CORBA

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano

Outline

CORBA
Definizioni e Nozioni Generali

CORBA: **C**ommon **O**bject **R**equest **B**roker **A**rchitecture.

CORBA è una specifica per un middleware object-oriented per applicazioni distribuite.

CORBA è parte del modello OMA (Object Management Architecture) ed è frutto del lavoro del consorzio OMG (Object Management Group).

CORBA
Object Management Group (OMG).

Il consorzio **OMG**: **O**bject **M**anagement **G**roup (`www.omg.org`)

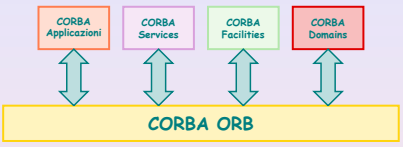
- è un'organizzazione no profit nata nel 1989;
- ha più di 800 membri, principalmente aziende;
- ha come scopo la standardizzazione e promozione di tecnologie e metodologie relative a sistemi software distribuiti orientati agli oggetti;

OMG non produce software ma specifiche per ottenere:

- interoperabilità;
- riusabilità; e
- portabilità

dei componenti software su sistemi eterogenei.

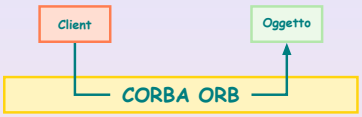
CORBA
L'Architettura OMA: Object Management Architecture.



La standardizzazione prevede:

- l'**ORB**: **O**bject **R**equest **B**roker
  - fornisce i meccanismi di comunicazione di base;
- i servizi (**CORBAservices**)
  - comprende i servizi agli oggetti (es. Naming, Notification, ...);
- i servizi di alto livello (**CORBAfacilities**)
  - definisce i servizi "orizzontali" alle applicazioni;
- i domini (**CORBAdomains**)
  - specifica servizi "verticali" cioè per domini applicativi specializzati.

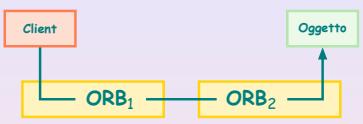
CORBA
L'ORB: Object Request Broker.



L'ORB fornisce una serie di servizi per l'invocazione di metodi su oggetti remoti

- fornisce un insieme di API per l'invocazione dei metodi remoti;
- l'ORB non è una entità separata dal client e dall'oggetto ma consiste di sottoprogrammi che vanno compilati e linkati al client e all'oggetto (stub e skeleton);
- l'ORB garantisce la trasparenza rispetto ai sistemi operativi, ai protocolli di rete, ai linguaggi di programmazione e alla locazione degli oggetti

CORBA
L'ORB: Object Request Broker.



La CORBA Interoperability Architecture prevede un modello per l'interoperabilità tra oggetti su ORB diversi

- formato universale per la rappresentazione dei riferimenti agli oggetti (**Interoperable Object Reference - IOR**);
- formato comune per la rappresentazione dei dati (**Common Data Representation - CDR**);
- protocollo per le interazioni fra ORB (**General Inter-ORB Protocol - GIOP**) che definisce il formato per le richieste e le risposte; e
- specifica per la realizzazione di GIOP su protocolli TCP/IP (**Internet Inter-ORB Protocol - IIOP**).

CORBA
IOR: Interoperable Object Reference

Un **Object Reference** è l'insieme delle informazioni necessarie per utilizzare un oggetto CORBA.

Repository ID

- indica l'interfaccia IDL implementata dall'oggetto;
- permette di individuare l'interfaccia nell'Interface Repository.

Endpoint Info

- contiene le informazioni necessarie all'ORB per stabilire la connessione con l'oggetto;
- con IIOP rappresenta la versione del protocollo, l'indirizzo IP e la porta TCP.

Object Key

- non è standard e dipende dall'ORB specifico.

CORBA
IOR: Esempio.

```
IOR:000000000000001349444c3a48656c6
c6f434f5242413a312e3000000000000100
0000000000006800010200000000a31302
e362e312e3236000fa900000021afabcb00
000002203b9bf47a0000000100000000000
0000000000040000000000a000000000000
0100000001000000200000000000100010
0000002050100010001002000010109000
0
00100010100
```

```
Interoperable Object Reference:
  Type ID: IDL:HelloCORBA:1.0
  Contains 1 profile.
  Profile 0-IIOP Profile:
    version: 1.2
    host: 10.6.1.26
    port: 4009
    Object Key:
ForeignId[key_string=%af%ab%cb%00
    %00%00%00%20;%9b%f4z%00%00%0
    0%01%00%00%00%00%00%00%00%00
    %00%00%00%04%00%00%00%00%0a]
    Code Sets Component: native char
codeset:ISO 8859_1 conversion_code_sets:
ISO UTF-8 ISO 646 (ASCII), native wchar
codeset:ISO UTF-16 conversion_code_sets:
ISO UCS-2 Level 1
```

versione "string" ⇨ printIOR ⇨ versione "esplicita"

CORBA
Modello ad Oggetti di CORBA.

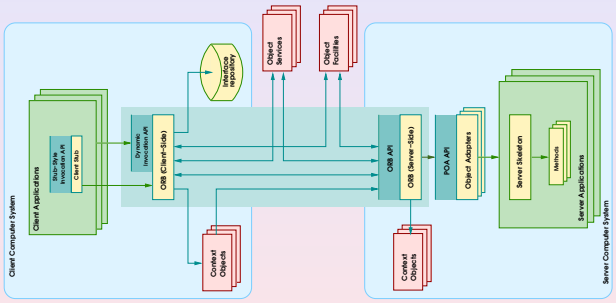
Incapsulamento

- chiara separazione fra l'interfaccia del modulo (visibile all'esterno) e la sua implementazione;
- l'interfaccia comprende operazioni e attributi ed è descritta col linguaggio IDL (Interface Description Language).

Oggetto

- è un'entità incapsulata che fornisce servizi ai client;
- possiede le proprietà (non sono variabili) e offre i servizi definiti da una interfaccia;
- un oggetto può richiedere servizi ad altri oggetti

CORBA
Architettura di CORBA.



- l'ORB è il sistema run-time che si occupa della comunicazione tra il client e l'oggetto;
- l'ORB fornisce alcuni servizi come la manipolazione dei riferimenti agli oggetti.

CORBA
Interfacce Statiche e Dinamiche (Lato Client).

Static Invocation Interface (SII).

- L'interfaccia IDL del servizio è nota in fase di implementazione del client;
- il client accede all'ORB attraverso lo static IDL stub che provvede alla costruzione della richiesta da passare all'ORB per l'invio.

Dynamic Invocation Interface (DII).

- L'interfaccia IDL del servente non è nota quando si implementa il client;
- si costruisce a run-time la richiesta all'oggetto remoto;
- richiede la conoscenza di meccanismi interni come la creazione di un oggetto di tipo Request.

CORBA
Interfacce Statiche e Dinamiche (Lato Server).

Static Skeleton Interface (SSI).

- L'interazione tra l'ORB e l'oggetto avviene attraverso uno stub skeleton generato a partire dall'interfaccia IDL;
- lo skeleton provvede all'estrazione dei parametri dalla richiesta (unmarshaling) e alla chiamata del servizio sull'oggetto servente.

Dynamic Skeleton Interface (DSI)

- L'interfaccia IDL del servente non è nota al momento della compilazione;
- si interpreta a run-time la richiesta all'oggetto remoto;
- richiede di programmare esplicitamente la gestione della Request ricevuta.

In teoria sono possibili tutte le combinazioni SII/DII lato client e SSI/DSI lato server

CORBA
Interfacce Repository (IR).

Fornisce a run-time informazioni sulle interfacce implementate dagli oggetti.

È un contenitore di oggetti che incapsulano le definizioni IDL.

Le interfacce possono essere utilizzate per

- costruzione dinamica delle richieste;
- compilazione dinamiche delle classi stub e skeleton.

L'ORB può utilizzare l'IR per

- effettuare controlli sulla validità di una Request;
- effettuare controlli sulla gerarchia tra interfacce.

CORBA
Portable Object Adapter (POA).

L'Object Adapter interfaccia l'oggetto servente (servant) con l'ORB

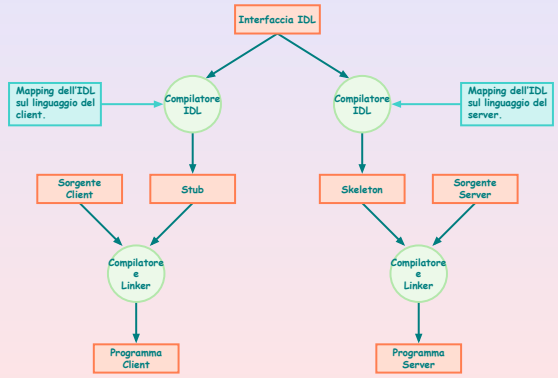
- assegna un riferimento univoco all'oggetto (Object Reference);
- ne gestisce il ciclo di vita;
- inoltra le richieste provenienti dall'ORB.

Fornisce al client l'immagine del servant come oggetto.

Il POA è stato introdotto a partire dalla versione CORBA 2.3 per rimpiazzare il Basic Object Adapter (BOA)

- la specifica del BOA era troppo vaga e portava ad implementazioni incompatibili fra loro.

CORBA
Sviluppo di un'Applicazione CORBA.



CORBA
Sviluppo di un'Applicazione CORBA: Esempio.

Esempio di interfaccia IDL.

```
interface HelloCORBA {
  attribute long counter; // proprietà
  string getHello(); // servizio offerto
};
```

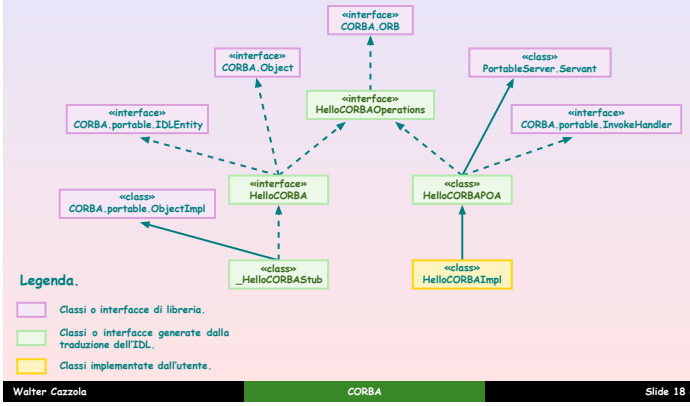
Per Java si compila l'interfaccia IDL col comando

```
idl2java HelloCORBA.idl
```

come risultato risultato della compilazione si hanno:

- `_HelloCORBAStub.java` – lo stub;
- `HelloCORBA.java` – interfaccia Java (vuota);
- `HelloCORBAOperations.java` – interfaccia con metodi;
- `HelloCORBAHolder.java` – classe per "contenere" oggetti di tipo `HelloCORBA`;
- `HelloCORBAHelper.java` – classe di utilità per implementare il client;
- `HelloCORBAPOA.java` – lo skeleton;

CORBA
Sviluppo di un'Applicazione CORBA: Gerarchia delle Classi.



CORBA
Sviluppo di un'Applicazione CORBA: Client.

Esempio di Client.

```
import org.omg.CORBA.*;
import java.io.*;

class HelloCORBAClient {
 public static void main(String args[]) {
   ORB orb = ORB.init(args,null);
   org.omg.CORBA.Object obj = orb.string_to_object(args[0]);
   HelloCORBA HCProxy = HelloCORBAHelper.narrow(obj);
   String msg = HCProxy.getHello();
   System.out.println("Messaggio ricevuto: "+msg);
 }
}
```

Inizializzazione dell'ORB.
Ottiene l'Object Reference.
Crea lo Stub.
Richiesta dei servizi.

L'IOR dell'oggetto remoto è fornito come argomento al programma.

CORBA
Sviluppo di un'Applicazione CORBA: Servant.

Implementazione del Servant.

```
import org.omg.PortableServer.*;

public class HelloCORBAImpl extends HelloCORBAPOA {
  private int cntr = 0;

  public int counter() {return cntr;}
  public void counter(int c) {cntr=c;}

  public String getHello() {
    cntr++;
    return "hello";
  }
}
```

Implementazione dell'attributo.
Implementazione dell'operazione.

CORBA
Sviluppo di un'Applicazione CORBA: Server.

Implementazione del Server.

```
import org.omg.PortableServer.*;

public class HelloCORBAServer {
 public static void main(String args[]) {
   try {
     org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

     HelloCORBAImpl HCServant = new HelloCORBAImpl();
     POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
     org.omg.CORBA.Object obj = rootPOA.servant_to_reference(HCServant);
     rootPOA.the_POAManager().activate();

     System.out.println(orb.object_to_string(obj));
     orb.run();
   } catch (Exception e) {
     e.printStackTrace();
   }
 }
}
```

Crea l'oggetto Servant.
Crea un'istanza del POA e registra il Servant.
Mette l'applicazione in attesa di richieste.

Interface Definition Language
Nozioni Generali.

Il linguaggio IDL è uno standard de jure (ISO 14650 e ITU X.920) per la specifica di interfacce

- i servizi sono specificati indipendentemente dalla loro implementazione
- viene disaccoppiato lo sviluppo del modulo servente dallo sviluppo dei client
- si possono usare linguaggi differenti per sviluppare i servant e i client

IDL è un linguaggio dichiarativo

- ha un insieme di tipi predefiniti
- si possono definire nuovi tipi di dato
- non è un linguaggio di programmazione!

Interface Definition Language
Caratteristiche di IDL.

Lo standard CORBA specifica le modalità di traduzione da IDL ai linguaggi di programmazione più diffusi.

Il linguaggio IDL consente di:

- definire le interfacce degli oggetti:
  - operazioni;
  - attributi (dotati di tipo eventualmente definito dall'utente);
- raggruppare le interfacce in moduli;
- definire tipi di utente e costanti;
- definire le eccezioni che le operazioni possono generare;
- definire gerarchie di interfacce con un meccanismo di ereditarietà.

Interface Definition Language
Attributi in IDL.

Un attributo è caratterizzato da:

- un nome;
- un tipo (predefinito dal linguaggio o definito dall'utente);
- una modalità di accesso (lettura/scrittura o a sola lettura **readonly**).

A ciascuna proprietà corrispondono due operazioni per leggere (get) o assegnare (set) il valore:

- se l'attributo è **readonly** esiste solo il metodo di lettura;
- la proprietà non è necessariamente implementata con una variabile dal servant.

```
interface TwoAttributes {
  readonly attribute long counter;
  attribute string message;
};
```

Interface Definition Language
Operazioni in IDL.

Per ciascuna operazione si definiscono:
- il nome;
- il tipo del valore di ritorno;
- i parametri dell'operazione.

Per i parametri occorre specificare la direzione dello scambio
- **in** – il parametro è trasmesso dal client all'oggetto;
- **out** – il parametro è trasmesso dall'oggetto al client;
- **inout** – il parametro è trasmesso in entrambe le direzioni.

Si può specificare la tipologia della chiamata:
- sincrona – il client si blocca fino al completamento della chiamata;
- **oneway** – individua un'operazione asincrona e il client non si blocca;

Se si descrive un'operazione che può essere chiamata asincronamente questa deve essere **void**, non può avere parametri **out** o **inout** e non può generare eccezioni.

Interface Definition Language
Esempio di Interfaccia con Operazioni in IDL.

```
interface FourOperations {
  string printSum(in long n1, in long n2);
  boolean doSum(in long n1, in long n2, out long sum);
  void accSum(inout long n1, in long n2);
  oneway void setMessage(in string message);
};
```

In Java i parametri di tipo **out** e **inout** sono trasformati in classi **Holder**
- Ad es. **inout long** diventa **org.omg.CORBA.IntHolder**.

La classe **Holder** concede di modificare il valore del parametro
- la variabile non può variare ma il contenuto dell'oggetto sì.

Interface Definition Language
Moduli.

Un modulo permette di raggruppare definizioni di interfacce, costanti, tipi di dato, eccezioni in una unità logica
- permette di evitare conflitti fra i nomi;
- si possono annidare le definizioni di modulo;
- il nome di un elemento interno **E** ad un modulo **M** diviene **M::E**

```
module FirstModule {
  interface MyInterface {
    attribute long mycounter;
  };
};
module SecondModule {
  interface MyInterface {
    attribute long mycounter;
  };
};
```

In Java i moduli dell'esempio verranno tradotti in due package distinti.

Interface Definition Language
Tipi di Dato.

Gli oggetti CORBA possono scambiarsi solo tipi di dati definiti in IDL.

Tipi primitivi.
- Lo standard definisce la corrispondenza con i tipi di dato dei linguaggi più diffusi
- **short**, **long**, **long long** (conversioni **unsigned**);
- **float**, **double**, **fixed<x,y>**, **char**, **wchar**, **string**, **wstring**, **boolean**, **octet**;
- è previsto il tipo **any** che corrisponde ad un tipo arbitrario da determinare in fase di esecuzione
  - in Java è tradotto come oggetto **org.omg.CORBA.Any**.
- Per ogni tipo è prevista la dimensione ma non è detto che sia rispettato l'intervallo dei valori
  - Ad esempio in Java non esistono gli interi **unsigned**.

Interface Definition Language
Tipi di Dato (Segue).

Tipi definiti dall'Utente.

Dichiarazioni **typedef**
- è possibile definire il nome di un tipo come sinonimo di uno preesistente

```
typedef string data;
```

Tipi Enumerativi
- definisce il nome di un tipo e per enumerazione i possibili valori;
- l'ordine con cui i valori compaiono definisce la relazione d'ordine

```
enum Valore {Uno,Due,Tre};
```

Tipi Strutturati
- definisce una struttura dati con campi

```
struct TipoStrutturato {string nome; long quantita};
```


Interface Definition Language
Tipi di Dato (Segue).

Array e Sequenze.

Array
- definisce liste di elementi dello stesso tipo di dimensione fissa

```
typedef string args[10];
```

Sequenze
- una sequenza ha dimensione variabile e la sua lunghezza deve poter essere determinata a run-time;
- si può definire la lunghezza massima di una sequenza;
- si possono definire sequenze di sequenze;

```
typedef sequence <long> seqNonLimitata;

typedef sequence <long,10> seqLimitata;
```


Interface Definition Language
Tipi di Dato e Java.

Per ogni tipo **T** esiste una classe **THolder** che è utilizzata come contenitore per gli oggetti di tipo **T**
- per i tipi primitivi le classi **Holder** sono predefinite;
- per i tipi utenti le classi **Holder** sono generate dal compilatore IDL;
- la classe **Holder** è usata se il tipo compare come parametro **out** o **inout**.

Per ogni tipo **T** definito dall'utente viene generata una classe **THelper** necessaria per le operazioni di marshaling/unmarshaling dei parametri.

I tipi enumerativi ed i tipi strutturati sono tradotti in una classe.

Una sequenza di elementi di tipo **T** è tradotta in un array Java di elementi di tipo **T**.

Interface Definition Language
Costanti.

È possibile definire costanti all'interno di un modulo. di un'interfaccia o anche a livello globale

```
const long C1 = 10;
```

In Java una costante è tradotta come:
- un membro di tipo **public static final** se è definita dentro una interfaccia;
- una interfaccia Java con lo stesso nome della costante e col valore nel membro value di tipo **public static final** se non è definita all'interno di una interfaccia.

Interface Definition Language
Eccezioni.

Le eccezioni permettono di segnalare condizioni anomale verificatesi nell'esecuzione di un'operazione

- occorre definire tutte le eccezioni;
- si deve dichiarare quali eccezioni possono essere generate da un'operazione.

```
interface Operazioni {
 exception DivisionePerZero {
   string msg;
   long val;
 };
 long div(in long n1,in long n2) raises (DivisionePerZero);
};
```

- **exception** permette la definizione di nuove eccezioni;
- **raises** permette di specificare le eccezioni sollevate da un'operazione.

Interface Definition Language
Ereditarietà tra Interfacce.

È possibile definire una gerarchia di interfacce con un meccanismo di ereditarietà singola o multipla

- un'interfaccia eredita tipi, attributi e operazioni delle interfacce base;
- un client che è progettato per interagire con un oggetto che implementa l'interfaccia base può interagire senza problemi con un oggetto che implementa l'interfaccia derivata ma non viceversa.

```
interface IF : IF1, IF2 {
  string operazioneDiIF(in long i);
};
```


Interface Definition Language
Valuetype.

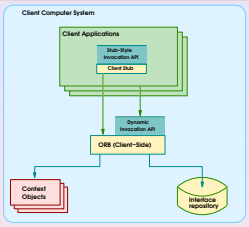
Previsto a partire dalla versione 2.3 dello standard.

Permette di definire oggetti dotati di stato e operazioni che possono essere passati come parametri delle operazioni.

- il passaggio avviene per valore;
- può essere passato come parametro **in**, **out**, **inout** o può essere il valore di ritorno di un'operazione.

```
valuetype Contatore {
  private long cntr; // variabile

  attribute string nome; // attributi e operazioni
  string stampa();

  factory create(in string s, in long n); // costruttore
};
```


Programmazione in CORBA
Lato Client.



Il tramite dell'invocazione è un oggetto `org.omg.CORBA.Request`

- l'interfaccia statica (SII) o dinamica (DII) si occupa di creare l'oggetto **Request**;
  - nel caso della SII i meccanismi di creazione e gestione della **Request** sono mascherati nello stub (Proxy);
- il nucleo dell'ORB (ORB Core) è responsabile dell'invio della richiesta verso l'oggetto remoto.

Programmazione in CORBA
Lato Client: Static Invocation Interface (SII).

Modello di programmazione SII

- inizializzazione della libreria
  - metodi statici `init()` della classe `org.omg.CORBA.ORB`
  - si ottiene il riferimento all'ORB che registra il cliente
- acquisizione del riferimento IOR all'oggetto remoto
  - si può utilizzare la rappresentazione dell'IOR o il servizio di Naming o Trading
  - si ottiene un riferimento ad un `org.omg.CORBA.Object`
- si instanzia un oggetto della classe stub (**proxy**)
  - si ottiene col metodo `narrow()` della classe `Helper` associata allo stub, applicato all'`Object` che rappresenta l'oggetto remoto
- si invocano i servizi dell'oggetto remoto utilizzando lo stub

Programmazione in CORBA
SII: Passaggio dei Parametri nell'Invocazione.

Parametro di tipo semplice.

Si consideri la seguente interfaccia IDL con un solo servizio che ha un solo argomento di tipo semplice.

```
interface Servizio1 {
  void addACC(in long i);
};
```

Compilando l'IDL si crea lo stub, l'interfaccia e un helper.

```
org.omg.CORBA.Object obj=orb.string_to_object(<IOR>);
Servizio1 S1Proxy = Servizio1Helper.narrow(obj);
    ...
int val = 2;
S1Proxy.addACC(val);
```

Uso dell'oggetto remoto (tramite lo stub locale).

Questo codice mostra come si richiede il servizio.

Collegamento con l'oggetto remoto tramite l'helper.

Programmazione in CORBA
SII: Passaggio dei Parametri nell'Invocazione (Segue).

Parametro inout di tipo semplice.

Si consideri la seguente interfaccia IDL con un solo servizio che ha un solo argomento di tipo semplice passato come **inout**.

```
interface Servizio2 {
  void addAndGet(inout long i);
};
```

Il passaggio dei parametri è fatto tramite un `Holder`, predefinito perché il parametro è un tipo semplice.

```
org.omg.CORBA.Object obj=orb.string_to_object(<IOR>);
Servizio2 S2Proxy = Servizio2Helper.narrow(obj);
   ...
val = 2;
IntHolder ih = new IntHolder(val);
S2Proxy.addAndGet(ih);
System.out.println("call addAndGet("+val+")="+ih.value);
```

Si passa il parametro inout utilizzando la classe Holder relativa (IntHolder).

Il valore modificato è accessibile nel campo value dell'Holder.

Programmazione in CORBA
SII: Passaggio dei Parametri nell'Invocazione (Segue).

Parametro inout di un tipo user-defined.

La seguente interfaccia IDL con un solo servizio che ha un solo argomento di tipo definito dall'utente passato come **inout**.

```
struct ComplexNr {
  float re; float im;
  factory create(in float r, in float i);
};
interface Servizio3 {
  void coniugato(inout ComplexNr cx);
};
```

In questo caso, la classe `Holder` per il tipo `ComplexNr` è generata automaticamente a partire dall'IDL.

```
ComplexNr cx = new ComplexNr(2,1);
ComplexNrHolder cxh = new ComplexNrHolder(cx);

S3Proxy.coniugato(cxh);
System.out.println("call coniugato(Re: "+cx.re+";Im: "+cx.im+")= Re: "+
                                        cxh.value.re+";Im: "+cxh.value.im);
```

Programmazione in CORBA

SII: Valori di Ritorno da un'Invocazione ed Eccezioni.

Valore di ritorno ed eccezioni definite dall'utente.

Si consideri la seguente interfaccia IDL con un solo servizio che ritorna un valore di tipo user-defined e solleva un'eccezione.

```
exception DivisionePerZero {
  string msg;
};
interface Servizio4 {
  ComplexNr reciproco(in ComplexNr cx) raises (DivisionePerZero);
};
```

Il metodo produce direttamente un oggetto del tipo definito dall'utente.

```
try {
  ComplexNr cxr = S4Proxy.reciproco(cx);
  System.out.println("call reciproco(Re: "+cx.re+
                     ";Im: "+cx.im+")= Re: "+cxr.re+";Im: "+cxr.im);
} catch (DivisionePerZero ex) {
  System.out.println("Divisione x Zero in Reciproco(Re: "+cx.re+";Im: "+cx.im+")");
}
```

Si gestisce l'eccezione utente con un blocco try-catch.

Programmazione in CORBA

SII: Valori di Ritorno da un'Invocazione ed Eccezioni (Segue).

Valore di ritorno di tipo any.

Si consideri la seguente interfaccia IDL con un solo servizio che ritorna un valore di tipo any.

```
interface Servizio5 {
  any qualsiasi();
};
```

Il valore di ritorno è un oggetto di tipo `org.omg.CORBA.Any`.

Si verifica il tipo del valore memorizzato nell'oggetto `Any`.

```
org.omg.CORBA.Any objRET = S5Proxy.qualsiasi();

if (objRET.type().kind().value()==TCKind._tk_string) {
  String s = objRET.extract_string();
  System.out.println("Call qualsiasi(): "+s);
}
```

`Any` ha un insieme di metodi per estrarre/inserire oggetti dei tipi predefiniti.

Programmazione in CORBA

Classi di Libreria: CORBA.Object.

org.omg.CORBA.Object

Rappresenta un oggetto remoto o locale definito con riferimento IOR

- se l'oggetto è remoto il riferimento avviene attraverso l'oggetto stub.

Ha metodi per:

- creazione di oggetti richiesta `Request` che si riferiscono implicitamente all'oggetto remoto e ad uno specifico metodo indicato come parametro (`_request()` e `_create_request()`);
- gestione dei riferimenti remoti (`_duplicate()`, `_is_equivalent()`, `_release()`, `_non_existent()`);
- introspezione dell'interfaccia degli oggetti remoti (`_get_interface_def()`, `_is_a(String repositoryIdentifier)`).

Programmazione in CORBA

Classi di Libreria: CORBA.Request.

org.omg.CORBA.Request

Rappresenta le informazioni per l'invocazione di un metodo remoto.

Costruzione

- si istanzia una `Request` attraverso i metodi di un `CORBA.Object`

```
Request rq = objRef._request("nomeMetodo");
```

- si specificano i parametri (sono una `NVList` - lista di oggetti `NamedValue`)

```
Any p1 = rq.add_in_arg()
p1.insert_long(2)
```

Sono definiti metodi analoghi per i parametri `inout` e `out` oltre a quelli per specificare anche il nome (es. `add_named_in_arg(String name)`).

Programmazione in CORBA

Classi di Libreria: CORBA.Request (Segue).

org.omg.CORBA.Request

Inoltro e recupero dei risultati.

La richiesta viene trasferita all'ORB core attraverso uno dei seguenti metodi:

- `invoke()`
  invocazione sincrona bloccante in attesa di una risposta.
  - La risposta si ottiene nel formato `Any` col metodo `return_value()`.
- `send_deferred()`
  invocazione asincrona non bloccante in attesa di risposta.
  - La disponibilità della risposta può essere verificata con un polling utilizzando il metodo `poll_response()`.
- `send_oneway()`
  invocazione asincrona senza risposta.

Programmazione in CORBA

Classi di Libreria: CORBA.portable.*.

Classi per definire stub portabili per l'interfaccia SII

- `org.omg.CORBA.portable.ObjectImpl`;
- `org.omg.CORBA.portable.OutputStream`;
- `org.omg.CORBA.portable.InputStream`;
- la richiesta è costruita come un `portable.Ouputstream` in cui si scrivono i parametri
  - sono definiti i metodi `write_T(T val)` per i tipi primitivi;
- il risultato dell'invocazione è un `portable.InputStream` da cui si possono leggere i risultati
  - sono definiti i metodi `T read_T()` per i tipi primitivi;
- per la programmazione SII le operazioni `read` e `write` sono incapsulate nelle classi `Helper`.

Programmazione in CORBA

Lato Client: Dynamic Invocation Interface (DII).

La Dynamic Invocation Interface permette di invocare metodi remoti senza avere a disposizione lo stub

- si crea esplicitamente la `Request`
- occorre conoscere la firma del metodo
  - si ottiene la firma di un metodo interrogando il servizio `InterfaceRepository` utilizzando l'introspezione sull'oggetto remoto
- si devono specificare e associare alla richiesta
  - i parametri della funzione;
  - il tipo di valore di ritorno; e
  - le eventuali eccezioni utente.

Programmazione in CORBA

DII: Passaggio dei Parametri nell'Invocazione.

Parametro inout di tipo user-defined.

Si consideri un'interfaccia IDL con un solo servizio che ha un solo argomento di tipo definito dall'utente e passato come `inout`.

```
interface Servizio3 {
  void coniugato(inout ComplexNr cx);
};
```

Crea la richiesta per l'oggetto `objRef` e il metodo `coniugato()`.

```
Request request = objRef._request("coniugato");

Any acx = request.add_inout_arg();
ComplexNr cx = new ComplexNr(2,1);
ComplexNrHelper.insert(acx,cx);

request.invoke();

ComplexNr cx1 = ComplexNrHelper.extract(acx);
OUT.println("call coniugato(Re: " + cx.re + ";Im: " + cx.im +
            ")= Re: " + cx1.re + ";Im: " + cx1.im);
```

Inserisce il parametro del tipo definito dall'utente con modalità `inout`.

Invoca il metodo.

Estrae il valore del parametro `inout` modificato dalla chiamata.

Questo codice mostra la richiesta tramite DII.

Programmazione in CORBA

DII: Valori di Ritorno da un'Invocazione ed Eccezioni.

Valore di ritorno ed eccezioni user-defined.

```
request = objRef._request("reciproco");

acx = request.add_in_arg();
ComplexNrHelper.insert(acx,cx);

request.set_return_type(ComplexNrHelper.type());
request.exceptions().add(DivisionePerZeroHelper.type());

request.invoke();

Exception e = request.env().exception();
if (e==null) {
  ComplexNr ris = ComplexNrHelper.extract(request.return_value());
    ...
} else {e.printStackTrace();}
```

Predispone il tipo del valore di ritorno.

Predispone le eccezioni utente.

Verifica la presenza di eccezioni.

Estrae il Risultato.

Programmazione in CORBA

Interface Repository.

Permette di ottenere una descrizione delle interfacce dinamicamente.

- Si deve attivare il servizio `InterfaceRepository`.
- Il servizio archivia definizioni IDL e le mette a disposizione attraverso specifici oggetti del package `org.omg.CORBA`.

`ModuleDef`	`InterfaceDef`	`AttributeDef`
`OperationDef`	`ExceceptionDef`	`SequenceDef`
`UnionDef`	`StructDef`	`EnumDef`
`AliasDef`	`ArrayDef`	`ConstantDef`

Programmazione in CORBA

Classi di Libreria: IRObject.

`org.omg.CORBA.IRObject`

Rappresenta la classe base di tutti gli oggetti registrati nell'interface repository.

- Ad ogni oggetto è associato il suo tipo (`def_kind()`).
- Gli oggetti possono implementare anche le interfacce:
  - `Container`
    contengono altre definizioni accessibili col metodo `contents()`;
  - `Contained`.
- Utilizzando i metodi offerti da queste interfacce e da quelle specifiche degli oggetti IR dei tipi è possibile navigare nel `Repository`.

Programmazione in CORBA

Esempio di Uso dell'Interface Repository.

```
org.omg.CORBA.Object objRep=null;
try {
  objRep = orb.resolve_initial_references("InterfaceRepository");
} catch (org.omg.CORBA.ORBPackage.InvalidName e) {
  e.printStackTrace();
}
org.omg.CORBA.Repository rep = org.omg.CORBA.RepositoryHelper.narrow(objRep);
org.omg.CORBA.Contained dsc[] = rep.contents(DefinitionKind.dk_all,true);

switch (dsc.def_kind().value()) {
 case DefinitionKind._dk_Interface:
    contained = ContainerHelper.narrow(dsc).contents(DefinitionKind.dk_all,true);
      ...
    break;
 case DefinitionKind._dk_Operation:
    OperationDef opdef = OperationDefHelper.narrow(dsc);
    ParameterDescription pardsc[] = opdef.params();
    for (int j=0;j<pardsc.length;j++) {
      ParameterMode pm = pardsc[j].mode;
      if (pm.value() == ParameterMode._PARAM_IN) {...}
        ...
    }
```

Ottiene il riferimento all'IR utilizzando il Naming Service.

Ottiene il contenuto della radice del repository (qualsiasi tipo di oggetto).

Determina il tipo di oggetto `Contained` che si sta analizzando.

`InterfaceDef` è un Container.

L'oggetto `Contained` è di tipo `OperationDef`.

Si usano i metodi specifici di `OperationDef` per determinare le caratteristiche dell'operazione.

Programmazione in CORBA

Il Servizio di Naming Service e lo Spazio dei Nomi.

Naming Service.

Gestisce l'associazione tra nomi simbolici e riferimenti remoti

- I server usano il Name Service per registrare i riferimenti agli oggetti CORBA e per associare ad essi un nome simbolico
- I client usano il servizio per ottenere il riferimento all'oggetto remoto a partire dal nome simbolico

Struttura dello Spazio dei Nomi.

Lo spazio dei nomi è organizzato in una gerarchia di oggetti `NamingContext`

- la struttura è analoga a quella di un filesystem;
- i nodi sono elementi di tipo `NamingContext`;

Programmazione in CORBA

Accesso al Name Service.

Il Name Service è un servizio distribuito.

- L'accesso alla radice (`root NamingContext`) avviene con una risoluzione di un servizio remoto.
- I client possono ottenere il riferimento al `NamingContext` radice con:

```
ncobj = orb.resolve_initial_references("NameService")
NamingContext rootNC = NamingContextHelper.narrow(ncobj)
```

- Nell'applicazione client si deve definire il nome di default del `NamingContext` radice (`-DSVCnameroot=<name>`).

Programmazione in CORBA

Classi di Libreria: NameComponent.

`org.omg.Cosnaming.NameComponent`

- Un `NamingContext` o un oggetto sono riferiti da un `Name` che è una sequenza di `NameComponent`.
- Ciascuna `NameComponent` contiene due attributi stringa:
  - `id` rappresenta il nome dell'entità;
  - `kind` è descrizione opzionale per il nome;

```
module CosNaming
  typedef string Istring;
  struct NameComponent {
    Istring id;
    Istring kind;
  };
  typedef sequence<NameComponent> Name;
};
```


Programmazione in CORBA

Classi di Libreria: NamingContext.

`org.omg.Cosnaming.NamingContext`

Rappresenta un `NamingContext` con le operazioni per:

- registrare un riferimento

```
bind(NameComponent[] n, Object o)
rebind(NameComponent[] n, Object o)
```

- risoluzione dei riferimenti

```
resolve(NameComponent[] n)
```

- aggiunta, modifica, eliminazione di nodi

```
bind_new_context(NameComponent[] n)
rebind_context(NameComponent[] n,NamingContext nc)
unbind(NameComponent[] n)
destroy()
```

- navigazione tra i rami

Programmazione in CORBA

Classi di Libreria: NamingContextExt.

org.omg.Cosnaming.NamingContextExt

Rappresenta un NamingContext esteso per gestire nomi rappresentati con stringhe invece che come sequenze di NameComponent

Contesto1/Contesto2/Contesto3.descrizione/NomeOggetto

Per risolvere un nome sotto forma di stringa si può usare

```
resolve_str("nome_Servizio")
```


Programmazione in CORBA

Registrazione di un Servizio.

```
org.omg.CORBA.Object nsObj = orb.resolve_initial_references("NameService");
NamingContext namesvc = NamingContextHelper.narrow(nsObj);

   ... creazione oggetto servant obj ....

NameComponent[] name = new NameComponent[2];
NameComponent[] esDir = new NameComponent[1];
name[0] = new NameComponent("Servizi","");
esDir[0] = name[0];
name[1] = new NameComponent("Esempio","");

try {
  namesvc.bind_new_context(esDir);
  namesvc.bind(name,obj);
} catch(org.omg.CORBA.UserException ue) {
  ue.printStackTrace();
  System.exit(-1);
}
```

Ottiene il riferimento al NamingContext radice.

Crea il nome "Servizi/Esempio" per registrare il servizio.

Aggiunge il NameContext "Servizi" alla radice.

Registra il servizio.

Programmazione in CORBA

Risoluzione di un Nome.

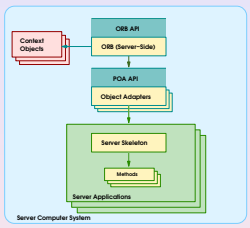
```
try {
  org.omg.CORBA.Object nsObj = orb.resolve_initial_references("NameService");
  NamingContextExt namesvc = NamingContextExtHelper.narrow(nsObj);

  obj = namesvc.resolve_str("Servizi/Esempio");
} catch (Exception e) {
  e.printStackTrace();
  System.exit(-1);
}
```

Ottiene il riferimento al NamingContext radice.

Risolve il nome utilizzando l'interfaccia estesa che gestisce i nomi definiti con stringhe.

Programmazione in CORBA

Lato Server.



Il servant è il responsabile dell'implementazione delle operazioni specificate nell'interfaccia IDL

- SSI – le classi skeleton sono ottenute con la compilazione dell'IDL.
- DSI – si devono usare esplicitamente gli oggetti definiti dalla piattaforma per il recupero, la decodifica e l'inoltro al servant delle richieste.

Programmazione in CORBA

ORB, POA e Servant.

L'**Object Adapter** gestisce:

- l'associazione fra l'oggetto servente e l'oggetto CORBA;
- lo smistamento delle richieste fra gli oggetti servant gestiti.

L'ORB riceve le richieste e le inoltra all'Object Adapter opportuno.
Lo skeleton si occupa dell' unmarshaling dei parametri, dell'up-calling del metodo del servant e del marshaling del valore di ritorno.

Programmazione in CORBA

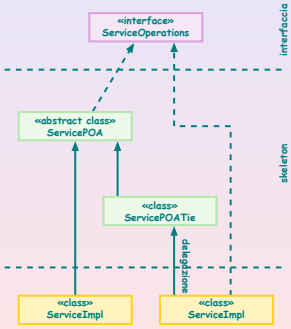
Realizzazione del Servant.

La classe servente si associa allo skeleton:

- per ereditarietà;
- per delega (tie approach).

L'uso del metodo per delega è utile in Java che non supporta l'ereditarietà multipla

- la classe servant non deve estendere la classe skeleton.



Programmazione in CORBA

Realizzazione del Servant (Segue).

Si consideri la seguente interfaccia:

```
interface ServizioA {
  double getSin(in double x);
}
```

Se la classe Servant estende la classe skeleton, si ha:

```
public class ServizioAImpl extends ServizioAPOA {
  public double getSin(double x) {
    return Math.sin(x);
  }
}
```

Invece se la classe servant implementa l'interfaccia, si ha:

```
public class ServizioAImpl implements ServizioAOperations {
  public double getSin(double x) {
    return Math.sin(x);
  }
}
```


Programmazione in CORBA

Creazione del Server.

L'attivazione di un servizio richiede:

- l'inizializzazione dell'ORB

```
ORB orb = ORB.init(args,null)
```

- la creazione di un Object Adapter collegato all'ORB

```
POAobj = orb.resolve_initial_references("RootPOA")
POA rootPOA = POAHelper.narrow(POAobj);
```

- la creazione dell'oggetto servant
  - la creazione dello skeleton se si usa l'approccio tie;
- il collegamento dell'oggetto servant al POA;
- l'attivazione dell'oggetto CORBA;
- l'inizio del ciclo di attesa delle richieste.

Programmazione in CORBA

Esempio di Creazione del Servant.

Creare un servant legato per ereditarietà allo skeleton.

```
ServizioAImpl AServant = new ServizioAImpl();
org.omg.CORBA.Object objA = rootPOA.servant_to_reference(AServant);
```

Creare un servant che usa il tie approach.

```
ServizioAImpl AServant = new ServizioAImpl();
ServizioAPOATie APOATie = new ServizioAPOATie(AServant);
org.omg.CORBA.Object objB = rootPOA.servant_to_reference(APOATie);
```

In entrambi i casi, l'attivazione dei servizi avviene tramite il POA.

```
rootPOA.the_POAManager().activate();
orb.run();
```


Programmazione in CORBA

Lato Server: Passaggio dei Parametri.

Parametro inout di tipo semplice.

Si consideri la seguente interfaccia:

```
interface Servizio2 {
  void addAndGet(inout long i);
};
```

Il server che implementa tale interfaccia è:

```
public class Servizio2Impl extends Servizio2POA {
  private int internalCntr = 0;

  public void addAndGet(org.omg.CORBA.IntHolder ih) {
    internalCntr += ih.value;
    ih.value = internalCntr;
  }
}
```

Si passa il parametro inout utilizzando la classe Holder relativa (IntHolder).

Programmazione in CORBA

Lato Server: Passaggio dei Parametri (Segue).

Parametro inout di tipo definito dall'utente.

Si consideri la seguente interfaccia:

```
struct ComplexNr {
  float re;
  float im;
};
interface Servizio3 {
  void coniugato(inout ComplexNr cx);
};
```

Il server che implementa tale interfaccia è:

```
public class Servizio3Impl extends Servizio3POA {
  public void coniugato(ComplexNrHolder cxh) {
    System.out.println("Call coniugato("+cxh.value.re+","+cxh.value.im+")");
    cxh.value.im = -cxh.value.im;
  }
}
```

La classe Holder è generata automaticamente a partire dall'IDL.

Programmazione in CORBA

Lato Server: Valori di Ritorno ed Eccezioni.

Valore di ritorno ed eccezioni definite dall'utente.

Si consideri la seguente interfaccia:

```
exception DivisionePerZero {
  string msg;
};
interface Servizio4 {
  ComplexNr reciproco(in ComplexNr cx) raises (DivisionePerZero);
};
```

Il server che implementa tale interfaccia è:

```
public class Servizio4Impl extends Servizio4POA {
  public ComplexNr reciproco(ComplexNr cx) throws DivisionePerZero {
    ComplexNr cx1 = new ComplexNr();
    float cxmod = cx.re*cx.re+cx.im*cx.im;
    if(cxmod==0) throw new DivisionePerZero("Mod="+cxmod);
    else { cx1.re = cx.re/cxmod; cx1.im = -cx.im/cxmod; return cx1; }
  }
}
```


Programmazione in CORBA

Lato Server: Valori di Ritorno ed Eccezioni (Segue).

Valore di ritorno ed eccezioni definite dall'utente.

Si consideri la seguente interfaccia:

```
interface Servizio5 {
  any qualsiasi();
};
```

Il server che implementa tale interfaccia è:

```
public class Servizio5Impl extends Servizio5POA {
  public org.omg.CORBA.Any qualsiasi() {
    org.omg.CORBA.Any a = orb.create_any();
    if (Math.random()<0.5) {
      String obj = "Oggetto di tipo stringa";
      a.insert_string(obj);
    } else a.insert_long(10);
    return a;
  }
}
```

Any ha un insieme di metodi per inserire oggetti dei tipi predefiniti.

Programmazione in CORBA

Portable Object Adapter (POA).

Gestisce il ciclo di vita degli oggetti CORBA.

Si possono configurare opportune politiche per gestire

- la persistenza degli object reference;
- l'attivazione trasparente degli oggetti CORBA;
- l'accesso agli oggetti serventi in caso di richieste concorrenti.

Il POA (Portable Object Adapter) ha un struttura gerarchica

- la radice è il RootPOA che si può ottenere dall'ORB

```
POAobj = orb.resolve_initial_references("RootPOA")
POA rootPOA = POAHelper.narrow(POAobj);
```

- da un POA si possono poi generare altri POA;
- ogni POA mantiene una mappa degli oggetti attivi (AOM);
- il RootPOA ha politiche di gestione degli oggetti serventi predefinite e non modificabili; i POA creati dall'utente possono essere configurati secondo le esigenze.

Programmazione in CORBA

POA: Active Object Map (AOM).

L'Active Object Map mantiene la corrispodenza fra oggetto CORBA e servant.

L'Object Id è utilizzato per identificare un oggetto all'interno del POA.

L'Object Id può essere generato automaticamente o assegnato dall'utente.

- Si può associare lo stesso servant a più oggetti CORBA (si possono associare più Object Id).

L'attivazione di un oggetto CORBA consiste nella creazione di un Object Reference cioè nella creazione di un elemento nella AOM.

Programmazione in CORBA

POA: Politiche di un POA.

Si possono creare più POA nell'applicazione per gestire gruppi di oggetti gestiti con politiche diverse.

Le politiche sono oggetti CORBA.Policy che possono essere generati da metodi factory (costruttori) del POA.

- Gli oggetti che rappresentano le politiche da attivare sono raccolti in un array che è passato come parametro al metodo che genera il nuovo POA;
- ServantRetentionPolicy;
- RequestProcessingPolicy;
- LifespanPolicy;

Programmazione in CORBA

POA: Retention e Processing Policies.

Definiscono le politiche per l'associazione Object Id e Servant

- di default il POA usa l'Active Object Map per memorizzare tutti gli oggetti serventi che gestisce (`RETAIN`) e permette di gestire richieste solo agli oggetti nella AOM (`USE_ACTIVE_OBJECT_MAP_ONLY`);
- si può definire una politica basata sull'uso di un Servant Manager associato al POA che ha il compito di reperire l'oggetto Servant associato ad un dato Object Id (`USE_SERVANT_MANAGER`);
  - in questo caso si può o meno usare l'AOM: `RETAIN` – si usa un Servant Activator; `NO_RETAIN` – si usa un Servant Locator.
- si può definire una politica di processing che invia tutte le richieste per Object Id non noti al POA a un Default Servant;
- il POA è associato a un Servant Manager od ad un Default Servant tramite appositi metodi (es. `poa.set_servant_manager()`).

Programmazione in CORBA

POA: POA Manager.

Incapsula lo stato di elaborazione delle richieste dai POA che gestisce.

Un POA Manager è associato ad un POA all'atto della creazione.

Si ottiene il riferimento al POA manager con

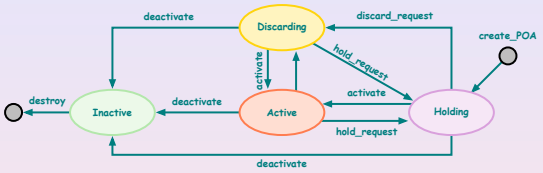
```
POAManager poam = poa.the_POAManager();
```

Uno stesso POA Manager può essere associato a più POA.

Il POA Manager può trovarsi in 4 stati diversi.

Programmazione in CORBA

POA: Stati del POA Manager.



Active – tutte le richieste in arrivo sono servite.

Holding – le richieste per i POA gestiti sono accodate senza consegnarle (si può esaurire lo spazio nella coda).

Discarding – tutte le richieste sono scartate al loro arrivo e rinviate ai client.

Inactive – Il POA Manager non è più in grado di ricevere richieste.

Riferimenti Bibliografici

- Michi Henning and Steve Vinoski.
  Advanced CORBA Programming with C++.
  Addison-Wesley, Reading, Massachusetts, 1999.
- Object Management Group.
  The Common Object Request Broker: Architecture and Specification.
  Tech. Rep. 2001.02.01 Revision V.2.4.2, OMG, February 2001.
- Object Management Group.
  Common Object Request Broker Architecture: Core Specification.
  Tech. Rep. 2002.12.06 Revision V.3.0.2, OMG, December 2002.
- Robert Orfali and Dan Harkey.
  Client/Server Programming with Java and CORBA.
  John Wiley & Sons, Inc., second edition, 1998.