA (funzione *narrow* per il casting) e dal mondo CORBA allo specifico linguaggio

Generazione automatica di skeleton e classi di supporto in Java - lato server (4)

Holder: <interface_name>Holder.java
Contiene il riferimento all'oggetto che implementa la signature interface, e mappa i parametri di in o out da IDL alla sintassi Java. In pratica realizzano dei **contenitori** di oggetti

Skeleton: <interface_name>POA.java oppure <interface_name>POATie.java
Rappresentano l'adattatore presente lato server che svolge diversi compiti:

- agisce come **skeleton** effettuando il **dispatching** delle richieste (upcall) agli oggetti registrati, inoltre svolge funzionalità di **registrazione** dell'oggetto, **attivazione** dei processi interni server e dell'oggetto stesso, ecc.

La prima classe l'abbiamo già utilizzata per realizzare il servant **per estensione**. E' però possibile realizzare il servant anche **per delega**, utilizzando l'altra classe (<interface_name>POATie), come vedremo tra poco…

Helper: narrowing in JacORB (lato server)

Siamo sul server → accedo a un oggetto locale. In questo caso il **narrowing** si riduce a ***un semplice casting***

```
public static MessageApp.Message narrow
  (org.omg.CORBA.Object obj)
{
if (obj == null) return null;
if (obj instanceof MessageApp.Message)
  return (MessageApp.Message)obj;
else throw new org.omg.CORBA.BAD_PARAM
  ("Narrow failed, not a MessageApp.Message");
}
```


Lo skeleton: MessagePOA

- Generato automaticamente dal compilatore idl
- **Non** realizza direttamente i metodi dichiarati nell'interfaccia dell'oggetto, pur implementandone l'interfaccia
  → la logica applicativa si trova nel servant, che è realizzabile:
  - Per estensione (già visto nel primo esempio)
  - Per delega
- Mantiene informazioni relative a tutte le interfacce implementate e riesce ad effettuare correttamente il dispatching delle chiamate remote
- Realizza diverse operazioni di supporto, fra cui:
  - Dispatching della chiamata remota sul servant locale → si vedano sia inizializzazione statica di `m_opsHash` sia il metodo `_invoke`
  - Marshaling e l'unmarshaling dei parametri di ingresso e uscita

MessagePOA in JacORB: tabella dei metodi

```
package serverPack.MessageApp;

abstract public class MessagePOA extends
 org.omg.PortableServer.Servant
 implements org.omg.CORBA.portable.InvokeHandler,
 MessageOperations {
 …

    static private final java.util.Hashtable m_opsHash =
      new java.util.Hashtable();
    static
    {
       m_opsHash.put("splitMessage", new java.lang.Integer(0));
    }
```

Registrazione metodi invocabili da remoto

MessagePOA: dispatching in JacORB

```
public org.omg.CORBA.portable.OutputStream _invoke(
   String method, org.omg.CORBA.portable.InputStream _input,
    org.omg.CORBA.portable.ResponseHandler handler)
       throws org.omg.CORBA.SystemException
{ org.omg.CORBA.portable.OutputStream _out = null;
  java.lang.Integer opsIndex =
    (java.lang.Integer)m_opsHash.get ( method );
  if ( null == opsIndex )
    throw new org.omg.CORBA.BAD_OPERATION(method +
       " not found");
  switch ( opsIndex.intValue() )
  {
```

Recupero l'operazione da invocare: scelta implementativa di JacORB

MessagePOA: esecuzione operazione in JacORB

```
case 0:  //splitMessage
    { try
      {
        org.omg.CORBA.StringHolder _arg0= new org.omg.CORBA.StringHolder();
        _arg0._read (_input);
        org.omg.CORBA.StringHolder _arg1= new org.omg.CORBA.StringHolder();
        java.lang.String _arg2=_input.read_string();
        _out = handler.createReply();
        splitMessage(_arg0,_arg1,_arg2);
        _out.write_string(_arg0.value);
        _out.write_string(_arg1.value);
      }
      catch(MessageApp.ErroreApplicativo _ex0)
      { _out = handler.createExceptionReply();
        MessageApp.ErroreApplicativoHelper.write(_out, _ex0);
      }
      break;
    } // case 0
  }  //switch
  return _out;
}  //invoke
```

Unmarshaling

Marshaling

Creazione risposta (Reply)

MessagePOA è una classe astratta, il metodo *slitMessage* invocato qui è implementato nelle sottoclassi

Uso di StringHolder **NON** String Java per parametri di out o in/out

Realizzazione servant in Java (per estensione)

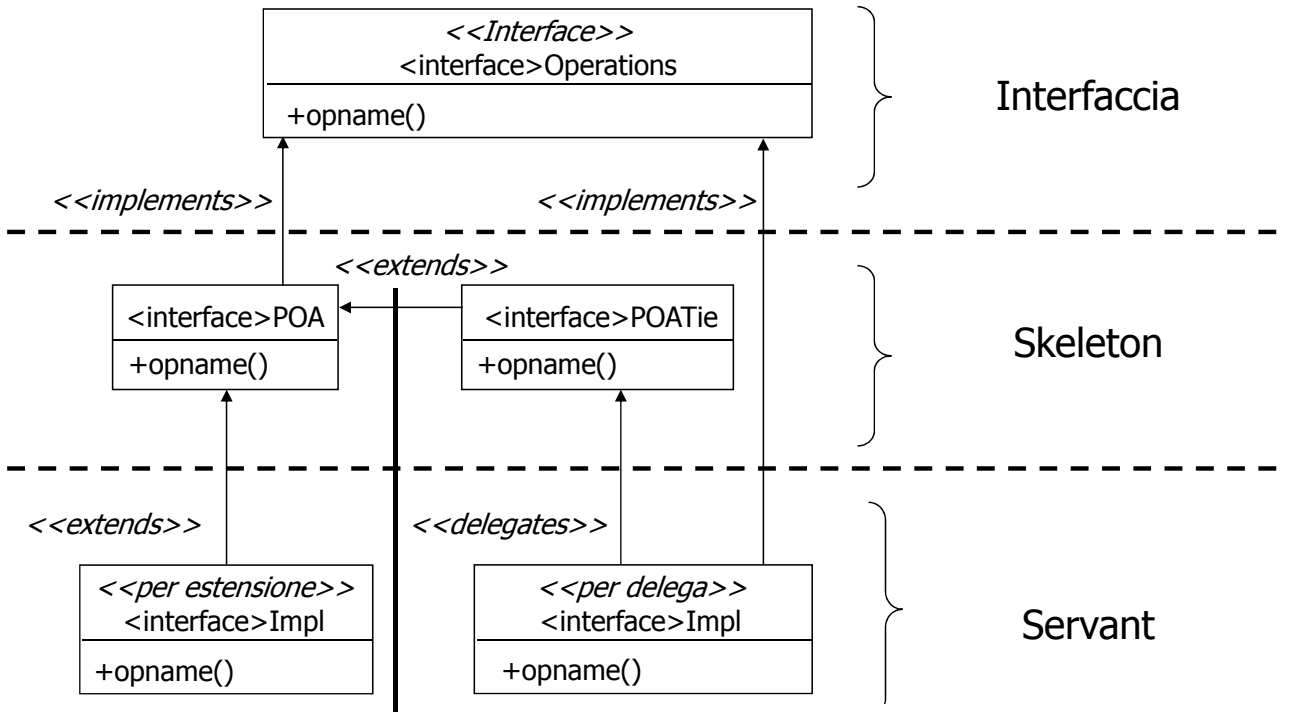
```
public class MessageImpl extends MessagePOA
{
 org.omg.PortableServer.POA m_poa = null;

 public void splitMessage(
   org.omg.CORBA.StringHolder      msg,
   org.omg.CORBA.StringHolder      inizio,
   java.lang.String                separatore )
      throws org.omg.CORBA.SystemException,
             serverPack.MessageApp.ErroreApplicativo
 {
  StringTokenizer tok =
     new StringTokenizer(msg.value, separatore);
   // inizio (out string)
   inizio.value = tok.nextToken();
   // msg (inout string)
   msg.value = tok.nextToken();
 }

}
```

Logica applicativa, unica parte scritta dal programmatore

Realizzazione servant in Java per delega



Realizzazione del servant per estensione (già vista nella scorsa esercitazione)

Realizzazione del servant per delega: il POATie realizza il metodo astratto di <interface>POA (opname) invocando il metodo realizzato dal delegato (<interface>Impl)

Realizzazione servant in Java per delega: MessagePOATie

- Estende la classe astratta POA realizzando i metodi dichiarati nell'interfaccia dell'oggetto CORBA **delegandone l'esecuzione** ad un oggetto Java che implementi l'Operations Interface: il **`_delegate`**. Nel nostro esempio:

```
public class MessagePOATie extends
  serverPack.MessageApp.MessagePOA {
    private serverPack.MessageApp.MessageOperations _delegate;
    …
    public void splitMessage(
      org.omg.CORBA.StringHolder msg,
      org.omg.CORBA.StringHolder inizio,
      java.lang.String separatore)
         throws serverPack.MessageApp.ErroreApplicativo
      { _delegate.splitMessage(msg, inizio, separatore); }
}
```

Invocazione dell'operazione sul delegato

Realizzazione servant in Java (per delega)

```
public class MessageImpl implements MessageOperations
{
  public void splitMessage(
    org.omg.CORBA.StringHolder msg,
    org.omg.CORBA.StringHolder inizio,
    java.lang.String separatore )
       throws org.omg.CORBA.SystemException,
              serverPack.MessageApp.ErroreApplicativo
  {
   StringTokenizer tok =
     new StringTokenizer(msg.value, separatore);
   // inizio (out string)
   inizio.value = tok.nextToken();
   // msg (inout string)
   msg.value = tok.nextToken();
  }

}
```

Realizza **unicamente** i metodi definiti nell'interfaccia MessageOperations. L'implementazione del metodo è identica a quella già vista.

Logica applicativa, unica parte scritta dal programmatore

Server: attivazione e stampa riferimento

Server: server.java

Crea e inizializza l'ORB, crea, configura e attiva il POA, crea un'istanza del servant, la registra e l'attiva sul POA, quindi stampa a video il riferimento all'oggetto remoto appena creato

Nel caso di ereditarietà:

```
MessageImpl servant = new MessageImpl();
obj = rootPOA.servant_to_reference(servant);
// Stampo l'Instance Object Reference (IOR) su standard output
System.out.println(orb.object_to_string(obj));
```

Nel caso di delega la parte in blu diventa:

```
MessagePOATie servant = new MessagePOATie(new MessageImpl());
```

Bibliografia

- **S.Russo, C.Savy, D.Cotroneo, A.Sergio, “Introduzione a CORBA”, Ed. McGrow-Hill (2002)**

- **F. Bolton, “Pure CORBA – A Code-Intensive Premium Reference”, Ed. SAMS (2002)**

- **R.Orfali, D.Harkey, *“Client/Server Programming with Java and CORBA, 2nd ed.”*, Ed. Wiley (1998)**

- **JacORB Programming Guide: `JACORB_HOME/doc/ProgrammingGuide/ProgrammingGuide.pdf`**