Note sulla Common Object Request Broker Architecture (CORBA)

A.A. 2007–2008

draft

Andrea Domenici

9 gennaio 2008

Indice

1 Introduzione

Lo sviluppo delle reti di calcolatori rende possibile la realizzazione di applicazioni distribuite di grandi dimensioni e capaci di evolversi per adeguarsi a un numero crescente di utenti e fornitori di nuovi servizi. I progettisti di tali applicazioni devono soddisfare requisiti che nei sistemi distribuiti di grandi dimensioni sono particolarmente importanti, fra i quali:

- **scalabilità**;
- **riusabilità**;
- **indipendenza dalla piattaforma di calcolo**;
- **indipendenza dai linguaggi di programmazione**.

È ragionevole aspettarsi che le tecniche di progetto orientate agli oggetti possano contribuire alla soddisfazione dei requisiti enunciati, pertanto sono state sviluppate delle *architetture a oggetti distribuiti*, cioè degli insiemi di strumenti (materiali, come librerie e programmi eseguibili, o concettuali, come protocolli di comunicazione o regole di programmazione) da usare nello sviluppo di sistemi distribuiti, e fondati sui concetti del progetto orientato agli oggetti. In questa dispensa tratteremo, in modo molto sommario e parziale, la *Common Object Request Broker Architecture* (*CORBA*) [5].

L'architettura CORBA è un insieme di specifiche prodotte e aggiornate dall'*Object Management Group* (v. `www.omg.org`), un consorzio internazionale di organizzazioni pubbliche o private che hanno interesse nello sviluppo o nell'applicazione di software. Le specifiche CORBA sono indirizzate a sistemi distribuiti, generalmente di grandi dimensioni ed eterogenei rispetto alle piattaforme hardware e software su cui si trovano i loro componenti, e capaci di evolversi nel tempo. Nel seguito, termini come "sistemi CORBA", "librerie CORBA" e simili, denoteranno dei componenti software che implementano le specifiche CORBA.

Le specifiche CORBA sono molto complesse, dovendo coprire una grande quantità di problemi e di tecniche, e dovremo limitarci ad una scelta molto limitata degli argomenti relativi. Lo scopo di questa dispensa è solo di fornire

alcuni concetti fondamentali (ma non tutti i concetti fondamentali) e di dare un'idea generale dell'architetura CORBA.

1.1 Applicazioni distribuite

Un'applicazione distribuita è formata da vari componenti che vengono eseguiti su diversi *nodi*, o *host*, collegati fra di loro da una rete di comunicazione. Per introdurre alcuni concetti elementari, consideriamo due calcolatori, A e B, collegati direttamente fra loro, per esempio attraverso una rete Ethernet. Un'applicazione distribuita sarà formata da due programmi, P_A e P_B, che girano rispettivamente sui due calcolatori. I due programmi dovranno scambiarsi informazioni e sincronizzare le proprie operazioni, scambiandosi dei messaggi. Bisognerà quindi, innanzitutto, definire un *protocollo* di comunicazione, cioè un insieme di regole che specificano quali tipi di messaggi possono essere scambiati, qual è il formato di ciascun tipo di messaggio, e quali sequenze di messaggi sono possibili. Su ciascun nodo, quindi, ci devono essere dei componenti software che implementano tale protocollo, cioè che costruiscono, trasmettono, ricevono e interpretano i messaggi come richiesto dal protocollo. Questi componenti, a loro volta, si servono di altri componenti che, controllando le schede Ethernet dei due calcolatori, provvedono a trasmettere e ricevere fisicamente i messaggi. Questi ultimi componenti, in particolare, dovranno conoscere gli indirizzi fisici dei due calcolatori, cioè gli identificatori delle schede Ethernet.

Naturalmente, le applicazioni distribuite sarebbero molto costose se per ciascuna di esse si dovesse realizzare *ex novo* tutta l'infrastruttura di comunicazione, cioè definire un protocollo e implementarlo fino al livello della programmazione della scheda di rete. Ma questo non è necessario, poiché esistono dei protocolli standard implementati da librerie (*software di rete*) disponibili per tutte le piattaforme. Queste librerie offrono dei servizi di trasmissione su reti di comunicazione generalmente molto piú complesse di quella qui considerata, e nascondono tale complessità dietro alla loro *interfaccia di programmazione* (*Application Programming Interface, API*). Per progettare un'applicazione distribuita basta definire un protocollo ad alto livello, specifico dell'applicazione, che viene poi implementato usando l'interfaccia di programmazione di un protocollo standard.

Il protocollo piú diffuso è il TCP/IP (che piú precisamente è un insieme di protocolli). Nel TCP/IP ogni nodo è identificato da un *indirizzo IP* (una quadrupla di numeri), e, nell'àmbito di ciascun nodo, ogni *servizio*, cioè ogni programma predisposto per ricevere messaggi da sistemi remoti, viene iden-

tificato da un numero di *port*. L'interfaccia di programmazione delle librerie TCP/IP ha delle chiamate che permettono di stabilire una *connessione* con un servizio, dato il suo numero di port e l'indirizzo IP del nodo su cui si trova. Nel nostro esempio, il calcolatore A avrebbe l'indirizzo IP I_A, ed il programma P_A offrirebbe un servizio sul port N_{PA}; il programma P_B, conoscendo questi dati, potrebbe chiedere al software di rete esistente su B di stabilire una connessione con P_A. I due estremi della connessione sono rappresentati da due *socket* ("prese"). Il socket è un'astrazione simile al file, e viene usato in modo simile: l'interfaccia di programmazione comprende operazioni di lettura e di scrittura che permettono di inviare o ricevere blocchi di byte (di lunghezza arbitraria) attraverso un socket.

Per mezzo del protocollo TCP/IP si possono trasmettere, come abbiamo visto, solo dati non strutturati, cioè semplici sequenze di byte: spetta ai programmi applicativi trasformare i propri dati in sequenze di byte (*marshaling*), e ricostruire i dati a partire dai byte ricevuti (*unmarshaling*). Supponiamo, per esempio, che il programma P_B debba spedire a P_A il valore di una variabile, e che questo valore sia una struttura dati complessa, come una lista o un albero. Una struttura di questo genere è "tenuta insieme" da una rete di puntatori, ma il valore dei puntatori usati da P_B ovviamente è significativo solo nello spazio di memoria di P_B, e non certo in quello di P_A che si trova su un altro calcolatore. Per trasmettere questa struttura bisognerebbe ricopiarla nodo per nodo in un'altra struttura, assegnando un identificatore a ciascun nodo, e sostituendo ad ogni puntatore l'identificatore del nodo indirizzato dal puntatore stesso. Quindi bisognerebbe calcolare la dimensione della nuova struttura e passare questo dato, insieme all'indirizzo di memoria, all'operazione di scrittura su socket. Il programma P_A, a sua volta, dovrebbe ricostruire la struttura a partire dai byte "grezzi" ricevuti dal proprio socket. A tutto questo si aggiunge l'eventuale necessità di convertire i dati elementari (come i numeri reali o interi) in formati diversi, dal momento che i due calcolatori potrebbero avere architetture fisiche diverse.

Dato il basso livello di astrazione delle primitive dell'interfaccia di programmazione del TCP/IP (o di altri protocolli di trasporto), sono state sviluppate delle architetture che permettono di costruire applicazioni distribuite usando direttamente i tipi di dati offerti dai linguaggi di programmazione, e anche di eseguire *chiamate di procedure remote* (*Remote Procedure Calls, RPC*): usando un'architettura con RPC, nel programma P_B del nostro esempio si potrebbero invocare delle procedure (con la normale sintassi delle chiamate di procedura), che verrebbero eseguite da P_A. Proseguendo su questa linea, sono state sviluppate le architetture a oggetti distribuiti, come l'architettura CORBA di cui si tratterà nel seguito.

1.2 L'architettura CORBA

L'architettura CORBA comprende:

- un modello computazionale orientato agli oggetti (*Object Model*);
- un'infrastruttura di comunicazione su rete (*Object Request Broker*, ORB);
- un linguaggio per la descrizione delle interfacce (*IDL*);
- librerie di interfacce per componenti standard (*CORBA Services*, *CORBA Facilities* e *CORBA Domains*).
- un modello computazionale orientato ai componenti (*CORBA Component Model*, CCM);

Nel modello computazionale CORBA, un'applicazione è formata da *oggetti* che offrono servizi specificati da *interfacce*. Un'interfaccia è la descrizione di un insieme di *operazioni* che un cliente può richiedere a un oggetto. Si dice che un cliente invia una *richiesta* a un oggetto per ottenerne un servizio. L'informazione associata ad una richiesta consiste di un'operazione, di un oggetto destinatario, e di eventuali *parametri attuali*. Un cliente può ottenere un *valore restituito*. Se si verificano condizioni anormali nell'esecuzione di una richiesta, viene restituita un'*eccezione*.

Osserviamo che un oggetto, a sua volta, può essere cliente di altri oggetti. Nel seguito gli oggetti saranno considerati solo nel loro ruolo di fornitori di servizi, ma non bisogna dimenticare che possono assumere anche il ruolo di clienti. Inoltre, ci riferiremo sempre a situazioni in cui clienti e oggetti si trovano su nodi distinti, poiché è questo tipo di situazione che dà la motivazione iniziale allo sviluppo dei sistemi distribuiti, però uno degli obiettivi principali dell'architettura CORBA, come di altre architetture distribuite, è la *trasparenza rispetto alla localizzazione*: si vogliono cioè realizzare delle applicazioni che possano essere trasportate senza sforzo da piattaforme distribuite a piattaforme non distribuite e viceversa, mantenendo inalterati il codice e la struttura dell'applicazione.

È opportuno distinguere fra l'oggetto e la sua implementazione concreta, chiamata *servente*, in un certo linguaggio di programmazione. Col termine *server* indicheremo il processo all'interno del quale vengono eseguiti uno o piú serventi.

I clienti possono accedere agli oggetti solo attraverso dei *riferimenti* (*object references*). I riferimenti sono dei "puntatori remoti" che contengono le informazioni necessarie a localizzare gli oggetti entro il sistema distribuito, e

vengono creati e interpretati dal sistema CORBA. Nel seguito, useremo il termine "riferimento CORBA", o semplicemente "riferimento", per "riferimento a un oggetto CORBA", da non confondere con i riferimenti del linguaggio C++.

La comunicazione fra clienti e oggetti avviene attraverso un meccanismo basato sul design pattern *Proxy* [3]: ogni cliente interagisce con un *proxy* (o *stub*) locale, che ha la stessa interfaccia dell'oggetto remoto che rappresenta. Nel proxy le operazioni definite dall'interfaccia sono implementate da metodi che inviano dei messaggi all'oggetto remoto. L'infrastruttura di comunicazione CORBA (*Object Request Broker*) trasmette il messaggio e, dal lato dell'oggetto, ritrasforma il messaggio in un'invocazione di operazione.

L'Object Request Broker è lo strato di software che permette la comunicazione attraverso un sistema distribuito. Dal lato del cliente, l'ORB fornisce il riferimento all'oggetto, intercetta (col meccanismo del proxy) le richieste fatte dal cliente attraverso tale riferimento, le traduce in messaggi e le spedisce all'oggetto remoto, e infine riceve i messaggi di risposta dell'oggetto, ricostruisce gli eventuali valori restituiti, e li consegna al cliente. Dal lato dell'oggetto, l'ORB individua l'oggetto destinatario (creandolo o attivandolo se necessario), traduce i messaggi in invocazioni di operazioni sull'oggetto, ottiene i risultati, li traduce in messaggi e li spedisce al cliente. Il compito principale dell'ORB è quindi di nascondere i meccanismi di comunicazione fra gli oggetti, permettendo al programmatore di manipolare in modo uniforme oggetti locali e remoti. La Fig. 1 schematizza il meccanismo della trasmissione di richieste, mettendo in evidenza alcuni componenti dell'architettura.

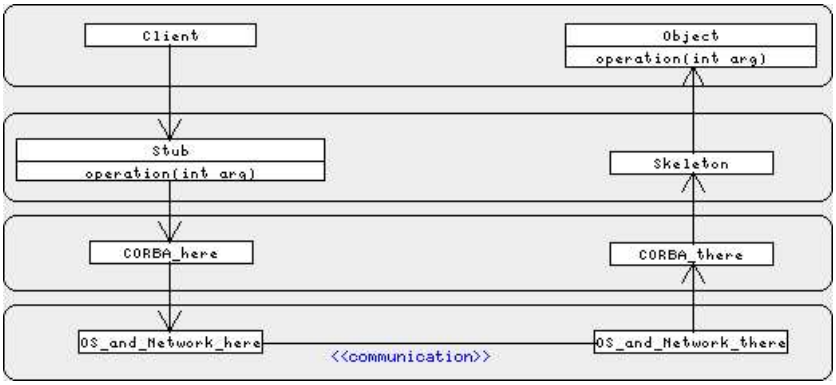

Figura 1: Architettura CORBA

I clienti e gli oggetti possono essere implementati in linguaggi diversi. L'interfaccia di ciascun oggetto viene specificato nel linguaggio *OMG Interface Definition Language* (*IDL*), che è un linguaggio puramente dichiarativo orientato agli oggetti, con eredità multipla e un sistema di tipi simile a quello del C++. Un'interfaccia IDL può essere implementata in qualsiasi linguaggio, purché l'implementazione soddisfi certi requisiti standard detti *mapping* di linguaggio. I mapping riguardano principalmente la corrispondenza fra i tipi di dati definiti nell'interfaccia IDL e quelli specifici del linguaggio d'implementazione, e inoltre specificano l'interfaccia di programmazione dell'ambiente CORBA, cioè le chiamate di libreria che il programmatore dei clienti o degli oggetti deve usare per accedere ai servizi dell'ambiente CORBA. Tutti i componenti di un'architettura CORBA si presentano al programmatore come oggetti, poiché i loro servizi sono specificati come interfacce IDL. I programmatori hanno a disposizione dei *compilatori IDL* che traducono ciascun'interfaccia IDL in un insieme dichiarazioni nel linguaggio d'implementazione, secondo il mapping corrispondente. Sono stati standardizzati i mapping per i linguaggi C, C++, Java, Smalltalk, Cobol, e Python.

Per ogni interfaccia, il compilatore IDL produce un proxy per il lato cliente ed uno *scheletro* per il lato server, che serve da base per l'implementazione dell'oggetto, cioè il servente. I programmatori usano dei *riferimenti locali* per accedere ai proxy, ai serventi, e ad altri componenti creati o gestiti del sistema CORBA. In C++ questi componenti sono istanze di classi, ed i riferimenti locali sono delle variabili (il cui tipo viene definito dal compilatore IDL) che si usano come puntatori.

I *CORBA Services*, le *CORBA Facilities* e i *CORBA Domains* definiscono le interfacce di oggetti standard che forniscono servizi a livello piú alto di quelli offerti dall'ORB. I CORBA Services sono concepiti per essere usati in tutti i tipi di applicazioni: per esempio, il servizio di *Naming* permette di assegnare agli oggetti dei nomi simbolici e di pubblicare in rete tali nomi, facilitando l'uso degli oggetti. I CORBA Domains sono orientati a settori d'applicazione specifici, come, per esempio, la medicina o le telecomunicazioni, e le CORBA Facilities specificano servizi di livello intermedio. Se sono disponibili programmi o librerie di oggetti che implementano le specifiche (espresse come interfacce IDL) di alcuni CORBA Services o Facilities, i progettisti le possono usare come componenti prefabbricati in applicazioni complesse.

2 Un esempio

Questo esempio è tratto da [4].

Vogliamo rendere disponibile in rete un servizio che consiste nel fornire l'ora di Greenwich (GMT), avendo a disposizione un'implementazione CORBA predisposta per lo sviluppo in ambiente C++. Questo significa che abbiamo delle librerie che implementano l'ORB e altri componenti, dei file di intestazione con le dichiarazioni in C++ delle risorse di tali librerie, un compilatore da IDL a C++, ed eventualmente altri strumenti ausiliari.

Prima di tutto scriviamo un file IDL (`time.idl`) contenente la definizione dell'interfaccia e dei tipi di dato usati nell'interfaccia stessa:

```
// time.idl

struct TimeOfDay {
    short   hour;
    short   minute;
    short   second;
};

interface Time {
    TimeOfDay   get_gmt();
};
```

L'interfaccia è costituita dall'operazione `get_gmt()`, che restituisce un valore di tipo `TimeOfDay`.

2.1 Uso del compilatore IDL

Il compilatore IDL produce, a partire da `time.idl`, alcuni file in C++che sono la base su cui si costruisce l'applicazione distribuita. Il numero e il contenuto di questi file non è fissato dagli standard CORBA, per cui cambia da un'implementazione all'altra, ma in ogni caso verranno prodotte:

1. le dichiarazioni di classi e tipi corrispondenti alle interfacce ed ai tipi dichiarati nel file IDL, secondo le regole del mapping;
2. un'implementazione parziale dell'oggetto, detta *scheletro* (*skeleton*);
3. un'implementazione del proxy, generalmente chiamata *stub* (se si desidera che anche i clienti si possano scrivere nello stesso linguaggio usato per implementare l'oggetto).

Lo scheletro e lo stub contengono il codice che collega all'ORB le parti dell'applicazione sviluppata *ad hoc* dai programmatori, rispettivamente dal lato del server e dal lato del cliente. Supponiamo che il nostro compilatore produca il file `ITime.h` per le dichiarazioni, i file `timeS.h` e `timeS.cc` per lo scheletro, e il file `timeC.cc` per lo stub. Tutti questi file, prodotti automagicamente del compilatore IDL, vengono usati "a scatola chiusa" dai programmatori.

Il file `timeS.h` contiene la dichiarazione della classe `POA_time`, che verrà usata dal programmatore per implementare l'oggetto. In questa classe viene dichiarata, fra l'altro, l'operazione virtuale pura `get_gmt()`:

```
// timeS.h

// ...

class POA_time : public virtual PortableServer::ServantBase {
    // ...
    virtual TimeOfDay get_gmt() throw (CORBA::SystemException) =0;
};
```

2.2 Implementazione dell'oggetto

Il programmatore deve ora realizzare ora una classe servente, che chiameremo `Time_impl`, che implementa effettivamente l'oggetto, calcolando l'ora di Greenwich. Supponiamo, per semplicità, che la definizione di questa classe sia contenuta nello stesso file, `myserver.cc`, in cui si trova il programma principale del server:

```
// myserver.cc

#include <iostream>
#include <time.h>
#include <orb.h>
#include "timeS.h"

class Time_impl : public virtual POA_Time {
    virtual TimeOfDay get_gmt() throw (CORBA::SystemException);
};

TimeOfDay
Time_impl::
get_gmt() throw (CORBA::SystemException)
{
    time_t time_now = time(0);
```

```
    struct tm* time_p = gmtime(&time_now);
    TimeOfDay tod;
    tod.hour = time_p->tm_hour;
    tod.minute = time_p->tm_min;
    tod.second = time_p->tm_sec;

    return tod;
}

int
main(int argc, char* argv[])
{
    // ...
}
```

Il file di intestazione `orb.h` fa parte della libreria CORBA (il nome dipende dall'implementazione). Il metodo `get_gmt()` usa le funzioni di libreria POSIX `time()` e `gmtime()`. La prima di queste restituisce, in secondi, il tempo trascorso dalle ore 00:00:00 UTC del 1^o gennaio 1970, la seconda estrae da questo numero la data (anno, mese,...) e la restituisce in una struttura di tipo `tm`. Queste funzioni sono dichiarate, insieme ai tipi `time_t` e `tm`, nel file standard d'intestazione `time.h`. La Fig. 2 schematizza la relazione fra le varie classi.

2.3 Implementazione del server

Infine, bisogna scrivere il programma principale del server. Le operazioni che deve svolgere sono:

1. inizializzare l'ORB;
2. creare un oggetto `Time` e un suo servente, cioè
   (a) creare un'istanza della classe `Time_impl`;
   (b) creare l'oggetto, cioè predisporre l'ORB a ricevere richieste per il servizio specificato dall'interfaccia `Time`;
   (c) associare il servente (istanza di `Time_impl`) all'oggetto;
3. pubblicare un riferimento CORBA all'oggetto `Time`;
4. mettersi in attesa di richieste.

Tutte queste operazioni vengono eseguite all'interno di un blocco `try`, poiché possono sollevare eccezioni.

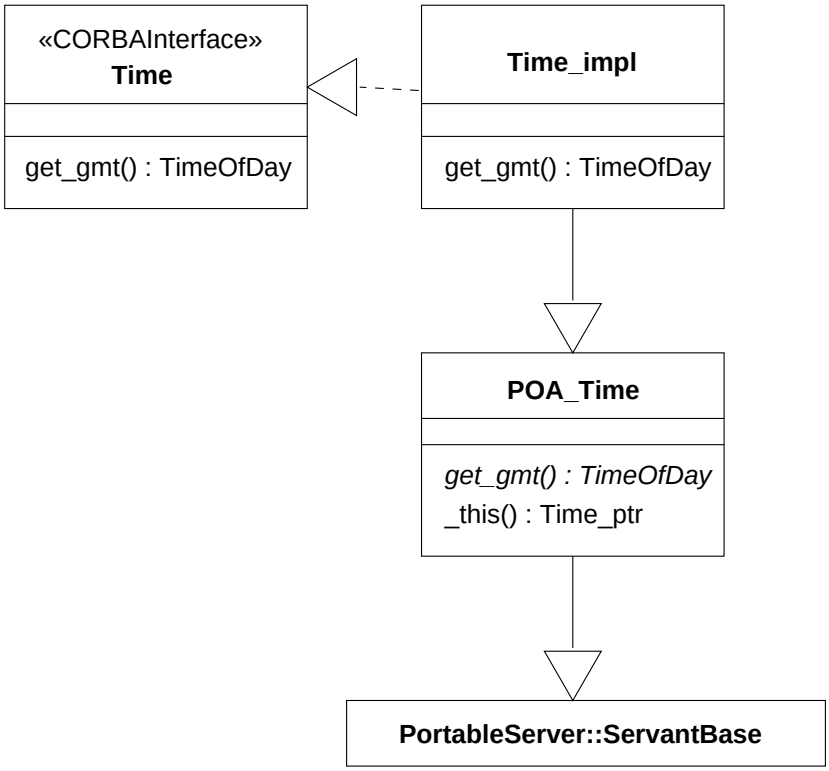

Figura 2: Implementazione di `Time`

L'inizializzazione dell'ORB comprende l'attivazione di tre componenti delle librerie CORBA: l'ORB propriamente detto, il *Portable Object Adapter*, e il gestore del POA. La funzione di questi ultimi due verrà spiegata in seguito (Sez. 5); per ora basta sapere che il server interagisce con essi attraverso dei riferimenti. Questi tre componenti si usano quindi come se fossero oggetti, e vengono chiamati *pseudo-oggetti*. L'inizializzazione dell'ORB avviene nel modo seguente:

```
// myserver.cc

// ...

int
main(int argc, char* argv[])
{
    try {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        CORBA::Object_var obj
            = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa
```

```
        = PortableServer::POA::_narrow(obj);
    PortableServer::POAManager_var mgr
        = poa->the_POAManager();
    mgr->activate();
// ...
```

La funzione `ORB_init` inizializza l'ORB e restituisce un *riferimento locale* allo pseudo-oggetto ORB. Attraverso questo riferimento (contenuto nella variabile `orb`) si invoca sull'ORB l'operazione `resolve_initial_references()`. Quest'operazione serve ad ottenere i riferimenti di alcuni componenti fondamentali dell'architettura, identificati da una stringa di caratteri. In questo caso si chiede il riferimento allo pseudo-oggetto POA di nome `RootPOA`. Osserviamo che `resolve_initial_references()` restituisce riferimenti locali a `Object`, il tipo base da cui derivano tutti i tipi (sia definiti dal programmatore che predefiniti) nell'architettura CORBA. Questo riferimento di tipo generico (`obj`) deve essere convertito, tramite l'operazione `_narrow()`, in un riferimento di tipo POA (`poa`). L'operazione `the_POAManager()` restituisce un riferimento (`mgr`) al gestore del POA, che viene quindi attivato dall'operazione `activate()`.

La creazione di un'istanza di `Time` avviene semplicemente dichiarando una variabile `time_servant` di tipo `Time_impl`:

```
Time_impl time_servant;
```

Osserviamo che la creazione di un'istanza di tipo `Time_impl` implica la creazione di un'istanza di `POA_Time`.

Affinché i clienti possano accedere al servizio `Time`, devono possedere un riferimento all'oggetto. Esistono diversi modi per mettere un riferimento a disposizione dei clienti, e in questo esempio useremo il piú semplice per il programmatore, anche se il meno comodo per l'utente dell'applicazione. Questo metodo sfrutta la possibilità di visualizzare un riferimento CORBA, in modo standard, per mezzo di una rappresentazione testuale, cioè sotto forma di una stringa di caratteri. Questa rappresentazione può essere diffusa con qualsiasi mezzo adatto a informazioni testuali, per esempio può essere inviata per posta elettronica, pubblicata su un sito web, o spedita con una cartolina postale. Nel nostro caso, il server si limita a scrivere sull'uscita standard la rappresentazione del riferimento:

```
Time_var tm = time_servant._this();
CORBA::String_var str = orb->object_to_string(tm);
cout << str << endl;
```

L'operazione `_this()` è definita nella classe `POA_Time` (che, ricordiamo, è stata prodotta dal compilatore IDL), e quindi appartiene all'interfaccia della classe `Time_impl` derivata da `POA_Time`. Quest'operazione crea un oggetto CORBA, associa il servant all'oggetto, e restituisce un riferimento locale (`tm`) al servant. L'operazione `object_to_string()`, appartenente all'interfaccia dell'ORB, produce la rappresentazione testuale del riferimento CORBA, che viene quindi scritta sull'uscita standard. Chi amministra il servizio `Time` provvederà a diffondere questa informazione nel modo appropriato.

Infine, il server si mette in attesa di richieste per l'oggetto `Time` invocando l'operazione `run()` sullo pseudo-oggetto ORB. Questa è l'ultima istruzione del blocco `try`, cui segue lo handler per le eccezioni eventualmente sollevate dalle librerie CORBA:

```
        orb->run();
    }
    catch (const CORBA::Exception &) {
        cerr << "Uncaught CORBA exception" << endl;
        return 1;
    }
    return 0;
}
```

Possiamo osservare che il codice del server non contiene alcuna dipendenza esplicita dal protocollo di rete o da altre informazioni relative al nodo su cui viene eseguito. Queste informazioni (come, per esempio, l'indirizzo del nodo) vengono gestite dall'ORB. Naturalmente le informazioni devono essere rese disponibili all'ORB, per esempio scrivendole in un file di configurazione, ma l'applicazione non ne dipende direttamente, per cui può essere portata su altri nodi della rete, o anche su reti diverse, con protocolli diversi, senza modifiche.

Su un sistema Unix il server può essere compilato con un comando di questo tipo, supponendo che il compilatore C++ si chiami `g++` e la libreria CORBA si chiami `liborb`:

```
% g++ -o myserver timeS.cc myserver.cc -lorb
```

Il server può quindi essere attivato mediante il comando:

```
% myserver > timeior.txt
```

La redirezione dell'uscita standard (rappresentata dall'operatore Unix '>') fa sí che la forma testuale del riferimento all'oggetto `Time` venga memorizzata nel file `timeior.txt`.

2.4 Implementazione di un cliente

I clienti del servizio `Time` possono essere scritti in qualsiasi linguaggio per cui esista un mapping da IDL, e qui immaginiamo di scriverne uno in C++.

Un cliente deve svolgere le seguenti operazioni:

1. inizializzare l'ORB;
2. ottenere un riferimento ad un oggetto `Time`;
3. usare l'oggetto `Time`;

L'inizializzazione dell'ORB nel cliente è piú semplice che nel server, poiché il cliente non usa il POA ed il relativo gestore:

```
#include <iostream>
#include <orb>
#include "ITime.h"

int
main(int argc, char* argv[])
{
    try {
        // ...
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

Per ottenere il riferimento all'oggetto `Time` implementato dal server, immaginiamo che la sua rappresentazione testuale venga passata come argomento sulla linea di comando che attiva il cliente. Il programma può accedere a tale argomento attraverso il puntatore `argv[1]`:

```
        CORBA::Object_var obj = orb->string_to_object(argv[1]);
        if (CORBA::is_nil(obj)) {
            cerr << "Nil Time reference" << endl;
            throw 0;
        }

        Time_var tm = Time::_narrow(obj);
        if (CORBA::is_nil(tm)) {
            cerr << "Argument is not a Time reference" << endl;
            throw 0;
        }
```

Un modo piú comodo per passare la rappresentazione testuale di un riferimento è il passaggio del nome di un file contenente la rappresentazione, nel seguente modo:

```
        CORBA::Object_var obj
                = orb->string_to_object("file://remoteior.txt");
```

L'operazione `string_to_object()` ricostruisce un riferimento CORBA a partire dalla sua rappresentazione testuale; se la ricostruzione non è possibile, per esempio nel caso che la rappresentazione non sia corretta, l'operazione restituisce un riferimento locale nullo, altrimenti istanzia un proxy di tipo `Object`. La successiva istruzione `if` serve quindi a verificare il successo dell'operazione. Anche l'operazione `_narrow()`, necessaria per convertire il riferimento al tipo generico `Object` in un riferimento a `Time`, può restituire un riferimento nullo, nel caso che il suo argomento non sia un riferimento a `Time`[1]. Se l'operazione ha successo, viene creato un proxy di tipo `Time`.

Dopo aver ottenuto un riferimento (`tm`), lo si può usare come un semplice puntatore:

```
        TimeOfDay tod = tm->get_gmt();
        cout << "Time in Greenwich is "
             << tod.hour << ":"
             << tod.minute << ":"
             << tod.second << endl;
    }
```

Queste istruzioni concludono il blocco `try`, seguito da due handler:

```
    catch (const CORBA::Exception &) {
        cerr << "Uncaught CORBA exception" << endl;
        return 1;
    }
    catch (...) {
        return 1;
    }
    return 0;
}
```

[1]L'operazione usa il riferimento locale (nell'esempio, `obj`) per accedere al proxy che contiene il riferimento CORBA. In certi casi può anche contattare il server che implementa l'oggetto remoto.

Il primo handler è per le eccezioni CORBA, il secondo per le eccezioni sollevate dal cliente stesso se le operazioni `string_to_object()` o `_narrow()` restituiscono un riferimento nullo.

Il cliente può essere compilato con un comando di questo tipo:

```
% g++ -o myclient timeC.cc myclient.cc -lorb
```

Nell'ipotesi che la rappresentazione testuale del riferimento all'oggetto sia contenuta nel file `remoteior.txt`, il cliente può essere attivato mediante il comando:

```
% myclient `cat remoteior.txt`
```

oppure

```
% myclient file://remoteior.txt
```

3 Il linguaggio IDL

In questa sezione esaminiamo alcuni aspetti del linguaggio IDL, limitandoci a quelli riguardanti la parte dell'architettura CORBA trattata in questo corso.

3.1 Sintassi

La sintassi dell'IDL è molto simile a quella del C++: si possono usare gli stessi commenti, si usano le parentesi graffe per delimitare blocchi di dichiarazioni, e le dichiarazioni hanno la stessa struttura. Inoltre il testo di un file IDL può essere elaborato da un preprocessore identico a quello del C++, quindi i file IDL possono contenere direttive di inclusione e di compilazione condizionale. Nel seguito metteremo in evidenza le differenze principali.

Gli identificatori devono iniziare con un carattere alfabetico, ma, a differenza che nel C++, il carattere di sottolineatura ('_') non fa parte dei caratteri alfabetici, quindi può apparire solo all'interno o alla fine di un identificatore. Non è permesso avere due identificatori (p.es., `TimeOfDay` e `timeofday`) che differiscono solo per l'uso delle maiuscole. Le parole chiave devono essere scritte in minuscolo.

3.2 Tipi base e stringhe

La seguente tabella riassume le specifiche dei tipi base e dei tipi stringa:

Tipo	Dimensione	Valori
`boolean`		TRUE, FALSE
`char`	$\geq$ 8 bit	ISO Latin-1
`wchar`		
`octet`	$\geq$ 8 bit	0 – 255
`short`	$\geq$ 16 bit	-2^{15} — $2^{15}-1$
`unsigned short`	$\geq$ 16 bit	0 — $2^{16}-1$
`long`	$\geq$ 32 bit	-2^{31} — $2^{31}-1$
`unsigned long`	$\geq$ 32 bit	0 — $2^{32}-1$
`long long`	$\geq$ 64 bit	-2^{63} — $2^{63}-1$
`unsigned long long`	$\geq$ 64 bit	0 — $2^{64}-1$
`fixed`		
`float`	$\geq$ 32 bit	IEEE-754 single
`double`	$\geq$ 64 bit	IEEE-754 double
`long double`	$\geq$ 80 bit	IEEE-754 extended
`string`	variabile	
`wstring`	variabile	

Non viene specificato un limite inferiore per la dimensione della rappresentazione dei tipi `boolean` e `wchar`.

Il tipo `char`, analogo al tipo corrispondente del C++, rappresenta i caratteri dell'insieme ISO Latin-1, mentre il tipo `wchar` (*wide character*, caratteri estesi) può rappresentare caratteri di qualsiasi insieme, come, per esempio, l'insieme Unicode.

Il tipo `octet` serve a rappresentare quantità la cui rappresentazione non deve subire trasformazioni di alcun tipo durante la trasmissione fra nodi diversi. La rappresentazione di un valore aritmetico, per esempio, potrebbe essere trasformata se i due nodi hanno convenzioni diverse sull'ordine dei byte piú significativi (*big-endian* se i byte piú significativi hanno gli indirizzi piú alti, *little-endian* altrimenti). L'uso del tipo `octet` índica che tali trasformazioni non devono avvenire, permettendo di trasmettere informazioni non numeriche, per esempio immagini o suoni.

I tipi `fixed` rappresentano numeri a virgola fissa, e ogni tipo `fixed` è caratterizzato dal numero di cifre significative e dal numero di cifre decimali. Per esempio, l'espressione "`fixed<9,2> fx`" significa che `fx` è un numero

con nove cifre significative di cui due decimali. I tipi `fixed` sono previsti principalmente per calcoli finanziari.

I tipi `string` e `wstring` rappresentano, rispettivamente, le stringhe di `char` e `wchar`. Una stringa si dice *limitata* se si esprime un limite al numero di caratteri che può contenere. Per esempio, "`string<10> s`" significa che `s` è una stringa di al piú dieci caratteri.

3.3 Il tipo `any`

Il tipo `any` viene usato per passare valori il cui tipo non è noto a tempo di compilazione. Una variabile di tipo `any` è una struttura contenente un valore di tipo qualsiasi ed una descrizione del tipo cui appartiene tale valore. È cosí possibile, a tempo di esecuzione, conoscere questo tipo.

```
struct Namedvalue {
    string name;
    any value;
};

typedef sequence<Namedvalue> ParamList;

interface AnInterf {
    void op(in ParamList p);
};
```

3.4 Tipi definiti dall'utente

Come "tipi definiti dall'utente" classifichiamo le *enumerazioni* (`enum`), le *strutture* (`struct`), le *unioni* (`union`), gli *array*, le *sequenze* (`sequence`), i *nomi* ***`typedef`***, e le *interfacce* (`interface`). Le interfacce saranno trattate in Sez. 3.7.

Le enumerazioni sono simili a quelle del C++. Nel seguente esempio il tipo `Color` è un'enumerazione i cui elementi possono assumere i valori `red`, `green`, o `blue`:

```
enum Color { red, green, blue };
```

I valori di un'enumerazione (detti *enumeratori*) sono ordinati in senso crescente con l'ordine di dichiarazione (`red` < `green` < `blue`), per cui si possono confrontare, ma, a differenza del C++, non hanno valori numerici.

Le strutture sono simili a quelle del C++. Un esempio di struttura IDL è `TimeOfDay`, in Sez. 2.

Le unioni sono simili ai record con parte variante del Pascal o dell'Ada: hanno un campo discriminante il cui valore determina il tipo del valore attuale dell'unione. Immaginiamo, per esempio, di dover rappresentare la quantità di un prodotto con un numero intero, se è confezionato, o un numero reale, se è sfuso:

```
enum ProductKind { packaged, bulk };
union Quantity switch (ProductKind) {
case packaged:
    unsigned short items;
case bulk:
    float weight;
};
```

I tipi array devono essere dichiarati come nomi `typedef`:

```
typedef Color ColorVector[10];
```

Questa dichiarazione significa che `ColorVector` è il tipo degli array contenenti dieci elementi di tipo `Color`. Osserviamo che il linguaggio IDL non specifica l'origine degli índici degli elementi.

Si possono dichiarare array multidimensionali, specificando esplicitamente tutte le dimensioni:

```
typedef string Table[10][20];
```

Le sequenze sono array unidimensionali con un numero variabile di elementi. È possibile specificare un numero massimo di elementi per una sequenza, che in questo caso si dice *limitata*. Questo limite si chiama *dimensione*, mentre il numero di elementi effettivamente contenuti si dice *lunghezza*:

```
typedef sequence<Color> Colors;
typedef sequence<long, 100> NumberList;
typedef sequence<NumberList> NunberLList;
```

Usando la derivazione di tipi per mezzo di strutture e unioni con membri di tipo sequenza si possono definire strutture dati ricorsive:

```
struct Node;
typedef sequence<Node> Nodes;
struct Node {
    long value;
    Nodes children;
};
```

3.5 Eccezioni

Le eccezioni sono strutture dati simili alle `struct`, ma non possono contenere membri di tipo eccezione e non possono essere passate come parametri o restituite come risultati. A differenza che nel C++, non c'è eredità fra eccezioni. Ecco due esempi di dichiarazione di eccezioni:

```
exception Failed {};

exception RangeError {
    unsigned long val;
};
```

Esiste un certo numero di eccezioni predefinite di sistema, a cui se ne possono aggiungere altre in futuro. Queste eccezioni segnalano problemi come l'invocazione di un'operazione non valida (`BAD_OPERATION`), errori di allocazione di memoria (`NO_MEMORY`), errori di comunicazione (`COMM_FAILURE`), e simili. Hanno tutte la stessa struttura, contenente due membri chiamati `minor` e `completed`, come in questo esempio:

```
exception BAD_OPERATION {
    unsigned long minor;
    completion_status completed;
};
```

Il significato del campo `minor` minor dipende dall'implementazione, e può essere usato dall'ORB per aggiungere all'eccezione informazioni piú dettagliate. Il campo `completed` serve a sapere se l'operazione che ha sollevato l'eccezione ha avuto effetto oppure no. Il valore di questo campo è di tipo `completion_status`, cosí definito:

```
enum completion_status { COMPLETED_YES, COMPLETED_NO,
                         COMPLETED_MAYBE };
```

dove gli enumeratori hanno questo significato:

- `COMPLETED_YES`: l'eccezione è avvenuta dopo che eventuali effetti collaterali richiesti dall'operazione hanno avuto luogo;
- `COMPLETED_NO`: l'operazione non è stata invocata o non ha avuto effetti collaterali;
- `COMPLETED_MAYBE`: non si può sapere se l'operazione ha avuto effetti collaterali o no.

3.6 Tipi *Value*

I tipi *Value* sono analoghi alle classi del C++, e le loro istanze possono essere passate per valore. Il seguente esempio mostra una dichiarazione di tipo value:

```
valuetype Node;

valuetype Node {
    factory init(in Object ref, in Node next);

    private Node next_node_;
    public Object ref_data_;

    Node next();
    void set_next();
};
```

L'operazione `init()` è un costruttore.

Si possono dichiarare tipi value astratti:

```
valuetype Node;

abstract valuetype ANode {
    Node next();
    void set_next();
};

valuetype Node : ANode {
    factory init(in Object ref, in Node next);

    private Node next_node_;
    public Object ref_data_;
};
```

3.7 Interfacce

Una dichiarazione di interfaccia può contenere dichiarazioni di tipo, di costante, di eccezione, di attributo, e di operazione. Per esempio:

```
interface AnInterface {
    typedef sequence<float> Fseq;
    const short MAX_LENGTH = 10;
    exception Exc {};
    Fseq get_seq(in short n) raises (Exc);
};
```

È possibile l'eredità, anche multipla, fra interfacce:

```
interface A : B, C { /* ... */ };
```

Ogni interfaccia è derivata implicitamente dall'interfaccia `Object`.

Le interfacce IDL sono simili alle classi del C++, ma ci sono alcune importanti differenze:

- un'interfaccia non può contenere definizioni di altre interfacce;
- un'interfaccia derivata non può ridefinire operazioni o attributi ereditati;
- l'overloading di operazioni non è possibile.

Le operazioni vengono specificate indicando il tipo del risultato (usando `void` se non viene restituito un risultato), il nome, eventuali parametri ed eccezioni sollevate:

```
void op();
short ops(in long l, inout float f, out char c);
Fseq get_seq(in short n) raises (Exc);
```

Per i parametri si devono indicare il nome, il tipo, e la *direzione* (`in`, `inout`, `out`).

Un'operazione può essere dichiarata `oneway` ("a senso unico"). Questo significa che per l'esecuzione dell'operazione ci si accontenta di una semantica *best-effort*: si accetta, cioè, che il sistema di comunicazione "faccia del suo meglio" per consegnare la richiesta al destinatario, ma senza alcuna garanzia

che la richiesta venga effettivamente consegnata. La semantica *best-effort* permette di sfruttare protocolli di comunicazione piú veloci anche se meno affidabili, qualora siano disponibili all'ORB. Un'operazione `oneway` non può restituire risultati, né avere parametri `out` o `inout`, né sollevare eccezioni:

```
oneway void send(in short data);
```

Osserviamo che alcuni testi affermano che le operazioni `oneway` vengono eseguite in modo asincrono, cioè senza che il chiamante resti bloccato in attesa del completamento dell'operazione. Questo può essere vero per alcune implementazioni dell'ORB, ma non è richiesto dalle specifiche CORBA e quindi è sconsigliabile fare affidamento sulla dichiarazione `oneway` per ottenere un'esecuzione asincrona dell'operazione.

In un'interfaccia si possono dichiarare degli attributi, che, sintatticamente, somigliano ai membri dato del C++, ma in effetti servono a specificare delle coppie di operazioni, cioè un'operazione di lettura e una di scrittura per ciascun attributo. Per esempio, le seguenti dichiarazioni di interfaccia sono logicamente equivalenti:

```
interface Thermometer {
    attribute short nominal_temp;
    readonly attribute short current_temp;
};
```

```
interface Thermometer {
    short get_nominal_temp();
    void set_nominal_temp(in short nt);
    short get_current_temp();
};
```

Nella seconda dichiarazione manca l'operazione di scrittura per l'attributo `current_temp`, poiché questo è stato dichiarato `readonly`.

3.8 Moduli

Le dichiarazioni IDL possono essere raggruppate in *moduli*, analoghi ai `namespace` del C++. Come nel C++, all'interno di un modulo si usa l'operatore di risoluzione di visibilità (`::`) per riferirsi a nomi dichiarati in altri moduli. I moduli possono contenere altri moduli:

```
module M {
    // ...
    module M1 {
        // ...
    };
    // ...
};
```

Un modulo può essere chiuso e riaperto come uno spazio di nomi del C++:

```
module A {
    interface X { /* ... */ };
};

module B {
    // ...
};

module A {
    interface Y { /* ... */ };
};
```

4 Il mapping per il C++

In questa sezione verranno trattati alcuni aspetti del mapping da IDL a C++.

4.1 Tipi base e stringhe

Gli identificatori usati nei file IDL vengono lasciati immutati nel mapping in C++, a meno che non corrispondano a parole chiave del C++, nel qual caso ricevono il prefisso `_cxx_`.

I moduli IDL diventano spazi di nomi C++, a meno che il compilatore C++ disponibile all'implementazione sia anteriore all'introduzione del costrutto `namespace` nel linguaggio; in questo caso i moduli IDL diventano classi. Nel seguito supporremo che si usino i `namespace`. Il modulo IDL chiamato CORBA, in particolare, diventa il `namespace CORBA`.

I tipi base IDL generalmente corrispondono a tipi definiti nel `namespace CORBA`, che a loro volta sono definiti in termini dei tipi C++, in modo che

dipende dall'implementazione: per esempio, il tipo `CORBA::Short` corrispondente al tipo IDL `short` può essere implementato come `short int` o `int` a seconda dell'architettura fisica. La seguente tabella riporta le corrispondenze fra tipi IDL e tipi C++:

Tipo IDL	Tipo C++
`boolean`	`CORBA::Boolean`
`char`	`CORBA::Char`
`wchar`	`CORBA::WChar`
`octet`	`CORBA::Octet`
`short`	`CORBA::Short`
`unsigned short`	`CORBA::UShort`
`long`	`CORBA::Long`
`unsigned long`	`CORBA::ULong`
`long long`	`CORBA::LongLong`
`unsigned long long`	`CORBA::ULongLong`
`float`	`CORBA::Float`
`double`	`CORBA::Double`
`long double`	`CORBA::LongDouble`
`string`	`char*`
`wstring`	`CORBA::WChar*`

Osserviamo che le stringhe vengono rappresentate semplicemente come puntatori a caratteri. Nel `namespace CORBA` sono definite delle funzioni per manipolare le stringhe di caratteri:

```
static char* string_alloc(Ulong len);
static char* string_dup(const char*);
static void string_free(char*);
```

La funzione `string_alloc()` alloca memoria per una stringa di `len` caratteri (cioè alloca `len` + 1 byte); la funzione `string_dup()` duplica una stringa, allocando la memoria necessaria; la funzione `string_free()` dealloca una stringa allocata da `string_alloc()`. Le funzioni `string_alloc()` e `string_free()` devono essere usate al posto degli operatori standard `new` e `delete[]`, poiché questi ultimi creano dei problemi su alcuni sistemi operativi molto diffusi, che usano un modello di memoria diverso da quello del sistema Unix.

Sono disponibili anche tre funzioni analoghe per le stringhe di caratteri estesi.

4.2 Classi wrapper

Le stringhe, i riferimenti e le sequenze sono tipi *a dimensione variabile*, cosí come le strutture, unioni o array che contengono stringhe, riferimenti o sequenze. Poiché la dimensione delle istanze di tali tipi non è nota a tempo di compilazione, tali istanze devono essere allocate in memoria dinamica. Il mapping del C++ prevede che ciascun tipo strutturato (potenzialmente a dimensione variabile) si traduca in due rappresentazioni, una "semplice" e una "incapsulata". La rappresentazione semplice richiede al programmatore di gestire esplicitamente la memoria dinamica, allocando e deallocando le istanze, mentre la rappresentazione incapsulata gestisce la memoria automaticamente. La rappresentazione semplice può essere una classe o un altro tipo, e la rappresentazione incapsulata è una classe detta *wrapper* o *puntatore intelligente*. I nomi delle classi wrapper terminano col suffisso '`_var`', per cui sono chiamate anche "classi `_var`".

La rappresentazione semplice di un tipo IDL strutturato T è un puntatore a `char` o `CORBA::WChar` per le stringhe, un array per gli array, una `struct T` per le strutture, una `class T` negli altri casi. A questi tipi si aggiungono altri tipi ausiliari. Le classi che rappresentano sequenze IDL, inoltre, hanno operazioni per leggere e modificare i loro elementi e per controllare certi aspetti implementativi che qui non verranno trattati.

Tutte le classi wrapper seguono uno stesso schema, offrendo determinati tipi di costruttori e distruttori, di operatori di copia ed assegnamento, di lettura e modifica, e di conversione. Se `T` è il nome della rappresentazione semplice di un tipo IDL T, la classe wrapper `T_var` contiene un puntatore privato `p` a `T`. Nella classe wrapper sono definiti:

- un costruttore default `T_var()` che inizializza `p` a zero.
- Un costruttore *di copia superficiale* (*shallow copy*) `T_var(T* tp)`, dove `tp` deve essere un puntatore ad un'istanza di `T` allocata in memoria libera. Il costruttore assegna `tp` a `p`, senza copiare l'istanza indirizzata da `tp`.
- Un costruttore *di copia profonda* (*deep copy*) `T_var(const T_var& tvr)`. Il costruttore copia l'istanza riferita da `tvr` (compresa l'istanza di `T` indirizzata da `tvr.p`), in una nuova area di memoria libera, il cui indirizzo viene assegnato a `p`.
- Un distruttore `T_var()` che dealloca l'istanza di `T` indirizzata da `p`.
- Un operatore di *assegnamento superficiale*

  ```
  T_var& operator=(T* tp)
  ```

dove `tp` deve essere un puntatore ad un'istanza di `T` allocata in memoria libera. L'assegnamento `tv = tp` viene eseguito deallocando l'istanza di `T` indirizzata da `tv.p` ed assegnando `tp` a `tv.p`, senza copiare l'istanza indirizzata da `tp`.

- Un operatore di *assegnamento profondo*

  ```
  T_var& operator=(const T_var& tvr)
  ```

  L'assegnamento `tv = tvr` viene eseguito deallocando l'istanza di `T` indirizzata da `tv.p` e copiando l'istanza riferita da `tvr` (compresa l'istanza di `T` indirizzata da `tvr.p`), in una nuova area di memoria libera, il cui indirizzo viene assegnato a `tv.p`.
- Due operatori di indirezione

  ```
  T* operator->()
  const T* operator->() const
  ```

  che permettono di invocare operazioni sull'istanza di `T` indirizzata da `p`.
- Due operatori di conversione

  ```
  operator T &()
  const operator T &() const
  ```

  che permettono di usare un'istanza di `T_var` dove si richiede un riferimento (nel senso del C++) a `T`.
- Per sequenze e array con elementi di tipo `TE`, due operatori di indicizzazione

  ```
  TE& operator[](CORBA::ULong)
  const TE& operator[](CORBA::ULong)
  ```

  per riferirsi a singoli elementi.

È importante osservare che le operazioni di copia e di assegnamento superficiali fanno sí che l'istanza di `T_var` oggetto dell'operazione diventi proprietaria dell'istanza di `T` indirizzata dal puntatore, cioè può accadere che la sua area di memoria venga deallocata da successive operazioni o dal distruttore di `T_var`. Quindi, per evitare errori di esecuzione, bisogna che l'istanza di `T` sia stata creata in memoria libera e non sullo stack o in memoria statica. Inoltre, dopo che un'istanza di `T_var` si è appropriata di una di `T`, bisogna evitare di accedere a quest'ultima direttamente attraverso puntatori a `T`.

Non tratteremo altre operazioni che possono appartenere all'interfaccia di classi wrapper, come certi operatori di conversione utili nel passaggio di parametri, ed altre operazioni per scopi particolari.

4.3 Interfacce (lato cliente)

Nel trattare il mapping delle interfacce bisogna distinguere fra lato cliente e lato server. Sul lato cliente un'interfaccia viene rappresentata da una classe proxy derivata da `CORBA::Object`. Per esempio, la seguente interfaccia:

```
interface MyObject {
    long get_value();
};
```

corrisponde a:

```
class MyObject : public virtual CORBA::Object {
public:
    virtual CORBA::Long get_value();
    // ...
};
```

Non è però possibile, nel codice cliente, istanziare esplicitamente una classe proxy. I proxy vengono istanziati internamente dall'ORB che, dopo aver creato un proxy per un oggetto remoto, rende disponibile al codice cliente un riferimento locale attraverso il quale interagire col proxy, e quindi con l'oggetto remoto.

L'oggetto remoto, come sappiamo, è identificato da un riferimento. Supponiamo che, come nell'esempio della Sez. 2, il cliente abbia una rappresentazione testuale del riferimento all'oggetto remoto che vuole usare, rappresentazione accessibile attraverso un puntatore `argv[1]`. Il cliente, avendo inizializzato l'ORB e ottenuto un riferimento `orb` allo pseudo-oggetto ORB, trasforma la rappresentazione testuale in un riferimento:

```
CORBA::Object_ptr obj = orb->string_to_object(argv[1]);
```

La chiamata `string_to_object()` compie due operazioni:

1. alloca un proxy di tipo `CORBA::Object`, e
2. restituisce un riferimento locale di tipo `CORBA::Object_ptr`.

Il proxy contiene le informazioni, ricavate dalla rappresentazione testuale del riferimento, necessarie a comunicare con l'oggetto remoto, e si serve di risorse fornite dal software di rete. Il riferimento locale `obj` è una struttura dati

(che potrebbe ridursi a un semplice puntatore C++) che permette di riferirsi al proxy e attraverso di esso all'oggetto remoto, quindi è funzionalmente equivalente a un riferimento all'oggetto. Però questo riferimento può essere usato solo per invocare le operazioni dell'interfaccia di `CORBA::Object`; per usare le operazioni specifiche dell'interfaccia `MyObject` (cioè l'operazione `get_value()`) bisogna convertirlo appropriatamente:

```
MyObject_ptr myobj = MyObject::_narrow(obj);
```

La chiamata `_narrow()` compie queste operazioni:

1. verifica che il riferimento sia del tipo giusto, e, se sí,
2. alloca un proxy di tipo `MyObject`, e quindi
3. restituisce un riferimento di tipo `MyObject_ptr`.

L'istanza originale del proxy di tipo `CORBA::Object` dovrà essere deallocata per risparmiare memoria libera. Piú oltre si accennerà alla gestione della memoria libera per i diversi tipi di riferimenti locali.

A questo punto si può usare `myobj` come se fosse un semplice puntatore:

```
CORBA::Long v = myobj->get_value();
```

I proxy possono essere creati con operazioni come `string_to_object()`, e duplicati e distrutti passando riferimenti locali come `myobj` ad altre operazioni:

```
MyObject_ptr yourobj = MyObject::_duplicate(myobj);
CORBA::release(myobj);
```

L'operazione `_duplicate()` fa una copia profonda del proxy e ne restituisce un riferimento; è possibile che l'implementazione non esegua una copia fisica ma si limiti a mantenere un conto delle copie "virtuali" richieste (meccanismo di *reference count*).

La distruzione dei proxy non piú usati è necessaria per evitare l'esaurimento delle risorse impiegate: non solo la memoria libera, ma anche le risorse del sistema operativo, come i socket. Se queste risorse non venissero liberate,

un cliente con molti proxy le potrebbe esaurire e "andare in crash". La distruzione deve avvenire esplicitamente, con l'operazione `release()`, per i riferimenti di tipo "`_ptr`". Esistono però anche i riferimenti di tipo "`_var`" che, analogamente alle classi wrapper, automatizzano la gestione della memoria.

I riferimenti `_var` hanno gli stessi tipi di costruttori e di operatori di assegnamento delle classi wrapper, ed hanno operatori di conversione nei corrispondenti riferimenti `_ptr`. Le operazioni definite nell'interfaccia dei proxy e degli pseudo-oggetti (come l'ORB) accettano come parametri e restituiscono come risultati dei riferimenti `_ptr`, ma gli operatori di conversione (invocati implicitamente dal compilatore C++) permettono di usare liberamente i riferimenti `_var`.

Si possono anche creare dei riferimenti nulli:

```
MyObject_ptr p = MyObject::_nil();
```

Ed è possibile verificare se un riferimento è nullo:

```
if (CORBA::is_nil(p))
    // ...
```

4.4 Operazioni (lato cliente)

Ogni operazione IDL viene tradotta in un'operazione C++ dello stesso nome. I parametri ed i valori restituiti vengono tradotti in base a certe regole dettate dall'esigenza di gestire correttamente la memoria dinamica. Data la varietà dei tipi di dato possibili, queste regole sono abbastanza complesse, ma si possono riassumere, con qualche approssimazione, come segue:

- i tipi a dimensione fissa vengono allocati e deallocati dal chiamante. I parametri `in` sono passati per valore se semplici (`char`, `short`,...), per riferimento se complessi (strutture e unioni a lunghezza fissa); i parametri `inout` e `out` sono passati per riferimento.
- Per i tipi a dimensione variabile, i parametri con direzione `in` o `inout` sono allocati e deallocati dal chiamante, e passati per riferimento. Quelli con direzione `out` sono allocati dal chiamato e deallocati dal chiamante, e passati per riferimento. I valori restituiti vengono allocati dal chiamato e deallocati dal chiamante.

Queste regole generali si applicano in vario modo ai diversi tipi (stringhe, array, strutture,...), tenendo conto delle loro particolarità. Per ogni tipo `T` è definito un tipo `T_out` da usare quando un'istanza di `T` deve essere passata come parametro `out`. Per esempio, la seguente operazione IDL:

```
long op_long(in long a, inout long b, out long c);
```

viene tradotta come:

```
virtual CORBA::Long op_long(CORBA::Long a,
                            CORBA::Long& b,
                            CORBA::Long_out c);
```

Infine, ricordiamo che l'uso di classi wrapper semplifica molto la gestione della memoria libera, liberando il programmatore dalla necessità di deallocare esplicitamente.

4.5 Interfacce (lato server)

Per il lato server, il compilatore IDL produce per ciascun'interfaccia una classe *scheletro*. Per esempio, la seguente interfaccia:

```
interface MyObject {
    long get_value();
};
```

si traduce nella seguente classe scheletro:

```
class POA_MyObject : public virtual PortableServer::ServantBase {
public:
    virtual CORBA::Long get_value()
                            throw (CORBA::SystemException) =0;
    // ...
};
```

La classe `PortableServer::ServantBase` fa parte della libreria CORBA e offre all'ORB (piú precisamente, all'Object Adapter) le operazioni necessarie a trasmettere all'oggetto le richieste remote. La classe `POA_MyObject`, prodotta dal compilatore IDL, eredita tali operazioni e definisce l'operazione virtuale pura `get_value()`. A questa operazione, che rappresenta quella

definita nell'interfaccia IDL, si aggiungono altre operazioni (non mostrate nell'esempio) usate dall'ORB o dall'applicazione.

La classe scheletro naturalmente è astratta, poiché il compilatore IDL non può "sapere" come deve essere implementata l'interfaccia: può solo produrre le dichiarazioni che permettono all'ORB di invocare le operazioni dell'interfaccia, e fornire i meccanismi che realizzano il collegamento fra l'ORB e l'implementazione dell'interfaccia. Questa implementazione è fornita dal programmatore, sotto forma di una classe *servente* derivata dalla classe scheletro:

```
class MyObject_impl : public virtual POA_MyObject {
public:
    MyObject_impl(CORBA::Long n) : value(n) {};
    virtual CORBA::Long get_value() throw (CORBA::SystemException);
    // ...
private:
    CORBA::Long value;
};

CORBA::Long
MyObject_impl::
get_value() throw (CORBA::SystemException)
{
    return value;
}

// ...
```

Questa classe contiene l'implementazione dell'interfaccia, insieme a qualsiasi cosa (membri dato e operazioni pubbliche e private, tipi, costanti,...) che possa servire all'implementazione. Osserviamo anche che in generale la classe servente da sola può non bastare a fornire il servizio richiesto, e quindi avrà bisogno della collaborazione di altre classi o di altri oggetti CORBA.

Il codice che costituisce la classe servente deve rispettare le regole di mapping già viste, sia per l'uso dei tipi corrispondenti a tipi IDL che per il passaggio di parametri ed altre convenzioni.

La classe servente deve essere istanziata, ma questo non basta a creare un oggetto CORBA, cioè un componente capace di ricevere richieste remote. Dopo aver istanziato un servente, bisogna registrarlo, cioè comunicare all'ORB (piú precisamente all'Object Adapter) che il servente è disponibile. Questo si può ottenere con l'operazione `_this()`, definita nella classe scheletro:

```
MyObject_impl anObject(666);
MyObject_var myobj = anObject._this();
```

L'operazione `_this()` registra il servente, creando cosí l'oggetto CORBA (si dice che l'oggetto è *incarnato* dal servente), e restituisce il riferimento a tale oggetto.

5 Il Portable Object Adapter

Un POA ha la funzione di creare, attivare, disattivare e distruggere gli oggetti CORBA, di registrare serventi per gli oggetti, e di smistare fra i serventi le richieste fatte agli oggetti. La Fig. 3 mostra in modo semplificato la relazione fra il POA, l'ORB, e le classi che implementano l'oggetto, per quanto riguarda lo smistamento delle richieste. Ad ogni POA e ad ogni istanza di una classe servente (**MyObject_impl**) è associato un identificatore (poaid e objectid, rispettivamente). Questi identificatori fanno parte del riferimento CORBA dell'oggetto implementato. Quando l'ORB riceve una richiesta, la inoltra al POA specificato nel riferimento, chiamando l'operazione upcall() del POA stesso. Questo, a sua volta, chiama l'operazione invoke() del servente specificato nella richiesta. Questa operazione, generata dal compilatore IDL e definita nella classe **POA_MyObject**, decodifica la richiesta e chiama l'operazione specificata, la cui implementazione è fornita dalla classe servente.

La funzioni di gestione degli oggetti, apparentemente semplici, possono essere svolte in diversi modi. Per esempio, si può volere che gli oggetti siano *persistenti*, cioè che continuino ad esistere anche quando il processo in cui sono stati creati è terminato, oppure che siano *transitori*, cioè che esistano solo finché esiste il POA che li ha creati. Ancora, si può volere che ciascun servente incarni un solo oggetto, oppure che possa incarnarne piú di uno. Queste scelte influenzano sensibilmente le prestazioni di un'applicazione distribuita, per cui è importante che il progettista le possa fare con la massima flessibilità. Per questo il POA è stato concepito come un componente configurabile, il cui comportamento può essere controllato attraverso un'interfaccia di programmazione, ovviamente definita tramite un'interfaccia IDL. Il progettista decide come devono essere gestiti gli oggetti e crea un POA configurandolo nel modo opportuno. Se nell'applicazione ci sono oggetti che devono essere gestiti in modo diverso, si possono creare piú POA, ciascuno configurato per un particolare gruppo di oggetti: per esempio, se un'applicazione richiedesse un oggetto persistente ed uno o piú oggetti transitori, allora servirebbero due POA. Ogni applicazione ha a disposizione un POA iniziale, il `RootPOA`, con

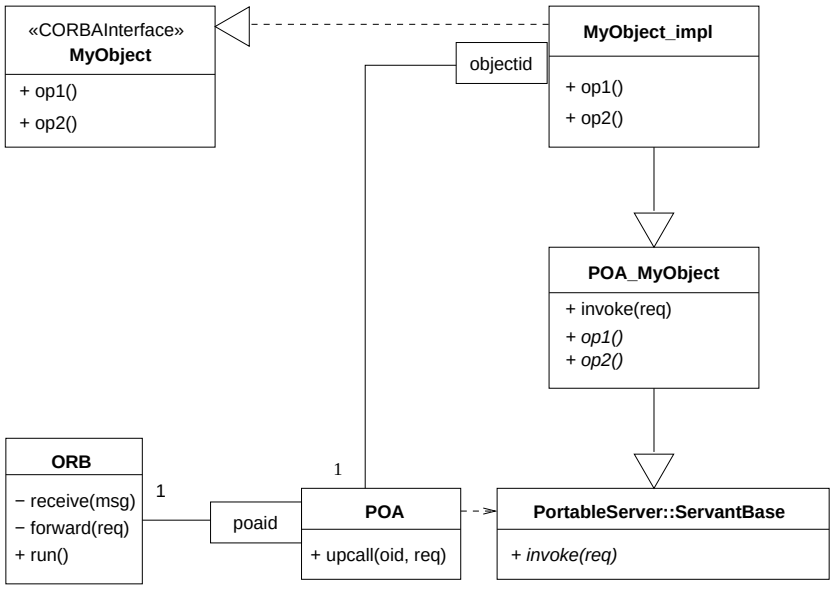

Figura 3: Portable Object Adapter

una configurazione standard, a cui si può chiedere la creazione di altri POA se necessario.

La Fig. 4 mostra la struttura tipica di un'applicazione CORBA dal lato server, mostrando le diverse interfacce dei principali componenti. Precisiamo che le interfacce mostrate in figura sono soltanto una schematizzazione molto semplificata degli insiemi di interfacce definite nelle specifiche CORBA. Piú precisamente, con ORBMgmt e POAMgmt indichiamo convenzionalmente le varie interfacce di gestione dell'ORB e del POA, con POAConfig l'interfaccia di gestione del POA (v. oltre), e con IDLInterf l'interfaccia IDL del servente, cioè l'insieme di operazioni specificate dall'interfaccia dell'oggetto.

5.1 Configurazione del POA

I criteri di gestione degli oggetti sono chiamati *politiche* (*policy*), e sono rappresentati da pseudo-oggetti definiti da interfacce IDL. Per esempio, la politica relativa alla persistenza degli oggetti viene rappresentata da un oggetto `LifespanPolicy`:

```
enum LifespanPolicyValue { TRANSIENT, PERSISTENT };
interface LifespanPolicy : CORBA::Policy {
```

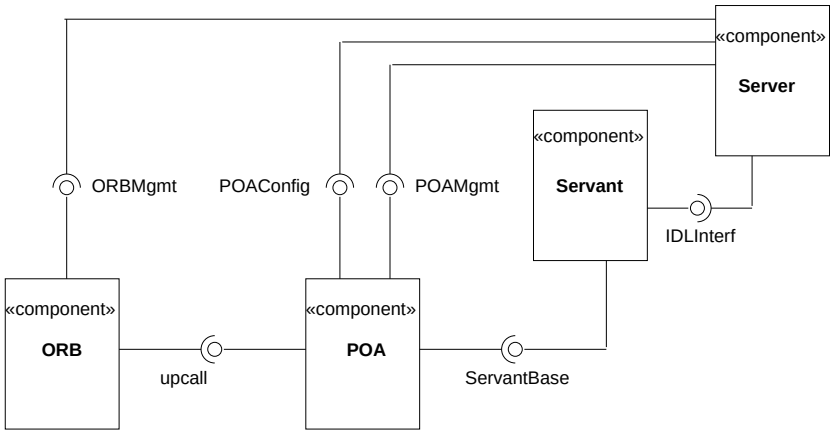

Figura 4: Portable Object Adapter

```
    readonly attribute LifespanPolicyValue value;
};
```

Questi oggetti devono essere creati da un POA, e vengono passati allo stesso POA per creare un nuovo POA che applica le politiche prescelte.

Per esempio, supponiamo di avere già un riferimento `rootPOA` al POA di default, che è configurato per gestire oggetti transitori. Se abbiamo bisogno di oggetti persistenti dobbiamo prima di tutto creare un oggetto `LifespanPolicy` col valore opportuno:

```
PortableServer::LifespanPolicy_var lsp =
        rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
```

Questo oggetto deve essere inserito in una sequenza di politiche, che a sua volta viene passata all'operazione che crea il nuovo POA:

```
CORBA::PolicyList pl;
pl.length(1);
pl[0] = PortableServer::LifespanPolicy::_duplicate(lsp);
PortableServer::POA_var newPOA = rootPOA->create_POA(/* ... */ pl);
```

5.2 Politiche del POA

Le politiche del POA permettono di controllare la persistenza (*lifespan*) degli oggetti, la gestione dei loro identificatori, la loro associazione ai *servant*, la

loro attivazione, la gestione delle richieste da parte dei servant, e la loro eventuale esecuzione con piú threaad.

L'associazione fra oggetti e *servant* viene realizzata per mezzo della *tabella degli oggetti attivi* (*Active Object Map*).

Nel seguito le politiche del POA vengono descritte sinteticamente. I valori di default sono scritti in neretto.

Lifespan (**TRANSIENT**, PERSISTENT) Gli oggetti transitori non possono durare piú del POA in cui sono stati creati.

IdAssignment (USER_ID, **SYSTEM_ID**) Gli identificatori degli oggetti possono essere scelti dall'utente o dal sistema.

IdUniqueness (**UNIQUE_ID**, MULTIPLE_ID) A ciascun *servant* può essere associato un unico oggetto, o piú oggetti.

ImplicitActivation (IMPLICIT_ACTIVATION, **NO_IMPLICIT_ACTIVATION**) L'attivazione di un oggetto, cioè la sua associazione ad un *servant*, può essere implicita, cioè contestuale alla sua creazione, o no, e in questo caso richiede due operazioni distinte. Il valore di default per questa politica è **NO_IMPLICIT_ACTIVATION**, ma il *root POA* fa eccezione, essendo configurato con IMPLICIT_ACTIVATION.

RequestProcessing (**USE_ACTIVE_OBJECT_MAP_ONLY**, USE_DEFAULT_SERVANT, USE_SERVANT_MANAGER) Le richieste possono essere eseguite solo dai *servant* registrati nella tabella del POA, opure da un *servant* di default, oppure da un *servant* scelto da un gestore.

ServantRetention (**RETAIN**, NON_RETAIN) Le associazioni fra oggetti e *servant* scelti dal gestore (v. sopra) possono essere conservate nella tabella del POA oppure cancellate al termine dell'esecuzione della richiesta.

Thread (**ORB_CTRL_MODEL**, SINGLE_THREAD_MODEL) Con la politica SINGLE_THREAD_MODEL, le richieste per gli oggetti del POA vengono elaborate sequenzialmente, altrimenti possono essere elaborate in modo concorrente sotto il controllo dell'ORB.

5.3 Gestione di oggetti e serventi

È possibile chiedere al POA di creare oggetti CORBA. In Sez. 4.5 abbiamo istanziato un servente e poi abbiamo creato un oggetto CORBA incarnato dallo stesso servente, attraverso l'operazione `_this()`; è anche possibile, però, separare la creazione degli oggetti CORBA e dei serventi. Un oggetto CORBA può essere creato, per esempio, con l'operazione `create_reference()`:

```
CORBA::Object_var obj = newPOA->create_reference(/* ... */);
```

In seguito si può creare un servente e attivare l'oggetto incarnandolo col servente, usando l'operazione `activate_object()`:

```
MyObject_impl thisobj;
PortableServer::ObjectId_var oid =
    newPOA->activate_object(&thisobj);
```

dove il valore restituito (di tipo `ObjectId`) è una sequenza di `octet` che identifica l'oggetto nell'àmbito del POA.

L'operazione `deactivate_object()` ovviamente disattiva l'oggetto.

5.4 Il gestore del POA

Infine accenniamo al *gestore del POA* (*POA manager*). Questo componente serve a controllare il flusso di richieste trasmesse dall'ORB al POA, che normalmente vengono passate immediatamente (stato **Active**, Fig. 5), ma possono essere accodate (stato **Holding**) o anche ignorate (stato **Discarding**); in quest'ultimo caso la richiesta solleva un'eccezione di sistema che viene inviata al cliente. Questi comportamenti vengono selezionati con le operazioni `POAManager::hold_requests()` e `POAManager::discard_requests()`, rispettivamente. Uno stesso gestore può controllare uno o piú POA.

6 Il servizio di Naming

Il servizio di naming permette di attribuire un nome simbolico a ciascun oggetto, e di ottenere un riferimento all'oggetto attraverso il suo nome, evitando cosí di dover trasmettere materialmente una rappresentazione testuale

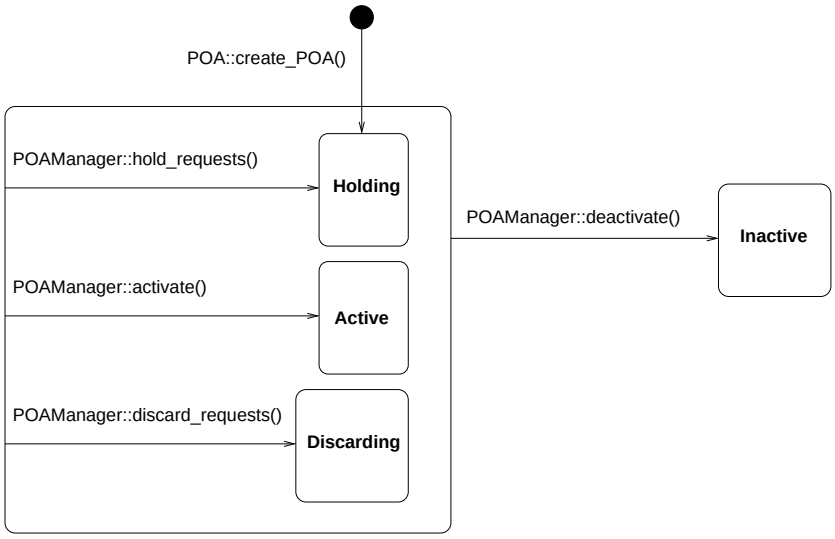

Figura 5: Stati del gestore del POA.

del riferimento come abbiamo fatto negli esempi visti finora. Questo servizio è definito da due specifiche, una per il servizio di naming di base, ed una per il servizio esteso.

6.1 Rappresentazioni dei riferimenti

Nelle sezioni precedenti abbiamo visto che un cliente può ottenere un riferimento CORBA (ovvero creare un proxy per un oggetto remoto ed ottenere un riferimento locale al proxy) passando alla funzione `string_to_object()` una rappresentazione testuale del riferimento. Questa funzione può accettare stringhe di caratteri in quattro formati diversi:

1. la rappresentazione testuale;
2. una URL che individua un file contenente la rappresentazione testuale, con protocollo `file:`, `ftp:` o `http:`;
3. una stringa in formato *corbaloc* (v. oltre);
4. una stringa in formato *corbaname* (v. oltre).

Il formato di una stringa corbaloc è il seguente:

```
corbaloc:<prot>:<vers>@<hostname>:<port>/<objectkey>
```

dove `prot` è il protocollo (per default è IIOP, il protocollo standard delle architetture CORBA), `vers` è il numero di versione, `hostname` e `port` sono l'indirizzo del server, e `objectkey` è l'identificatore dell'oggetto. Il protocollo, la versione e il port sono opzionali. Un esempio di riferimento rappresentato con un corbaloc è il seguente:

```
corbaloc::www.acme.com/Rocket
```

I protocolli possibili sono `iiop` e `rir`, abbreviazione di *resolve initial references*. Passando una stringa corbaloc, con protocollo `rir`, a `string_to_object()`, questa passa l'identificatore dell'oggetto a `resolve_initial_references()`.

Il formato corbaname verrà mostrato nella sezione sul servizio di naming esteso.

6.2 Il servizio di naming di base

La struttura dei nomi prevista dal servizio di naming è simile a quella dei nomi di file in un file system gerarchico. In un file system un nome completo è un *path*, cioè una sequenza di componenti, dei quali l'ultimo è il nome semplice di un file ordinario, e gli altri sono nomy di directory. Com'è noto, un directory può contenere sia dei file ordinari che altri directory. Nel servizio di naming un *contesto di naming* corrisponde a un directory e un oggetto corrisponde a un file ordinario.

La differenza fra un nome CORBA (nel servizio di naming di base) e un nome di un file system è che i nomi CORBA non avevano una rappresentazione testuale: un nome CORBA non è una stringa di caratteri, ma una struttura dati che viene costruita e usata nel programma, come vedremo fra poco. Il servio di naming esteso, che vedremo dopo, permette la rappresentazione testuale dei nomi CORBA.

Il servizio di naming è definito da un insieme di tipi e interfacce contenuti nel modulo IDL `CosNaming` (dove "`Cos`" sta per "Corba Service"). La struttura dei nomi viene specificata cosí:

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

A differenza che nei file system, un componente del nome è costituito da due stringhe, ma non è obbligatorio usare il campo `kind` di `NameComponent`, che può essere ignorato.

L'interfaccia `NamingContext` permette di creare e distruggere contesti di naming, e di inserirvi altri contesti e oggetti. Inoltre permette di *risolvere* un nome, cioè ottenere un riferimento all'oggetto individuato da un nome.

Si può anche elencare il contenuto di un contesto, con un'operazione `list()` che restituisce un iteratore di tipo `BindingIterator`.

6.3 Esempio

Immaginiamo di avere un'applicazione chiamata "*Climate Control System*" che permette il controllo remoto di un complesso impianto di condizionamento d'aria. L'oggetto principale di questa applicazione è un `Controller` a cui si collegano i clienti. Vogliamo assegnare a questo oggetto il nome "`Controller`" nel contesto di naming "`CCS`".

Il server deve istanziare e registrare un servente:

```
CCS::Controller_impl ctrl_servt;
CCS::Controller_var ctrl_ref = ctrl_servt._this();
```

Quindi deve ottenere un riferimento al contesto iniziale del servizio di naming:

```
CORBA::Object_var obj_ns =
        orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc =
        CosNaming::NamingContext::_narrow(obj_ns);
```

Naturalmente l'ORB deve essere stato configurato in modo da poter contattare un servizio di naming, che potrebbe trovarsi su un nodo diverso.

Ora si può creare il contesto `CCS` entro il contesto iniziale `inc`. Prima si costruisce il nome `n` (una sequenza con un solo elemento) contenente la stringa `CCS`, poi si crea il nuovo contesto `nc` "legato" al nome:

```
CosNaming::Name n;
n.length(1);
n[0].id = CORBA::String_dup("CCS");
CosNaming::NamingContext_var nc = inc->bind_new_context(n);
```

Poi si aggiunge al nome un componente con la stringa `Controller`, e si lega il nome al riferimento `ctrl_ref`:

```
n.length(2);
n[1].id = CORBA::String_dup("Controller");
inc->rebind(n, ctrl_ref);
```

Un cliente può quindi ottenere un riferimento all'oggetto attraverso il servizio di naming. A questo scopo deve ottenere un riferimento al contesto iniziale del servizio di naming, costruire il nome dell'oggetto, e chiedere al contesto iniziale di risolvere il nome:

```
CORBA::Object_var obj_ns =
        orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc =
        CosNaming::NamingContext::_narrow(obj_ns);
CosNaming::Name n;
n.length(2);
n[0].id = CORBA::String_dup("CCS");
n[1].id = CORBA::String_dup("Controller");
CORBA::Object_var obj_ctrl = inc->resolve(n);
CCS::Controller_var ctrl_ref = CCS::Controller::_narrow(obj_ctrl);
```

Naturalmente l'ORB del cliente deve essere configurato in modo da contattare il servizio di naming usato dal server, direttamente o attraverso dei servizi intermedi.

6.4 Il servizio di naming esteso

Il servizio di naming esteso definisce una rappresentazione testuale (`StringName`) dei nomi CORBA. Questa si ottiene scrivendo di seguito i nomi dei contesti, separati da barre, col nome dell'oggetto in fondo, analogamente ai path dei file nel sistema Unix. Se il nome di un contesto ha un campo `kind`, questo viene separato con un punto dal campo `id`. Il nome CORBA visto nell'esempio precedente viene rappresentato come:

```
/CCS/Controller
```

La specifica del servizio di naming esteso definisce delle operazioni di conversione fra i nomi CORBA e le loro rappresentazioni testuali, e un'operazione di risoluzione:

```
interface NamingContextExt : NamingContext {
    StringName to_string(in Name n);
    Name to_name(in StringName sn);
    Object resolve_str(in StringName sn);
    // ...
};
```

La funzione `string_to_object()` accetta stringhe in formato corbaname, che è simile al corbaloc ma prevede che l'identificatore dell'oggetto venga seguito da uno `StringName`. L'identificatore dell'oggetto deve riferirsi al contesto iniziale di un servizio di naming, e infatti il suo valore di default è `NameService`. Questo è il formato corbaname:

```
corbaname:<prot>:<vers>@<hostname>:<port>/<objectkey>#<stringname>
```

7 Il Servizio Eventi

Il Servizio Eventi, costituito da un insieme di interfacce IDL, definisce un meccanismo che permette di realizzare architetture distribuite i cui componenti siano accoppiati meno strettamente che nel modello client/server visto finora.

Immaginiamo, per esempio, di voler aggiungere un sistema di monitoraggio remoto al sistema CCS descritto nella Sez. A. Il sistema di monitoraggio dovrebbe visualizzare (magari per mezzo di un'interfaccia grafica) la situazione attuale dell'impianto, aggiornandola quando ci sono cambiamenti nella temperatura ambiente o nelle impostazioni dei termostati. Usando il modello client/server puro, il sistema di monitoraggio verrebbe realizzato come un cliente che periodicamente interroga tutti i dispositivi per mezzo delle operazioni previste dalle loro interfacce, applicando il noto meccanismo di *polling*. Questo metodo impone un forte carico sul cliente, sul server e sulla rete, ed è poco scalabile nel caso che ci siano piú sistemi di monitoraggio attivi contemporaneamente. Il carico e la complessità sarebbero ovviamente ancora piú grandi se ogni cliente dovesse monitorare piú di un impianto.

Si può ottenere una realizzazione migliore usando il pattern *Observer* [3]: ogni sistema di monitoraggio attivo si registra presso il sistema CCS e resta in attesa di comunicazioni. Il sistema CCS comunica ai sistemi di monitoraggio i cambiamenti di temperatura o di impostazioni man mano che si verificano. Per applicare questo modello occorre che l'interfaccia CCS venga estesa con

operazioni che permettano ai monitor di registrarsi. L'implementazione deve essere modificata in modo da rilevare i cambiamenti di stato dell'impianto e comunicarli ai sistemi di monitoraggio. Questi ultimi, a loro volta, devono offrire un'interfaccia con un'operazione che permetta al CCS di notificare i cambiamenti. Osserviamo che in questo pattern il cliente e il server si scambiano i ruoli: il sistema CCS diventa un cliente del sistema di monitoraggio, poiché deve invocare una sua operazione per notificare un cambiamento di stato. In questo modello la distinzione fra *produttore* e *consumatore* di eventi (cambiamenti di stato) è piú utile di quella fra cliente e server.

Con l'applicazione del pattern *Observer* si fa un migliore uso della rete e si impone un minor carico sul sistema di monitoraggio, però il CCS deve gestire la comunicazione con i monitor, e il carico dovuto a questo compito aumenta col numero di monitor attivi, per cui si pone di nuovo il problema della scalabilità.

La soluzione che proponiamo, e che porta al Servizio Eventi, consiste nel combinare il pattern *Observer* col pattern *Mediator* [3]: fra produttori e consumatori si inserisce un componente intermedio, un *canale di eventi* che provvede a smistare le comunicazioni. I consumatori di eventi non si registrano direttamente presso i produttori, ma presso il canale, mentre i produttori non inviano gli aggiornamenti direttamente ai consumatori, ma al canale.

7.1 Modelli di comunicazione

Il modello di comunicazione fra produttori e consumatori di eventi visto nella sezione precedenti è quello piú comunemente usato, ed è noto come modello *push/push*: il produttore "spinge" (*push*) l'informazione verso il canale, che a sua volta la spinge verso il consumatore, come mostra la Fig. 6. La figura successiva (Fig. 7) mostra una possibile interazione in cui sono coinvolti due produttori e tre consumatori. Osserviamo che in questo modello i produttori sono clienti del canale e il canale è cliente dei consumatori.

Un altro modello possibile è il *pull/pull*, in cui il consumatore "estrae" (*pull*) informazione dal canale, che a sua volta la deve estrarre dal produttore, come mostrato in Fig. 8. In questo modello il consumatore è cliente del canale, che è cliente del produttore. Anche in questo modello si possono avere piú produttori e piú consumatori.

Possiamo poi considerare i due modelli ibridi *push/pull* e *pull/push*. Nel primo il produttore è cliente del canale, che è server del consumatore, nel

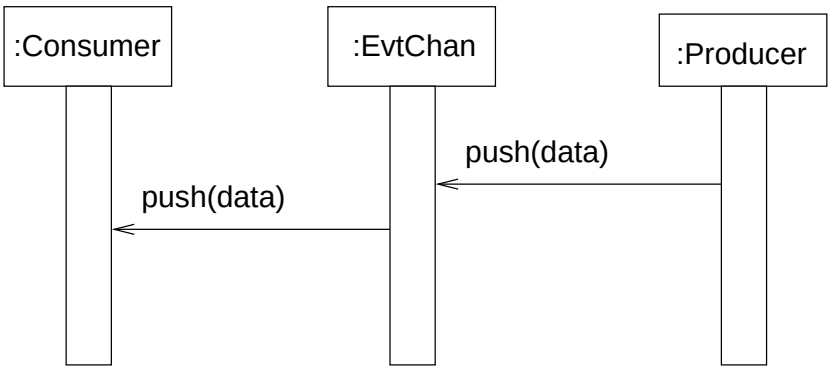

Figura 6: Modello *push/push* (1).

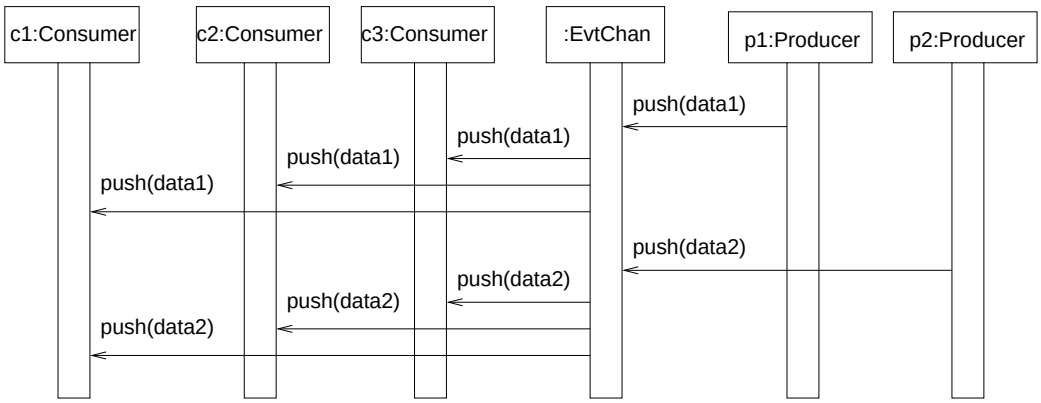

Figura 7: Modello *push/push* (2).

secondo il produttore è server del canale e il canale è cliente del consumatore. Osserviamo che in questo caso il canale, essendo cliente di tutti gli altri componenti, assume il ruolo di coordinatore dell'interazione.

Un'applicazione distribuita può applicare integralmente uno dei quattro modelli sú esposti, o anche applicare dei modelli misti, in cui alcuni produttori interagiscono con alcuni consumatori secondo un certo modello mentre altri produttori e consumatori interagiscono secondo altri modelli.

7.2 Il Servizio Eventi

La specifica IDL del Servizio Eventi (uno standard OMG) è costituita da due moduli, **CosEventComm** e **CosEventChannelAdmin**. Il modulo **CosEvent-**

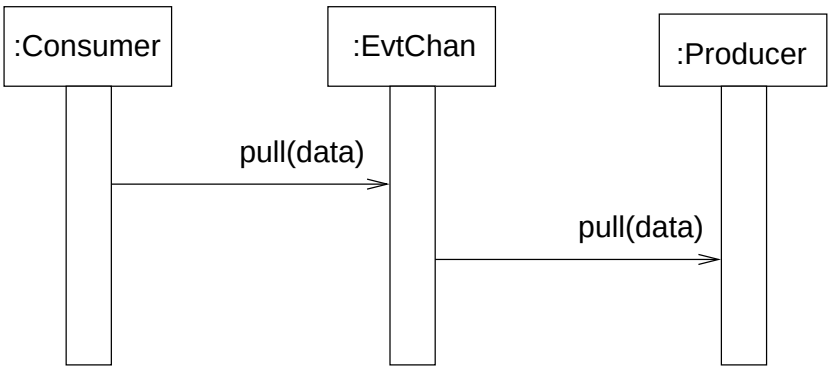

Figura 8: Modello *pull/pull*.

Comm contiene le interfacce che devono essere implementate dai produttori e dai consumatori, sia di tipo *push* che di tipo *pull*. Il modulo **CosEventChannelAdmin** contiene le interfacce implementate dal canale; alcune di queste servono a collegare produttori e consumatori al canale, altre servono a trasmettere informazioni attraverso il canale.

Nel modulo **CosEventComm** sono definite le seguenti interfacce di tipo *push*:

```
interface PushConsumer {
    void push(in any data) raises(Disconnected);
    void disconnect_push_consumer();
};

interface PushSupplier {
    void disconnect_push_supplier();
};
```

L'interfaccia **PushConsumer**, implementato dai consumatori di tipo *push*, comprende due operazioni: `push()`, chiamata dal canale con un argomento `data` di tipo `any`, destinato a contenere le informazioni relative all'evento, e `disconnect_push_consumer()`, chiamata da un produttore per interrompere la connessione col consumatore.

L'interfaccia **PushSupplier**, implementato dai produttori di tipo *push*, comprende solo l'operazione `disconnect_push_supplier()`, chiamata da un consumatore. Ovviamente un produttore *push* non offre alcuna operazione per la trasmissione di informazioni, poiché nel modello *push* il produttore è un cliente, cioè un oggetto che *invoca* operazioni su altri oggetti.

L'eccezione `Disconnected`, anch'essa dichiarata nel modulo **CosEventComm**, viene sollevata quando si invoca un'operazione su un oggetto non ancora collegato al canale.

Le interfacce di tipo *pull* dichiarate nel modulo **CosEventComm** sono le seguenti:

```
interface PullConsumer {
    void disconnect_pull_consumer();
};

interface PullSupplier {
    any pull() raises(Disconnected);
    any try_pull(out boolean has_event) raises(Disconnected);
    void disconnect_pull_supplier();
};
```

L'interfaccia **PullSupplier**, implementata dai produttori di tipo *pull*, ha due operazioni per la trasmissione di informazioni: `pull()`, che blocca il chiamante in attesa che un evento sia disponibile, e `try_pull()`, che restituisce il controllo al chiamante immediatamente, assegnando all'argomento `has_event` il valore `true` se un evento è disponibile (e passando le informazioni relative come valore restituito), o il valore `false` altrimenti.

Il modulo **CosEventChannelAdmin** contiene delle interfacce *proxy* in cui sono dichiarate operazioni di connessione, oltre a quelle di trasmissione di informazioni, e delle interfacce di configurazione. Il canale svolge la sua funzione di intermediario fra produttori e consumatori di eventi offrendo a ciascuno di essi un *proxy* del tipo complementare: un produttore vede il canale come un consumatore e un consumatore vede il canale come un produttore. Vediamo quindi che il Servizio Eventi è modellato sul pattern *Proxy* [3], oltre che sui pattern *Observer* e *Mediator* già citati. Per ottenere i riferimenti ai *proxy* richiesti, produttori e consumatori si servono di operazioni appartenenti alle interfacce di configurazione del canale.

Le interfacce *proxy* sono le seguenti:

```
interface ProxyPushConsumer : CosEventComm::PushConsumer {
    void connect_push_supplier(
                in CosEventComm::PushSupplier supplier)
          raises(AlreadyConnected);
};

interface ProxyPushSupplier : CosEventComm::PushSupplier {
    void connect_push_consumer(
                in CosEventComm::PushConsumer consumer)
```

```
        raises(AlreadyConnected, TypeError);
};

interface ProxyPullConsumer : CosEventComm::PullConsumer {
    void connect_pull_supplier(
                in CosEventComm::PullSupplier supplier)
        raises(AlreadyConnected, TypeError);
};

interface ProxyPullSupplier : CosEventComm::PullSupplier {
    void connect_pull_consumer(
                in CosEventComm::PullConsumer consumer)
        raises(AlreadyConnected);
};
```

Possiamo vedere che c'è un'interfaccia *proxy* per ciascun tipo di produttore o consumatore, e ciascuna interfaccia *proxy* deriva da quella del corrispondente produttore o consumatore. Le interfacce *proxy* aggiungono alle rispettive classi base delle operazioni per la connessione al canale. Piú precisamente:

- Un produttore *push* invoca l'operazione `connect_push_supplier()` su un oggetto di tipo **ProxyPushConsumer** per collegarsi al canale. Se il produttore deve essere notificato di una eventuale sconnessione, bisogna passare a `connect_push_supplier()` un riferimento a un oggetto di tipo **PushSupplier**, altrimenti si passa un riferimento nullo.
- Un consumatore *push* implementa l'interfaccia **PushConsumer** e invoca l'operazione `connect_push_consumer()` su un oggetto di tipo **ProxyPushSupplier** per collegarsi al canale. A questa operazione deve passare un riferimento a se stesso, di tipo **PushConsumer**.
- Un produttore *pull* implementa l'interfaccia **PullSupplier** e invoca l'operazione `connect_pull_supplier()` su un oggetto di tipo **ProxyPullConsumer** per collegarsi al canale. A questa operazione deve passare un riferimento a se stesso, di tipo **PullSupplier**.
- Un consumatore *pull* invoca l'operazione `connect_pull_consumer()` su un oggetto di tipo **ProxyPullSupplier** per collegarsi al canale. Se il consumatore deve essere notificato di una eventuale sconnessione, bisogna passare a `connect_pull_consumer()` un riferimento a un oggetto di tipo **PullConsumer**, altrimenti si passa un riferimento nullo.

Le eccezioni `AlreadyConnected` e `TypeError` sono dichiarate nel modulo **CosEventChannelAdmin**.

Le restanti interfacce di **CosEventChannelAdmin** definiscono operazioni per mezzo delle quali si possono ottenere riferimenti a oggetti *proxy*. Questo avviene in due tempi:

1. Un consumatore o un produttore richiede ad un oggetto **EventChannel** un riferimento a un oggetto **ConsumerAdmin** o, rispettivamente, **SupplierAdmin**;
2. ottenuto tale riferimento, il consumatore o produttore lo usa per ottenere un riferimento a un oggetto *proxy* di tipo adeguato.

Seguono le dichiarazioni delle interfacce sú descritte:

```
interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    ProxyPullSupplier obtain_pull_supplier();
};
interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ProxyPullConsumer obtain_pull_consumer();
};
interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
};
```

L'operazione `destroy()` distrugge il canale, insieme agli oggetti di tipo **ConsumerAdmin** e **SupplierAdmin**, e agli oggetti *proxy*.

La Fig. 9 mostra le interfacce che fanno parte del Servizio Eventi. Sono stati omessi i parametri di alcune operazioni.

La Fig. 10 mostra una tipica interazione, secondo il modello *push/push*, fra produttori, consumatori e canale, rappresentando l'infrastruttura di comunicazione con un unico pseudo-oggetto di tipo **ORB**. Precisiamo che il produttore di eventi, avendo sempre il ruolo di cliente, non deve necessariamente implementare l'interfaccia **PushSupplier**. Implementare tale interfaccia è necessario solo se il produttore deve essere sconnesso. Osserviamo inoltre che il diagramma non mostra i passi iniziali relativi all'inizializzazione dell'ORB e alla creazione di riferimenti al Servizio Eventi.

Si possono fare scelte alternative rispetto a quelle mostrate nel diagramma: per esempio, l'operazione `connect_push_consumer()` può essere invocata da un altro oggetto (per esempio nel corpo del programma principale in cui viene istanziato il consumatore, programma che consideriamo come il

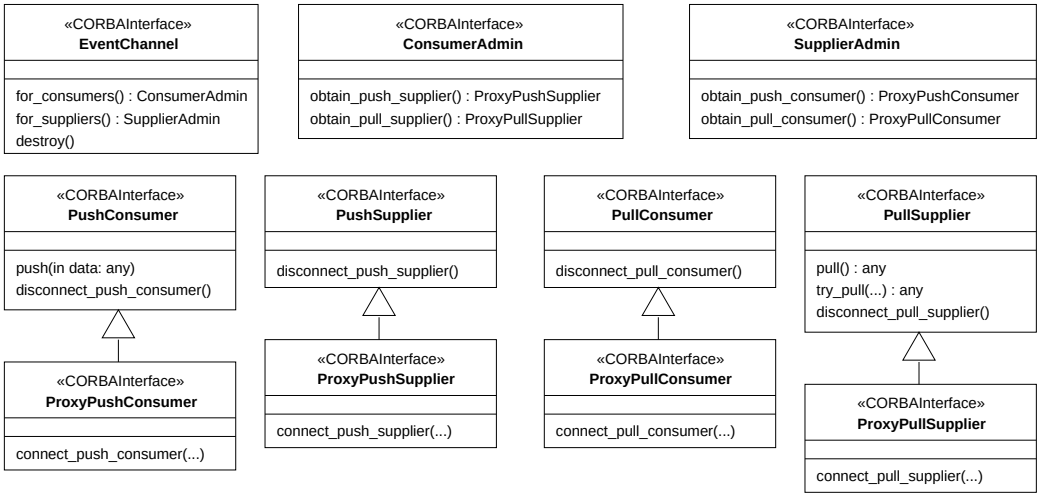

Figura 9: Interfacce per il Servizio Eventi.

flusso di controllo di un oggetto attivo). L'importante è che l'argomento di tale operazione sia un riferimento a un oggetto di tipo **PushConsumer**.

7.3 Esempio

Il seguente esempio mostra schematicamente un fornitore di eventi. Un oggetto `Sensor` produce un evento ogni volta che cambia il valore osservato dal sensore.

```
class Sensor_impl : public virtual POA_Sensor {
    CosEventChannelAdmin::ProxyPushConsumer_var consumer_;
    void value_changed(int val)
    {
        CORBA::Any a <<= val; consumer_->push(a);
    }
public:
    Sensor_impl(CosEventChannelAdmin::ProxyPushConsumer_var c)
        : consumer_(c) {/*...*/}
    // ...
};

int
main(int argc, char* argv[])
{
    CosEventChannelAdmin::EventChannel_var evch_;
    CosEventChannelAdmin::SupplierAdmin_var supadmin_;
    evch_ = string_to_object("file:/evtchannel_ior.txt");
```

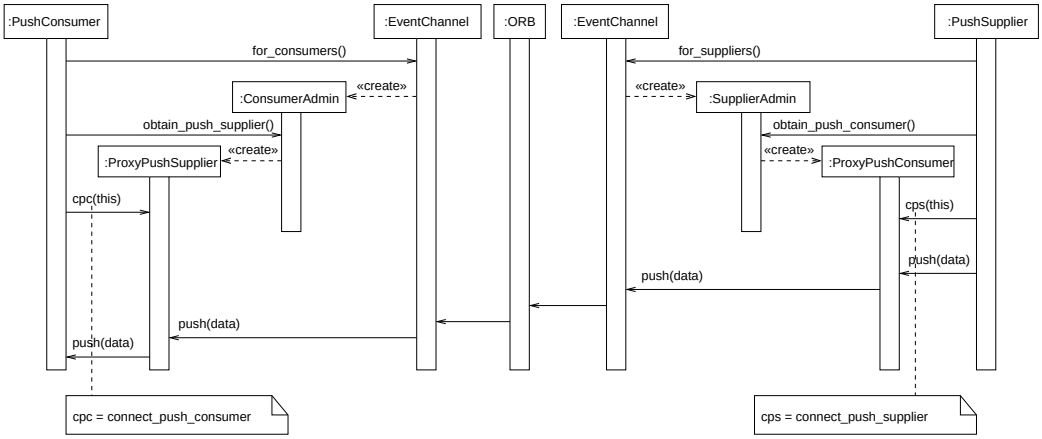

Figura 10: Interazione nel modello *push/push*.

```
    supadmin_ = evch_->for_suppliers();
    CosEventChannelAdmin::ProxyPushConsumer_var pxy
        = supadmin_->obtain_push_consumer();
    CosEventComm::PushSupplier_var nil
         = CosEventComm::PushSupplier::_nil();

    Sensor_impl(pxy) sensor(pxy);
    // ...
}
```

8 Il *CORBA Component Model*

Nella versione 3.0 delle specifiche CORBA è stato introdotto il *modello dei componenti* CORBA (*CORBA Component Model*, CCM). Come illustrato nella parte del corso relativa alle tecniche di progetto orientate agli oggetti, un componente è un modulo di cui sono definite esplicitamente sia le interfacce offerte che quelle richieste, e che è stato progettato rispettando certe convenzioni che ne favoriscono l'intercambiabilità con altri componenti funzionalmente equivalenti, ma implementati diversamente. Tali convenzioni riguardano, per esempio, il ciclo di vita del componente, cioè le operazioni di creazione e distruzione, la definizione di operazioni comuni come l'accesso ad attributi o proprietà del componente, la sua configurazione, le sue interazioni con l'embiente di esecuzione ed i servizi esterni, le informazioni necessarie per la sua istallazione (*deployment*), ed altri aspetti ancora.

Il CCM definisce le convenzioni che devono essere rispettate da un'architettura distribuita orientata ai componenti e basata sull'infrastruttura CORBA. Questo modello consiste dei seguenti elementi:

- Un modello dei componenti, con le estensioni del linguaggio IDL richieste per la loro specifica.
- Un modello dell'ambiente di esecuzione dei componenti, detto *contenitore* (*container*).
- Un modello (*framework*) di implementazione, con un linguaggio per la descrizione strutturale delle implementazioni (*Component Implementation Definition Language*, CIDL).
- Un modello per la configurazione e l'istallazione di applicazioni basate su componenti.

Chiameremo IDL3 il linguaggio IDL esteso ai componenti, e IDL2 il suo sottoinsieme precedente a tale estensione.

8.1 Struttura dei componenti

Un componente è definito dalle interfacce che implementa, dette *port*, e dai suoi *attributi*. Questi ultimi, come già visto per le interfacce in IDL, sono una forma sintetica per dichiarare operazioni di lettura e scrittura di valori che rappresentano proprietà del componente. Generalmente gli attributi rappresentano parametri di configurazione, che vengono inizializzati prima dell'uso del componente e vengono modificati raramente.

Si distinguono i seguenti tipi di port:

Interfaccia equivalente: l'interfaccia principale (eventualmente l'unica) del componente.
Facet: un'interfaccia offerta, distinta da quella principale.
Receptacle: un'interfaccia richiesta.
Sorgente di eventi: un'interfaccia richiesta, per inviare messaggi asincroni (*event source*).
Consumatore di eventi: un'interfaccia che riceve messaggi asincroni (*event sink*).

A ciascun port è associato un riferimento CORBA, per cui un cliente usa un componente remoto come un normale oggetto CORBA. Il cliente

inizialmente accede ad un componente per mezzo di un riferimento alla sua interfaccia equivalente. In questa interfaccia sono definite implicitamente delle operazioni che permettono al cliente *navigare* nel componente, cioè di ottenere riferimenti agli altri eventuali port del componente.

L'interfaccia equivalente può offrire anche delle operazioni definite dallo sviluppatore, che generalmente svolgono funzioni ausiliarie rispetto allo scopo del componente, mentre le operazioni principali sono definite nelle interfacce associate ai facet. Per esempio, se un componente dovesse gestire un sensore di temperatura, l'interfaccia equivalente potrebbe offrire le operazioni per accendere e spengere il sensore (`start()` e `stop()`), e un facet potrebbe offrire l'operazione che legge la temperatura (`get_temp()`). In un caso semplice come questo, però, si può assegnare all'interfaccia equivalente anche l'operazione di lettura.

Per ogni tipo di componente bisogna definire un componente ausiliario, detto *home*, con la funzione di creare istanze del tipo di componente in questione.

8.2 Definizione e implementazione di un componente

Il seguente esempio mostra la definizione in IDL3 di un semplice componente, che offre una sola operazione, dichiarata per comodità nell'interfaccia equivalente:

```
interface Hello {
    void sayHello ();
};

component HelloWorld supports Hello {
    attribute string message;
};

home HelloHome manages HelloWorld {
    attribute string initial_message;
};
```

Per implementare il componente, bisogna innanzitutto usare il compilatore IDL3 e un altro strumento, che chiameremo *compilatore CCM*. Il compilatore IDL3 genera il codice delle classi scheletro e dei proxy per l'interfaccia `Hello` e per le interfacce corrispondenti alle dichiarazioni del componente `HelloWorld` e della home `HelloHome`. Ricordiamo che i componenti e le rispettive home verranno usati come semplici oggetti CORBA, quindi le loro

definizioni possono essere tradotte in "IDL2 equivalente", ottenendo interfacce in cui si trovano sia le operazioni dichiarate esplicitamente dal progettista, sia le numerose operazioni implicite previste dal CCM. Il compilatore CCM, a sua volta, produce il codice necessario per l'integrazione dei componenti nell'ambiente di esecuzione (ovviamente dal lato server dell'applicazione), fra cui degli "scheletri di componente" da usare come base per l'implementazione dei componenti e delle rispettive home.

Nel nostro caso, usiamo questi comandi:

```
% idl hello.idl
% ccm --standalone hello.idl
```

dove `ccm` è il compilatore CCM[2], che produce i file `hello_ccm.h` e `hello_ccm.cc`, contenenti fra l'altro la definizione dei due "scheletri di componente" `CCM_HelloWorld` e `CCM_HelloHome`. Inoltre, l'opzione `--standalone` ha causato la generazione di un programma principale che costituisce un contenitore del tipo piú semplice, cioè un programma eseguibile compilato insieme ai componenti. Altre opzioni permettono invece di includere i componenti in librerie dinamiche che vengono caricate da un contenitore compilato separatamente.

Segue il codice scritto dal progettista per l'implementazione:

```
#include "hello.h"

using namespace std;

class HelloWorld_impl : virtual public CCM_HelloWorld {
    CORBA::String_var _message;
public:
    HelloWorld_impl(const char* initial)
    {
        _message = initial;
    }
    void sayHello()
    {
        cout << _message << endl;
    }
    void message(const char* val)
    {
        _message = CORBA::string_dup(val);
    }
```

[2] Questo programma è stato provato col compilatore `mico-ccm` dell'ambiente di sviluppo MicoCCM (`www.fpx.de/MicoCCM/`), ma, per semplicità di esposizione, gli esempi mostrati sono leggermente diversi da quelli reali.

```
    char* message()
    {
        return CORBA::string_dup(_message);
    }
};

class HelloHome_impl : virtual public CCM_HelloHome {
    CORBA::String_var _initial_message;
public:
    HelloHome_impl()
    {
        _initial_message = CORBA::string_dup("Hello World");
    }
    Components::EnterpriseComponent_ptr create()
    {
        return new HelloWorld_impl(_initial_message);
    }
    void initial_message(const char* val)
    {
        _initial_message = CORBA::string_dup(val);
    }
    char * initial_message()
    {
        return CORBA::string_dup(_initial_message);
    }
};

extern "C" {
    Components::HomeExecutorBase_ptr create_HelloHome ()
    {
        return new HelloHome_impl;
    }
}
```

Il codice scritto per implementare un cliente è simile a quello che si scriverebbe per accedere a un semplice oggetto CORBA:

```
#include <CORBA.h>
#include <coss/CosNaming.h>
#include "hello.h"

int
main(int argc, char* argv[])
{
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj
            = orb->resolve_initial_references("NameService");
    CosNaming::NamingContextExt_var nc
            = CosNaming::NamingContextExt::_narrow(obj);
    assert (!CORBA::is_nil(nc));
    obj = nc->resolve_str("HelloHome");
```

```
    assert(!CORBA::is_nil(obj));
    HelloHome_var hh = HelloHome::_narrow(obj);

    HelloWorld_var hw = hh->create();
    hw->message("Hello World");
    hw->sayHello();
    hw->remove();

    return 0;
}
```

Il codice del server sarebbe molto piú complesso, ma, come abbiamo visto, viene generato automaticamente dal compilatore CCM.

Il server ed un cliente possono essere compilati con i seguenti comandi:

```
% g++ -o server hello_impl.cc hello_ccm.cc -lorb -lccm
% g++ -o client client.cc hello.cc -lorb -lccm
```

ed eseguiti con i seguenti comandi:

```
./server -ORBInitRef NameService=file://nsd.ior &
./client -ORBInitRef NameService=file://nsd.ior
```

dove l'opzione `-ORBInitRef` fornisce il riferimento al servizio di naming, contenuto nel file `nsd.ior`.

8.3 L'ambiente di esecuzione

Un componente CORBA viene implementato da un *esecutore*, analogo al *servant* degli oggetti CORBA. Gli esecutori "girano" in un contenitore che ne gestisce il ciclo di vita e l'associazione con i riferimenti, e che fornisce l'accesso ai servizi fondamentali.

Come abbiamo visto, nel caso dei serventi di oggetti CORBA la gestione del ciclo di vita e dell'associazione con i riferimenti è affidata al POA. Il contenitore di componenti fa da intermediario fra gli esecutori ed il POA. In particolare, il CCM permette di evitare la configurazione esplicita del POA, poichè lo sviluppatore di componenti deve soltanto scegliere fra pochi modelli di uso e di esecuzione per i componenti e per il contenitore. Ciascuno di questi modelli corrisponde ad una certa combinazione di politiche del POA, che vengono impostate automaticamente dagli strumenti di sviluppo.

L'accesso ai servizi avviene per mezzo di alcune interfacce standardizzate fra il contenitore e gli esecutori, che vedremo piú oltre. Il contenitore permette l'accesso ai servizi di transazioni, di notifica (un servizio eventi semplificato), di persistenza e di sicurezza.

8.3.1 Il modello di programmazione del contenitore

L'interazione fra contenitore ed esecutore viene specificata ad alto livello per mezzo di alcune interfacce (definite in IDL) fra contenitore e componente. Le interfacce offerte dal contenitore al componente sono dette *interne*, quelle offerte dal componente al contenitore sono dette *callback*.

La politica di gestione degli esecutori viene determinata dalla *categoria* dei componenti. Questa, a sua volta, è la combinazione di due parametri, che sono il *tipo del contenitore* ed il *modello d'uso* del componente.

Il tipo di contenitore specifica la persistenza dei componenti: *session* per componenti transitori, *entity* per componenti persistenti.

Il modello d'uso specifica il tipo di interazione con i clienti:

stateless: non vengono mantenute informazioni di stato del componente fra un'invocazione e l'altra;
conversational: le informazioni di stato vengono mantenute per la durata di una serie di invocazioni da parte di uno stesso cliente;
durable: le informazioni di stato vengono mantenute indefinitamente.

A ciascun modello d'uso corrisponde una combinazione delle politiche del POA di Lifespan e IdUniqueness, come mostrato dalla seguente tabella:

Modello d'uso	Politiche
stateless	TRANSIENT, MULTIPLE_ID
conversational	TRANSIENT, UNIQUE_ID
durable	PERSISTENT, UNIQUE_ID

Un componente durevole può essere *keyful*, cioè avere una chiave primaria che lo identifica, o *keyless*, altrimenti.

La tabella successiva mostra la definizione della categoria del componente in base ai parametri già visti:

Categoria	tipo di contenitore	modello d'uso
service	session	stateless
session	session	conversational
process	entity	durable (keyless)
entity	entity	durable (keyful)

8.3.2 Interfacce interne

Le interfacce interne derivano da `CCMContext`, che definisce operazioni comuni ai due tipi di contenitore:

```
local interface CCMContext {
    Principal get_caller_principal();
    CCMHome get_CCM_home();
    // ...
};
```

L'operazione `get_caller_principal()` restituisce le credenziali dell'utente che ha richiesto i servizi del componente. L'operazione `get_CCM_home()` permette al componente di accedere alla propria home.

I contenitori di tipo session implementano l'interfaccia `SessionContext`:

```
local interface SessionContext : CCMContext {
    Obiect get_CCM_object() raises(IllegalState);
    // ...
};
```

L'operazione `get_CCM_object()` restituisce il riferimento usato per chiamare il componente, che permette di sapere attraverso quale facet è stato chiamato.

I contenitori di tipo entity implementano l'interfaccia `EntityContext`:

```
local interface EntityContext : CCMContext {
    Obiect get_CCM_object() raises(IllegalState);
    PrimaryKeyBase get_primary_key() raises(IllegalState);
};
```

L'operazione `get_primary_key()` restituisce la chiave primaria del componente.

8.3.3 Interfacce callback

Anche le interfacce callback sono diverse per i due tipi di contenitori e derivano da un'interfaccia comune, che in questo caso è vuota:

```
local interface EnterpriseComponent { };
```

I contenitori di tipo session richiedono l'interfaccia `SessionComponent`:

```
local interface SessionComponent : EnterpriseComponent {
    void set_session_context(in SessionContext ctx)
                                        raises(CCMException);
    void ccm_activate() raises(CCMException);
    void ccm_passivate() raises(CCMException);
    void ccm_remove() raises(CCMException);
};
```

L'operazione `set_session_context()` fornisce al componente un riferimento al suo contesto. Le altre tre operazioni vengono chiamate dal contenitore, rispettivamente, prima dell'attivazione, della disattivazione e della distruzione del componente.

I contenitori di tipo session richiedono l'interfaccia `EntityComponent`:

```
local interface EntityComponent : EnterpriseComponent {
    void set_entity_context(in EntityContext ctx)
                                        raises(CCMException);
    void ccm_activate() raises(CCMException);
    void ccm_passivate() raises(CCMException);
    void ccm_remove() raises(CCMException);
    void ccm_load() raises(CCMException);
    void ccm_store() raises(CCMException);
};
```

Le operazioni `ccm_load()` e `ccm_store()` servono, rispettivamente, a recuperare lo stato del componente dalla memoria persistente e a memorizzarlo.

8.4 Un esempio

Questo esempio è tratto da un tutorial sull'ambiente *CIAO* (*Component-Integrated ACE ORB*) [2].

Consideriamo un sistema costituito da un "distributore" (**StockDistributor**) che periodicamente trasmette i valori aggiornati delle quotazioni di titoli azionari ad uno o piú "agenti di borsa" (**StockBroker**). Ciascuno di questi agenti si occupa solo delle azioni di alcune società, quindi la trasmissione di informazioni funziona in due passi: prima il distributore trasmette a tutti gli agenti i simboli delle società su cui sono disponibili informazioni, poi ciascun agente chiede al distributore di fornirgli le informazioni sulle azioni a cui è interessato. La Fig. 11 mostra la struttura del sistema. Il componente **StockDistributor** pubblica eventi di tipo **StockName** ed offre un'interfaccia di tipo **StockQuoter**. Inoltre, l'interfaccia principale di **StockDistributor** deriva dall'interfaccia **Trigger** che serve a far partire e arrestare il componente.

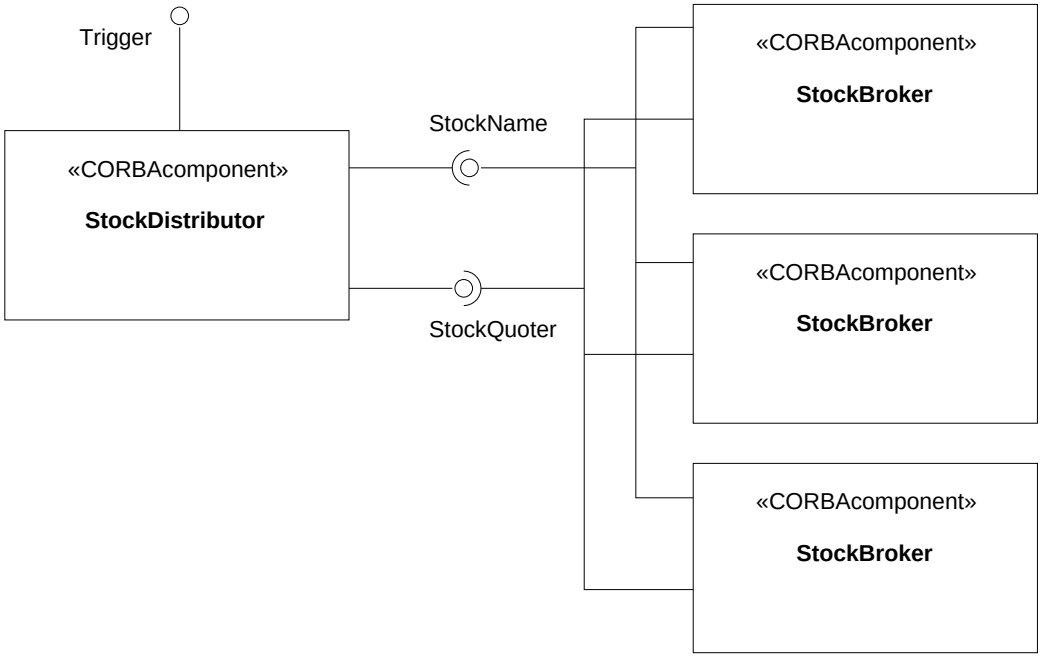

Figura 11: Esempio *StockBroker* (1).

La Fig. 12 mostra in dettaglio le varie definizioni necessarie.

Nel componente **StockDistributor** la sorgente di eventi è stata chiamata notifier_out e il *facet* è stato chiamato quoter_info_out, e inoltre si è specificato che il componente ha un parametro di configurazione rate che fissa la frequenza con cui avvengono gli aggiornamenti. Nel componente **StockBroker** i port complementari a quelli di **StockDistributor** si chiamano, rispettivamente, notifier_in e quoter_info_in.

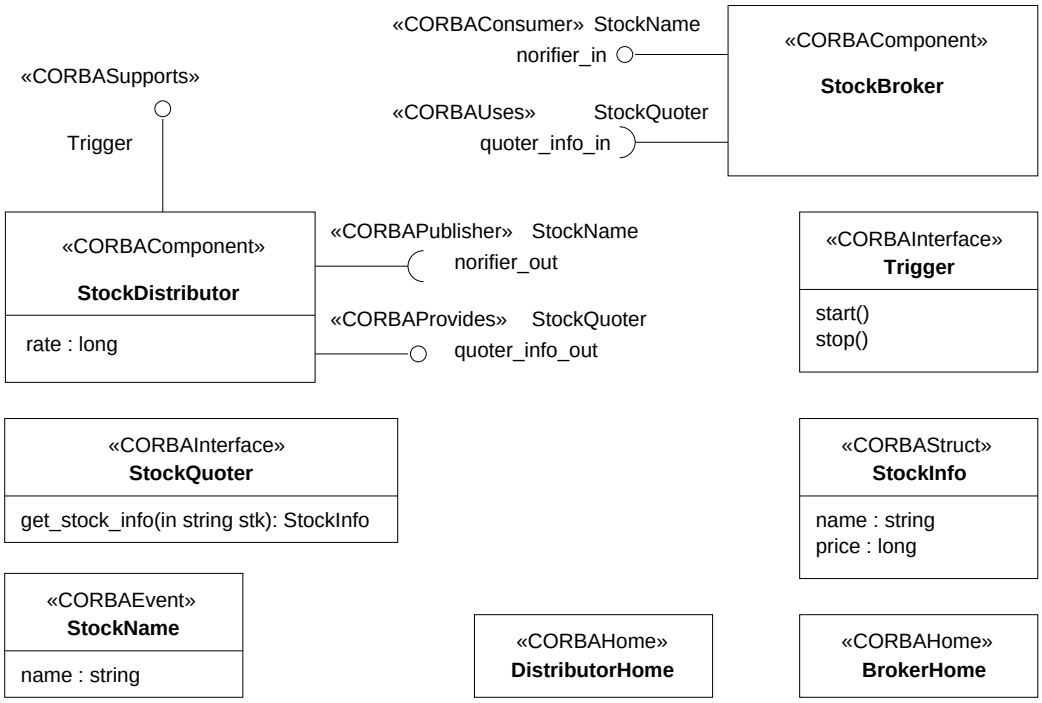

Figura 12: Esempio *StockBroker* (2).

L'interfaccia **Trigger** contiene le operazioni per avviare ad arrestare il componente, mentre l'interfaccia **StockQuoter** contiene l'operazione get_stock_info() che restituisce le informazioni relative alle azioni della società specificata dal parametro stk. Queste informazioni sono definite dalla struttura **StockInfo**. Il tipo degli eventi pubblicati dal distributore, definito da **StockName** ha un campo che contiene il nome della società quotata.

Infine, sono mostrate le dichiarazioni delle home dei due componenti.

Le corrispondenti dichiarazioni in IDL3 sono:

```
module Stock {
struct StockInfo {
    string name;
    long price;
};

interface Trigger {
    void start();
    void stop();
};
```

```
interface StockQuoter {
    StockInfo get_stock_info(in string stk);
};

eventtype StockName {
    public string name;
};

component StockBroker {
    consumes StockName notifier_in;
    uses StockQuoter quoter_info_in;
};

component StockDistributor supports Trigger {
    publishes StockName notifier_out;
    provides StockQuoter quoter_info_out;
    attribute long notification_rate;
};

home StockBrokerHome manages StockBroker {};
home StockDistributorHome manages StockDistributor {};
};
```

Possiamo ora osservare come le dichiarazioni in IDL3 vengano tradotte in dichiarazioni equivalenti in IDL2. Per esempio, la dichiarazione di `StockName` genera le seguenti dichiarazioni IDL2:

```
valuetype StockName : Components::EventBase {
    public string name;
};
interface StockNameConsumer : Components::EventConsumerBase {
    void push_StockName(in StockName the_stockname);
};
```

La definizione di `StockNameConsumer` specifica l'interfaccia che deve essere offerta dai componenti che devono consumare eventi di tipo `StockName`. Nel CCM gli eventi vengono trasmessi per mezzo del servizio di notifica, un servizio eventi semplificato basato sul modello *push-push*.

La dichiarazione del componente `StockBroker` equivale alla dichiarazione seguente:

```
interface StockBroker : Components::CCMObject {
    // consumes StockName notifier_in
    StockNameConsumer get_consumer_notifier_in();

    // uses StockQuoter quoter_info_in;
    void connect_quoter_info_in(in StockQuoter c);
```

```
    StockQuoter disconnect_quoter_info_in();
    StockQuoter get_connection_quoter_info_in();
};
```

La dichiarazione del port `notifier_in` genera un'operazione che restituisce un riferimento di tipo `StockNameConsumer`, che verrà usato dal servizio di notifica per inviare eventi al componente `StockBroker`. La dichiarazione del receptacle `quoter_info_in` genera operazioni per creare la connessione con componenti che offrano un facet di tipo `StockQuoter` (`connect_quoter_info_in()`) e per distruggerla (`disconnect_quoter_info_in()`). L'operazione (`get_connection_quoter_info_in()`) restituisce un riferimento al facet collegato.

Seguono le dichiarazioni in IDL2 per il componente `StockDistributor`:

```
interface StockDistributor : Components::CCMObject, Trigger {
    // publishes StockName notifier_out;
    Components::Cookie
           subscribe_notifier_out(in Stock::StockNameConsumer c);
    Stock::StockNameConsumer
           unsubscribe_notifier_out(in Components::Cookie ck);

    // provides StockQuoter quoter_info_out;
    StockQuoter provide_quoter_info_out();

    attribute long notification_rate;
};
```

Il source `notifier_out` genera le operazioni per fare e disdire l'abbonamento alla sorgente di eventi. Ciascun abbonato è identificato da un *cookie*. Il facet `quoter_info_out` genera un'operazione che restituisce un riferimento di tipo `StockQuoter`.

La traduzione in IDL2 delle dichiarazioni in IDL3 mostra che vengono generate automaticamente tutte le operazioni necessarie a stabilire le connessioni fra i componenti.

Le implementazioni dei componenti vengono descritte in linguaggio CIDL. Per esempio, questa e' la definizione dell'implementazione di `StockBroker`:

```
composition session StockBroker_Impl {
    home executor StockBrokerHome_Exec  {
        implements Stock::StockBrokerHome;
        manages StockBroker_Exec;
    };
};
```

Questa definizione specifica che l'implementazione `StockBroker_Impl` è di categoria *session* ed è costituita da esecutori du tipo `StockBroker_Exec`, gestiti da una home di tipo `StockBrokerHome`, a sua volta implementata da un esecutore di tipo `StockBrokerHome_Exec`.

Possiamo ora accennare ad una possibile implementazione del componente `StockDistributor`, cominciando dall'interfaccia `StockQuoter`:

```
class StockQuoter_exec_i
        : public virtual ::Stock::CCM_StockQuoter {
    StockDistributor_exec_i& distributor_;
public:
    StockQuoter_exec_i(StockDistributor_exec_i& distributor)
        : distributor_(distributor) { };
    virtual ::Stock::StockInfo*
    get_stock_info(const char* stk);
};

::Stock::StockInfo*
StockQuoter_exec_i::
get_stock_info(const char* stk)
{
    Stock::StockInfo_var info = new Stock::StockInfo;
    info->name = CORBA::string_dup(stk);
    info->price = distributor_->get_price(stk);
    return info._retn();
}
```

L'esecutore `StockQuoter_exec_i` ha un puntatore all'esecutore di `StockDistributor`, che fornisce il prezzo del titolo identificato dalla stringa `stk` come risultato dell'operazione `get_price()`.

Quindi consideriamo l'esecutore di `StockDistributor`:

```
class StockDistributor_exec_i
        : public virtual StockDistributor_Exec {
    CORBA::Long rate_;
    ::Stock::CCM_StockDistributor_Context_var context_;
    pulse_Generator pulser_;
    // ... altri dati
public:
    StockDistributor_exec_i(void);
    virtual ~StockDistributor_exec_i(void);
    // interfaccia Trigger
    virtual void start();
    virtual void stop();
    // attributo rate
    virtual ::CORBA::Long rate();
    virtual void rate(::CORBA::Long);
```

```
    // facet quoter_info_out
    virtual ::Stock::CCM_StockQuoter_ptr provide_quoter_info_out();
    // interfaccia callback
    virtual void
        set_session_context(::Components::SessionContext_ptr ctx);
    virtual void ccm_activate();
    virtual void ccm_passivate();
    virtual void ccm_remove();

    void notify_evt();
    CORBA::Long get_price(const char* stk);
};
```

Questa classe deriva da `StockDistributor_Exec`, generata dal compilatore IDL3, e contiene le dichiarazioni delle operazioni previste dalla traduzione in IDL2 (ovviamente espresse in C++ secondo le regole del mapping). A queste si aggiungono le operazioni dell'interfaccia callback, e infine l'operazione `notify_evt()`, creata interamente dallo sviluppatore, che deve inviare gli eventi ai broker abbonati al servizio, e l'operazione `get_price()`, che restituisce il prezzo del titolo richiesto.

La parte privata della classe contiene vari membri dato, fra cui il valore della frequenza di aggiornamento (`rate_`), un puntatore all'interfaccia interna (`context_`), e un puntatore ad un'istanza della classe `pulse_Generator` (`pulser_`). Questa, definita dallo sviluppatore, è un oggetto attivo che periodicamente invoca l'operazione `notify_evt()`.

La classe `pulse_Generator` usa il framework *Adaptive Communication Environment* (ACE) [1], che offre fra l'altro una libreria per la programmazione multi-threaded. In questo framework, un oggetto attivo è una classe che deriva da `ACE_Task_Base`, come mostrato di seguito:

```
class pulse_Generator : public ACE_Task_Base {
    StockDistributor_exec_i* pulse_callback_;
    // ...
public:
    int start(CORBA::Long hertz);
    int stop();
    virtual int handle_timeout(const ACE_Time_Value& tv,
                               const void* arg);
    // ...
}
```

L'operazione `start()` imposta un timer che, con la frequenza richiesta, invoca l'operazione `handle_timeout()`. Quest'ultima chiama `notify_evt()` all'istanza di `StockDistributor_exec_i`, indirizzata dal puntatore `pulse_callback_`:

```
int
pulse_Generator::
handle_timeout(const ACE_Time_Value&, const void*)
{
    this->pulse_callback_->notify_evt();
    return 0;
}
```

L'operazione `notify_evt()`, per ogni titolo disponibile, crea un evento e lo invia al servizio di notifica. Questo avviene semplicemente invocando l'operazione `push_notify_out()`, dichiarata e implementata automaticamente, dell'interfaccia interna:

```
void
StockDistributor_exec_i::
notify_evt()
{
    // per ogni nome n nel database dei titoli:
    Stock::StockName_var evt = new OBV_Stock::StockName;
    evt->name (CORBA::string_dup(n));
    this->context_->push_notify_out(evt.in());
}
```

La funzione `in()` è un operatore di conversione definito nel mapping da IDL a C++, che serve a garantire che gli argomenti di ingresso vengano passati correttamente.

Appendici

A Un esempio

L'esempio presentato in questa sezione è tratto da [4]: si tratta di un sistema che controlla sia un impianto di condizionamento d'aria, sia alcuni macchinari (come forni o refrigeratori) che devono mantenere determinate temperature. L'impianto è costituito da un certo numero di dispositivi di due tipi: *termometri* (elementi sensori) e *termostati* (elementi sensori e attuatori). I dispositivi sono collegati a un calcolatore attraverso una rete locale, e sul calcolatore è installata una libreria, fornita dal produttore dei dispositivi, che permette di controllarli per mezzo di una semplice interfaccia di programmazione (API) costituita da alcune funzioni in C. Ci proponiamo di sviluppare un'applicazione distribuita, basata sull'architettura CORBA, che permetta il controllo remoto dell'impianto.

A.1 Caratteristiche dei dispositivi

Ogni dispositivo ha un controllore digitale che rende disponibili o permette d'impostare queste informazioni:

- *Numero identificatore* (*asset number*). È un numero, memorizzato permanentemente alla produzione, che identifica univocamente ciascun dispositivo. Viene usato dalla API del software di controllo come indirizzo di rete.
- *Modello.* Una stringa di caratteri, memorizzata permanentemente alla produzione, che identifica il modello del dispositivo.
- *Posizione.* Una stringa di caratteri che identifica la posizione del dispositivo nell'impianto. Quest'informazione è modificabile.
- *Temperatura attuale.* Un numero intero che rappresenta la temperatura attuale rilevata dal dispositivo. Quest'informazione viene fornita sia dai termometri che dai termostati.
- *Temperatura minima.* Un numero intero che rappresenta la minima temperatura che può essere mantenuta da un termostato. Viene memorizzato permanentemente alla produzione, ed è disponibile solo per i termostati.
- *Temperatura massima.* Un numero intero che rappresenta la massima temperatura che può essere mantenuta da un termostato. Viene memorizzato permanentemente alla produzione, ed è disponibile solo per i termostati.
- *Temperatura nominale.* Un numero intero che rappresenta la temperatura che deve mantenere un termostato. Quest'informazione viene impostata solo per i termostati, è ovviamente modificabile e può essere letta.

A.2 Interfaccia di programmazione

L'interfaccia di programmazione fornita dal produttore comprende quattro funzioni che eseguono queste operazioni:

- *Connettere un dispositivo alla rete*, dato il suo identificatore.
- *Sconnettere un dispositivo dalla rete*, dato il suo identificatore.
- *Leggere il valore di un attributo* per un dispositivo, dato il suo identificatore.
- *Impostare il valore di un attributo* per un dispositivo, dato il suo identificatore.

Queste funzioni restituiscono un valore speciale (-1) in caso di errore, in particolare se si cerca di leggere il valore di un attributo non definito per un tipo di dispositivi, o se si cerca di modificare il valore di un attributo non modificabile. Le dichiarazioni delle funzioni, e altri dettagli, verranno date piú oltre.

A.3 Requisiti per il sistema di controllo remoto

Il sistema di controllo che vogliamo realizzare (*CCS*, *Climate Control System*) deve permettere le seguenti operazioni:

1. *Elencare tutti i dispositivi* connessi all'impianto.
2. *Ricercare i dispositivi* con determinati valori di:
   (a) *identificatore*;
   (b) *posizione*;
   (c) *modello*.
3. *Leggere i valori degli attributi* di ciascun dispositivo collegato.
4. *Impostare i valori degli attributi modificabili* di ciascun dispositivo collegato.
5. *Impostare una variazione di temperatura nominale* su un insieme qualsiasi di termostati collegati.

Le operazioni che modificano la temperatura nominale devono sollevare un'eccezione se la nuova temperatura non rientra nell'intervallo ammissibile, e lasciare immutata l'impostazione. All'eccezione devono essere associate le informazioni relative alla temperatura richiesta e all'intervallo ammissibile, e un messaggio d'errore.

Osserviamo che non sono richieste operazioni per configurare l'impianto, cioè per connettere o sconnettere dispositivi: si suppone che queste operazioni non siano eseguibili remotamente (e quindi non facciano parte dell'applicazione CORBA). Si suppone inoltre che la configurazione sia statica, cioè che l'insieme dei dispositivi collegati non cambi dopo la configurazione.

A.4 Specifica IDL

L'insieme dei servizi offerti dall'applicazione CCS può essere specificato per mezzo di tre interfacce IDL:

- **Thermometer**: permette di controllare i singoli termometri.
- **Thermostat**: permette di controllare i singoli termostati.
- **Controller**: permette di controllare l'impianto, rendendo disponibili le operazioni per elencare o ricercare dispositivi e impostare variazioni di temperatura su insiemi di dispositivi.

Le dichiarazioni di queste interfacce, insieme ad alcune dichiarazioni `typedef`, sono contenute in un modulo:

```
// CCS.idl

module CCS {
    typedef unsigned long   AssetType;
    typedef string          ModelType;
    typedef short           TempType;
    typedef string          LocType;

    interface Thermometer {
        // ...
    };

    interface Thermostat : Thermometer {
        // ...
    };
    interface Controller {
        // ...
    };
};
```

L'interfaccia **Thermometer** contiene gli attributi comuni ai due tipi di dispositivi:

```
    interface Thermometer {
        readonly attribute ModelType    model;
        readonly attribute AssetType    asset_num;
        readonly attribute TempType     temperature;
                 attribute LocType      location;
    };
```

L'interfaccia **Thermostat** deriva da **Thermometer**, ereditandone quindi gli attributi, a cui aggiunge le operazioni per leggere e impostare la temperatura nominale. L'impostazione della temperatura può sollevare un'eccezione, quindi l'interfaccia contiene la dichiarazione dell'eccezione `BadTemp` e del tipo `BtData` che specifica le informazioni associate a tale eccezione:

```
interface Thermostat : Thermometer {
    struct BtData {
        TempType    requested;
        TempType    min_permitted;
        TempType    max_permitted;
        string      error_msg;
    };
    exception BadTemp { BtData details; };

    TempType    get_nominal();
    TempType    set_nominal(in TempType new_temp)
                    raises(BadTemp);
};
```

L'interfaccia **Controller** specifica le operazioni per elencare i dispositivi (`list()`), cercare quelli che rispondono a certe caratteristiche (`find()`), e applicare variazioni di temperatura (`change()`). Queste operazioni richiedono la dichiarazione di tipi ed eccezioni:

```
interface Controller {
    // ...
    // ... dichiarazioni di tipi
    // ...
    ThermometerSeq  list();
    void            find(inout SearchSeq slist);
    void            change(in ThermostatSeq tlist,
                           in short delta
                          ) raises(EChange);
};
```

L'operazione `list()` restituisce una lista di riferimenti a **Thermometer**, di tipo `ThermometerSeq` cosí definito:

```
typedef sequence<Thermometer>   ThermometerSeq;
```

L'operazione `change()` accetta in ingresso una lista (`tlist`) di riferimenti a **Thermostat** e il valore (`delta`) della variazione di temperatura. L'argomento `tlist` è di tipo `ThermostatSeq`:

```
typedef sequence<Thermostat>    ThermostatSeq;
```

Poiché l'operazione `set_nominal()` di **Thermostat** può sollevare l'eccezione `BadTemp`, e `change()` a sua volta deve sollevare un'eccezione, si introduce una

nuova eccezione, `EChange`. L'informazione associata a `EChange` (`errors`) è una lista (di tipo `ErrSeq`) di strutture `ErrorDetails`, ciascuna contenente un riferimento (`tmstat_ref`) al termostato che ha sollevato l'eccezione `BadTemp` e i dati (`info`) relativi. L'eccezione `EChange` permette quindi di aggregare i dati relativi a piú errori di impostazione di temperatura. Questo è necessario perché l'operazione `change()` agisce su piú dispositivi nella stessa chiamata. Seguono le dichiarazioni relative a `EChange`:

```
struct ErrorDetails {
    Thermostat          tmstat_ref;
    Thermostat::BtData  info;
};
typedef sequence<ErrorDetails>  ErrSeq;

exception EChange {
    ErrSeq  errors;
};
```

L'operazione `find()` ha, nonostante la semplicità della sua dichiarazione, un comportamento piuttosto complicato. Gli autori del testo ([4]) da cui è ripreso quest'esempio avvertono che le scelte di progetto fatte per l'operazione `find()` non sono adatte a un'applicazione reale, ma sono state fatte per mostrare come una specifica (di progetto, in questo caso) apparentemente semplice possa richiedere un'implementazione complessa, e anche per illustrare certe tecniche di programmazione. Il comportamento di `find()` si può riassumere come segue:

- Il chiamante di `find()` può richiedere, con una sola invocazione, i riferimenti ai dispositivi che soddisfano diverse interrogazioni: per esempio, con una sola chiamata si può chiedere al **Controller** quali dispositivi hanno la posizione "Stanza n. 10", quali la posizione "Capannone n. 4", e quali sono di tipo "Modello A".
- Ogni interrogazione è rappresentata da una struttura di tipo `SearchType`. Questa struttura ha due campi: `key` contiene la chiave di ricerca, cioè una unione di tipo `KeyType` che specifica l'attributo e il valore su cui fare la ricerca, mentre `device` conterrà, al ritorno della chiamata, un riferimento al dispositivo trovato, o al primo di essi se ce ne sono piú d'uno, o un riferimento nullo se non ce ne sono.
- In una chiave di ricerca, l'attributo su cui fare la ricerca viene indicato dal campo discriminate dell'unione. Questo campo assume valori appartenenti all'enumerazione `SearchCriterion`.

- Il chiamante inizializza una struttura `SearchType` per ciascuna interrogazione, inserisce le strutture inizializzate in una lista di tipo `SearchSeq`, e chiama `find()` passandole tale lista.
- L'operazione `find()`, per ogni interrogazione della lista iniziale (che come vedremo può acquisire nuovi elementi durante l'esecuzione), esegue questi passi:
  1. Cerca i dispositivi che soddisfano l'interrogazione.
  2. Se non ne trova, scrive un rifermento nullo nel campo `device`.
  3. Se ne trova almeno uno, scrive nel campo `device` un riferimento al primo dispositivo trovato.
  4. Se ne trova piú d'uno, tratta il primo come specificato al punto precedente, quindi per ogni dispositivo successivo inizializza una nuova struttura `SearchType` con la stessa chiave di ricerca della richiesta e il campo `device` contenente un riferimento al dispositivo, e aggiunge questa struttura alla lista di interrogazioni.

Le strutture dati richieste sono cosí dichiarate:

```
enum SearchCriterion { ASSET, LOCATION, MODEL };

union KeyType switch(SearchCriterion) {
case ASSET:
    AssetType   asset_num;
case LOCATION:
    LocType     loc;
case MODEL:
    ModelType   model_desc;
};

struct SearchType {
    KeyType     key;
    Thermometer device;
};
typedef sequence<SearchType>    SearchSeq;
```

A.5 Architettura del server

Le scelte iniziali nel progetto del server sono queste:

- viene istanziato un solo *servant* per ogni oggetto CORBA;
- gli oggetti CORBA sono transienti.

Sono possibili altri scelte, ma queste permettono una realizzazione semplice che si ritiene adeguata per l'applicazione.

Il compilatore IDL produce, dalle specifiche delle interfacce, le classi scheletro `POA_CCS::Thermometer`, `POA_CCS::Thermostat`, e `POA_CCS::Controller`. Al progettista spetta il compito di strutturare le classi *servant*: `Thermometer_impl`, `Thermostat_impl` e `Controller_impl`. Le scelte adottate a livello architetturale si possono sintetizzare come segue:

1. esiste una sola istanza della classe `Controller_impl`;
2. tale istanza mantiene una lista dei dispositivi collegati;
3. ogni dispositivo, all'atto della sua creazione, si registra nella lista tenuta dal controllore, e si deregistra all'atto della distruzione;
4. la classe `Thermostat_impl` viene derivata da `Thermometer_impl`.

Osserviamo che la scelta (4) non è una conseguenza necessaria del fatto che l'interfaccia **Thermostat** deriva da **Thermometer**. Le regole di mapping da IDL a C++ non richiedono che una relazione di generalizzazione fra interfacce venga tradotta meccanicamente in una generalizzazione fra le classi di implementazione.

Infine, avvertiamo che nel seguito chiameremo *operazioni IDL* le operazioni delle classi servant che implementano le operazioni dichiarate nelle interfacce IDL, per distinguere tali operazioni da altre operazioni delle classi servant.

A.6 La libreria ICP

In questa sezione mostriamo le dichiarazioni che costituiscono l'interfaccia di programmazione dell'ipotetica libreria, chiamata *ICP* (*Instrument Control Protocol*), che permette la gestione a basso livello dell'impianto.

Le funzioni disponibili sono le seguenti:

```
extern "C" {
    int ICP_online(unsigned long id);   // Add device
    int ICP_offline(unsigned long id);  // Remove device
    int ICP_get(unsigned long   id,     // Get attribute
                const char*     attr,
                void*           value,
                size_t          len
               );
    int ICP_set(unsigned long   id,     // Set attribute
```

```
                const char*     attr,
                const void*     value
               );
}
```

Tutte le funzioni restituiscono zero se l'esecuzione termina con successo, il valore -1 altrimenti. Il parametro `id` è l'identificatore del dispositivo.

Le funzioni destinate a leggere o impostare gli attributi dei dispositivo hanno il parametro `attr`, che è un puntatore a una stringa di caratteri corrispondente al nome dell'attributo in questione. I nomi degli attributi sono stringhe predefinite, come `model`, `location`, eccetera. Il chiamante di queste funzioni deve allocare (come variabile automatica, statica o dinamica) lo spazio di memoria per il valore del valore dell'attributo: per impostare un valore (`ICP_set()`) il chiamante inizializza la variabile e ne passa l'indirizzo attraverso il puntatore `value`, per leggere un valore (`ICP_get()`) passa anche la dimensione `len` della variabile, dove il valore viene scritto al termine dell'esecuzione.

A.7 La classe Thermometer_impl

La classe `Thermometer_impl` deve fornire i seguenti servizi:

- implementare le operazioni dell'interfaccia **Thermometer**;
- fornire le operazioni e le strutture dati richieste dalla scelta di progetto n. 3, relativa alla gestione della lista dei dispositivi.

Per realizzare le operazioni IDL ci serviamo di alcune operazioni ausiliarie che nascondono l'interfaccia a basso livello offerto dalla libreria ICP. Queste operazioni non sono strettamente necessarie, ma possono facilitare l'aggiornamento del codice nel caso che si voglia sostituire la libreria ICP con un'altra.

Per le operazioni di registrazione e deregistrazione dei dispositivi presso il controllore, occorre che:

- la classe `Controller_impl`, responsabile della gestione dei dispositivi, offra due operazioni pubbliche per registrare e deregistrare un dispositivo;
- le istanze della classe `Thermometer_impl` abbiano a disposizione l'indirizzo dell'istanza (unica) di `Controller_impl` che le controlla.

Possiamo vedere che `Thermometer_impl` dipende piuttosto strettamente dall'interfaccia di `Controller_impl`, in quanto non è possibile progettare e realizzare `Thermometer_impl` senza conoscere la parte dell'interfaccia di `Controller_impl` relativa alla gestione della lista dei dispositivi. Questa parte è costituita dalle operazioni `Controller_impl::add_impl()` e `Controller_impl::remove_impl()`: la prima serve a registrare un dispositivo, accettando come parametri il suo identificatore e un puntatore alla relativa istanza, la seconda serve a deregistrare un dispositivo, accettandone come argomento l'identificatore. Queste operazioni vengono invocate rispettivamente nel costruttore e nel distruttore di `Thermometer_impl`.

Per accedere al controllore, i dispositivi possono memorizzarne l'indirizzo in un membro dato. Questo membro può essere statico, dal momento che questa informazione è condivisa da tutti i dispositivi. Il membro statico deve essere inizializzato nel programma principale (il server).

Con queste premesse, possiamo considerare la dichiarazione di `Thermometer_impl`:

```
class Controller_impl;

class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    virtual CCS::ModelType  model()
                                throw(CORBA::SystemException);
    virtual CCS::AssetType  asset_num()
                                throw(CORBA::SystemException);
    virtual CCS::TempType   temperature()
                                throw(CORBA::SystemException);
    virtual CCS::LocType    location()
                                throw(CORBA::SystemException);
    virtual void            location(const char* loc)
                                throw(CORBA::SystemException);

    Thermometer_impl(CCS::AssetType anum, const char* location);
    virtual ~Thermometer_impl();

    static Controller_impl*     m_ctrl;

protected:
    const CCS::AssetType        m_anum;

private:
    CCS::ModelType  get_model();
    CCS::TempType   get_temp();
    CCS::LocType    get_loc();
    void            set_loc(const char* new_loc);

    Thermometer_impl(const Thermometer_impl&);
```

```
    void operator=(const Thermometer_impl&);
};
```

Le operazioni IDL usano le funzioni ausiliarie `get_model()`,..., `set_loc()`, che a loro volta usano le funzioni ICP; mostriamo, per esempio, l'implementazione di `model()`:

```
CCS::ModelType
Thermometer_impl::
model() throw(CORBA::SystemException)
{
    return get_model();
}

CCS::ModelType
Thermometer_impl::
get_model()
{
    char buf[32];
    if (ICP_get(m_anum, "model", buf, sizeof(buf)) != 0)
        abort();
    return CORBA::string_dup(buf);
}
```

La funzione POSIX `abort()` causa la terminazione anormale del programma.

Possiamo osservare che quasi tutti gli attributi dell'interfaccia **Thermometer** vengono implementati non con membri dato ma con delle operazioni che interrogano i dispositivi fisici. Soltanto il numero identificatore `m_anum` viene implementato come membro dato, in modo che ogni istanza di `Thermometer_impl` abbia a disposizione il numero identificatore del dispositivo che rappresenta. Questo membro viene inizializzato dal costruttore, ed ha visibilità protetta perché deve essere ereditato dalla classe `Thermostat_impl`.

Il costruttore inserisce in rete il dispositivo (`ICP_online()`), imposta l'identificatore e la posizione, che vengono passati come argomenti al costruttore, e infine registra il dispositivo presso il controllore:

```
Thermometer_impl::
Thermometer_impl(CCS::AssetType  anum,
                 const char*     location) : m_anum(anum)
{
    if (ICP_online(anum) != 0)
        abort();
    set_loc(location);
    m_ctrl->add_impl(anum, this);
}
```

Il distruttore ovviamente compie le operazioni inverse rispetto al costruttore:

```
Thermometer_impl::
~Thermometer_impl()
{
    try {
        m_ctrl->remove_impl(m_anum);
        ICP_offline(m_anum);
    } catch (...) {
        abort();
    }
}
```

A.8 La classe Thermostat_impl

Come già osservato, la classe `Thermostat_impl` viene derivata da `Thermometer_impl`. Si ricorre quindi all'eredità multipla, poiché una classe servant deve essere derivata dalla corrispondente classe scheletro, in questo caso `POA_CCS::Thermostat`, prodotta dal compilatore IDL:

```
class Thermostat_impl :
    public virtual POA_CCS::Thermostat,
    public virtual Thermometer_impl {
public:
    virtual CCS::TempType   get_nominal()
                                throw(CORBA::SystemException);
    virtual CCS::TempType   set_nominal(CCS::TempType new_temp)
                                throw(CORBA::SystemException,
                                     CCS::Thermostat::BadTemp);
    Thermostat_impl(
        CCS::AssetType  anum,
        const char*     location,
        CCS::TempType   nominal_temp
    );
    virtual ~Thermostat_impl() {}
private:
    CCS::TempType   get_nominal_temp();
    CCS::TempType   set_nominal_temp(CCS::TempType new_temp)
                        throw(CCS::Thermostat::BadTemp);
    Thermostat_impl(const Thermostat_impl&);
    void operator=(const Thermostat_impl&);
};
```

Anche in questa classe si fa uso di operazioni ausiliarie (`get_nominal_temp()` e `set_nominal_temp()`) che nascondono le chiamate ICP:

```
CCS::TempType
Thermostat_impl::
get_nominal_temp()
{
    short temp;
    if (ICP_get(m_anum, "nominal_temp", &temp, sizeof(temp)) != 0)
        abort();
    return temp;
}

CCS::TempType
Thermostat_impl::
set_nominal_temp(CCS::TempType new_temp)
throw(CCS::Thermostat::BadTemp)
{
    short old_temp;

    if (ICP_get(m_anum, "nominal_temp",
                &old_temp, sizeof(old_temp)
                ) != 0) {
        abort();
    }
    if (ICP_set(m_anum, "nominal_temp", &new_temp) != 0) {
        CCS::Thermostat::BtData btd;
        ICP_get(m_anum, "MIN_TEMP",
                &btd.min_permitted, sizeof(btd.min_permitted)
               );
        ICP_get(m_anum, "MAX_TEMP",
                &btd.max_permitted, sizeof(btd.max_permitted)
               );
        btd.requested = new_temp;
        btd.error_msg = CORBA::string_dup(
            new_temp > btd.max_permitted ? "Too hot" : "Too cold"
        );
        throw CCS::Thermostat::BadTemp(btd);
    }
    return old_temp;
}
```

L'operazione `set_nominal_temp()` deve restituire il valore precedente della temperatura nominale. Se l'impostazione della nuova temperatura non è possibile, una struttura di tipo `BtData` viene creata ed inizializzata con i valori estremi ammissibili per il dispositivo, e quindi passata al costruttore dell'oggetto-eccezione creato nell'istruzione `throw`.

A.9 La classe Controller_impl

Questa classe deve implementare le operazioni IDL `list()`, `change()` e `find()`, deve gestire l'elenco dei dispositivi collegati, richiesto per l'imple-

mentazione di tali operazioni, e fornire alle istanze di `Thermometer_impl` e `Thermostat_impl` che controllano i dispositivi l'interfaccia che permette loro di registrarsi e deregistrarsi. Questa interfaccia è costituita dalle operazioni `add_impl()` e `remove_impl()`.

La principale struttura dati è quindi l'elenco dei dispositivi collegati, o meglio dei servant che li controllano. Poiché ci si riferisce ai dispositivi per mezzo del loro identificatore, conviene realizzare questa struttura dati come una tabella che associ a ciascun identificatore un puntatore al servant corrispondente.

La libreria standard di template del C++ (*STL*, *Standard Template Library*) fornisce il template `map`, che permette di realizzare facilmente la tabella richiesta; l'uso di questo template verrà spiegato nel seguito.

Entro la classe `Controller_impl` si trova anche la dichiarazione della classe `StrFinder` usata dall'implementazione dell'operazione `find()`; anche questa classe verrà spiegata nel seguito.

La dichiarazione della classe `Controller_impl` è quindi la seguente:

```
class Controller_impl : public virtual POA_CCS::Controller {
public:
    virtual CCS::Controller::ThermometerSeq* list()
                                        throw(CORBA::SystemException);
    virtual void find(CCS::Controller::SearchSeq& slist)
                    throw(CORBA::SystemException);
    virtual void change(const CCS::Controller::ThermostatSeq& tlist,
                        CORBA::Short                          delta)
                            throw (CORBA::SystemException,
                                   CCS::Controller::EChange);

    void add_impl(CCS::AssetType anum, Thermometer_impl* tip);
    void remove_impl(CCS::AssetType anum);

    Controller_impl() {}
    virtual ~Controller_impl() {}
private:
    typedef map<CCS::AssetType, Thermometer_impl*> AssetMap;
    AssetMap m_assets;

    Controller_impl(const Controller_impl&);
    void operator=(const Controller_impl&);

    class StrFinder {
    // ...
    // ... dichiarazione di StrFinder
    // ...
    };
};
```

Il membro `m_assets` è la tabella dei servant, ciascuna delle cui righe è una coppia di elementi di nome `first` e `second`, il primo dei quali è l'identificatore del dispositivo e il secondo è un puntatore al servant corrispondente. La tabella viene gestita per mezzo delle operazioni `add_impl()` e `remove_impl()`:

```
void Controller_impl::
add_impl(CCS::AssetType anum, Thermometer_impl* tip)
{
    m_assets[anum] = tip;
}

void Controller_impl::
remove_impl(CCS::AssetType anum)
{
    m_assets.erase(anum);
}
```

Il template `map` permettere di accedere ad una riga della tabella usando l'operatore di indicizzazione (`[]`, ove l'indice assume valori del tipo specificato per il primo elemento della riga (in questo caso `CCS::AssetType` equivale a un intero, ma `map` permette l'uso di tipi arbitrari). Se non esiste una riga il cui primo elemento ha il valore specificato, ne viene creata una. L'operazione `erase()` cancella la riga avente il campo `first` uguale all'argomento.

L'operazione IDL `list()` crea un riferimento CORBA per il servant di ciascun dispositivo elencato in `m_assets`, inserisce tale riferimento in una sequenza, e infine restituisce un puntatore alla sequenza stessa:

```
CCS::Controller::ThermometerSeq* Controller_impl::
list() throw(CORBA::SystemException)
{
    CCS::Controller::ThermometerSeq_var listv
        = new CCS::Controller::ThermometerSeq(m_assets.size());
    listv->length(m_assets.size());

    CORBA::ULong count = 0;
    AssetMap::iterator i;
    for (i = m_assets.begin(); i != m_assets.end(); i++)
        listv[count++] = i->second->_this();
    return listv._retn();
}
```

Osserviamo l'uso di tipi e operazioni predefinite del template `map`: l'operazione `size()` restituisce il numero di elementi della tabella, il tipo `iterator` definisce gli iteratori sulla tabella, l'operazione `begin()` restituisce un iteratore inizializzato alla prima riga, l'operazione `end()` restituisce un iteratore

inizializzato alla fine della tabella ("alla riga dopo l'ultima"). L'operatore di incremento fa avanzare di una riga l'iteratore, e l'operatore di indirezione permette di accedere alla riga riferita dall'iteratore.

L'operazione IDL `change()` deve calcolare la nuova temperatura nominale di un insieme (`tlist`) di termostati a partire dalla temperatura nominale attuale e dalla differenza (`delta`) di temperatura richiesta. Inoltre deve sollevare un'eccezione di tipo `EChange` se la nuova temperatura è al di fuori dei limiti di uno o piú termostati. L'eccezione sollevata deve contenere le informazioni viste nella Sez. A.4.

Segue il codice dell'operazione:

```
void Controller_impl::
change(const CCS::Controller::ThermostatSeq& tlist,
       CORBA::Short                           delta)
          throw(CORBA::SystemException, CCS::Controller::EChange)
{
    CCS::Controller::EChange ec;

    for (CORBA::ULong i = 0; i < tlist.length(); i++) {
        if (CORBA::is_nil(tlist[i]))
            continue;
        CCS::TempType tnom = tlist[i]->get_nominal();
        tnom += delta;
        try {
            tlist[i]->set_nominal(tnom);
        } catch (const CCS::Thermostat::BadTemp& bt) {
            CORBA::ULong len = ec.errors.length();
            ec.errors.length(len + 1);
            ec.errors[len].tmstat_ref = tlist[i];
            ec.errors[len].info = bt.details;
        }
    }
    if (ec.errors.length() != 0)
        throw ec;
}
```

A.10 Operazione find() e classe StrFinder

L'implementazione dell'operazione IDL `find()` sfrutta alcune interessanti tecniche di programmazione rese possibili dalla STL: gli *algoritmi generici* (`find_if()`) e gli *oggetti funzione* (`StrFinder`).

Gli algoritmi generici della STL sono un insieme (circa 60) di funzioni generiche (*template*) che eseguono operazioni comuni, come la ricerca o

l'ordinamento, su sequenze di elementi il cui tipo viene fissato a tempo di compilazione. La sequenza su cui operare viene determinata da una coppia di iteratori che ne individuano gli estremi. In particolare, l'algoritmo `find_if()` serve a trovare il primo elemento che soddisfi un predicato entro una sequenza. L'algoritmo ha tre parametri di ingresso: i due iteratori che definiscono la sequenza, e un'espressione che possa essere valutata come una funzione a valori booleani che accetti in ingresso i valori degli elementi della sequenza. Tale funzione è ovviamente il predicato da soddisfare.

Questa espressione può essere semplicemente l'identificatore di una funzione opportunamente definita, o un puntatore a una funzione, ma può essere anche un *oggetto funzione*. Un oggetto funzione è un'istanza di una classe in cui l'operatore di applicazione di funzione (rappresentato dalle parentesi tonde) sia stato ridefinito per overloading. Nel nostro caso, la classe `StrFinder` è stata definita in modo che le sue istanze vengano usate come oggetti funzione. Il costruttore di questa classe inizializza due membri privati (`m_sc` ed `m_str`) con un criterio di ricerca (`LOCATION` o `MODEL`) e col valore da cercare. L'operatore di applicazione di funzione viene ridefinito in modo che abbia come parametro una riga di una tabella di tipo `AssetMap`, e confronti il valore cercato con l'attributo `location` o `model`, a seconda del criterio di ricerca, del dispositivo corrispondente.

La dichiarazione della classe `StrFinder` è la seguente:

```
class StrFinder {
public:
    StrFinder(CCS::Controller::SearchCriterion sc,
              const char *                      str)
                     : m_sc(sc), m_str(str) {}

    bool operator()(pair<const CCS::AssetType,
                         Thermometer_impl*>& p) const
    {
        switch (m_sc) {
        case CCS::Controller::LOCATION:
            return strcmp(CORBA::String_var(p.second->location()),
                          m_str) == 0;
            break;
        case CCS::Controller::MODEL:
            return strcmp(CORBA::String_var(p.second->model()),
                          m_str) == 0;
            break;
        default:
            abort();
        }
        return 0;
    }
```

```
private:
    CCS::Controller::SearchCriterion    m_sc;
    const char*                         m_str;
};
```

La classe generica `pair<>` definisce il tipo delle righe della tabella.

Possiamo adesso considerare l'implementazione dell'operazione `find()`. Il suo parametro di ingresso è una sequenza (`slist`) di richieste. Per ogni richiesta si esamina il criterio di ricerca. Se la ricerca è per numero identificatore, il dispositivo viene trovato dall'operazione `find()`, predefinita per la classe generica `map`; ovviamente in questo caso un solo dispositivo può soddisfare la richiesta. Se la ricerca è per posizione o per modello, si crea un'istanza di `StrFinder` inizializzandola col criterio di ricerca e con la stringa di caratteri che rappresenta la posizione o il modello richiesti. Questa istanza viene passata a `find_if()` insieme a una coppia di iteratori che definiscono la sequenza costituita da tutte le righe della tabella. L'algoritmo `find_if()` passa ciascuna riga all'oggetto funzione, e restituisce un iteratore che punta alla prima riga che soddisfa la richiesta; se nessuna riga la soddisfa, restituisce l'iteratore `end`. Ulteriori righe che soddisfano la richiesta vengono cercate invocando iterativamente `find_if()`, a cui si passa, come estremo iniziale della sequenza, l'iteratore `where` incrementato.

Segue il codice dell'operazione:

```
void Controller_impl::
find(CCS::Controller::SearchSeq& slist)
                    throw(CORBA::SystemException)
{
    CORBA::ULong listlen = slist.length();
    for (CORBA::ULong i = 0; i < listlen; i++) {
        AssetMap::iterator where;
        int num_found = 0;
        slist[i].device = CCS::Thermometer::_nil();
        CCS::Controller::SearchCriterion sc = slist[i].key._d();
        if (sc == CCS::Controller::ASSET) {
            where = m_assets.find(slist[i].key.asset_num());
            if (where != m_assets.end())
                slist[i].device = where->second->_this();
        } else {
            const char * search_str;
            if (sc == CCS::Controller::LOCATION)
                search_str = slist[i].key.loc();
            else
                search_str = slist[i].key.model_desc();
            where = find_if(
                        m_assets.begin(), m_assets.end(),
                        StrFinder(sc, search_str)
```

```
                );
            while (where != m_assets.end()) {
                if (num_found == 0) {
                    slist[i].device = where->second->_this();
                } else {
                    CORBA::ULong len = slist.length();
                    slist.length(len + 1);
                    slist[len].key = slist[i].key;
                    slist[len].device = where->second->_this();
                }
                num_found++;
                where = find_if(
                            ++where, m_assets.end(),
                            StrFinder(sc, search_str)
                        );
            }
        }
    }
}
```

La variabile `num_found` serve a distinguere il primo elemento trovato dai successivi.

A.11 Implementazione del server

Il server è molto semplice: un unico processo che, dopo aver inizializzato l'ORB, il POA e il servant manager, crea un'istanza di `Controller_impl` e alcuni servant per i dispositivi, e infine si mette in attesa di richieste da clienti remoti:

```
int
main(int argc, char * argv[])
{
    try {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        CORBA::Object_var obj
            = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var poa
            = PortableServer::POA::_narrow(obj);
        PortableServer::POAManager_var mgr
            = poa->the_POAManager();
        mgr->activate();

        Controller_impl ctrl_servant;
        Thermometer_impl::m_ctrl = &ctrl_servant;
        CCS::Controller_var ctrl = ctrl_servant._this();
        CORBA::String_var str = orb->object_to_string(ctrl);
        cout << str << endl << endl;
```

```
        Thermometer_impl thermo1(2029, "Deep Thought");
        Thermometer_impl thermo2(8053, "HAL");
        Thermometer_impl thermo3(1027, "ENIAC");
        Thermostat_impl tmstat1(3032, "Colossus", 68);
        Thermostat_impl tmstat2(4026, "ENIAC", 60);
        Thermostat_impl tmstat3(4088, "ENIAC", 50);
        Thermostat_impl tmstat4(8042, "HAL", 40);

        orb->run();
    } catch (const CORBA::Exception& e) {
        cerr << "Uncaught CORBA exception: " << e << endl;
        return 1;
    } catch (...) {
        abort();
    }
    return 0;
}
```

A.12 Un cliente

In questa sezione mostriamo una semplice applicazione cliente per il servizio CCS. Ovviamente questo programma ha un puro scopo dimostrativo: una vera applicazione avrebbe un'interfaccia a linea di comando (*CLI*, *Command Line Interface*), o un interprete di comandi, o un'interfaccia grafica che permettano all'utente di eseguire le operazioni volute, mentre questo esempio si limita a esercitare il sistema con una serie di operazioni prefissate.

L'applicazione deve:

1. ottenere una lista di riferimenti ai dispositivi presenti;
2. scrivere sull'uscita standard i dati relativi ai dispositivi;
3. cambiare la posizione del primo dispositivo della lista, e mostrare l'avvenuto cambiamento;
4. cambiare due volte la temperatura del primo termostato della lista, una volta con una temperatura ammissibile e l'altra con una non ammissibile;
5. cercare i dispositivi presenti in due posizioni, e mostrare i risultati della ricerca;
6. cambiare la temperatura di tutti i termostati;
7. in caso di eccezioni, mostrare le informazioni relative.

Per svolgere questi compiti si usano varie funzioni ausiliarie, di cui con-

sideriamo quella che scrive sull'uscita standard i dati dei dispositivi e quella che imposta la temperatura di un termostato.

La prima è una ridefinizione per overloading dell'operatore di uscita (<<) della libreria ostream del C++:

```
static ostream&
operator<<(ostream& os, CCS::Thermometer_ptr t)
{
    if (CORBA::is_nil(t)) {
        os << "Cannot show state for nil reference." << endl;
        return os;
    }

    CCS::Thermostat_var tmstat = CCS::Thermostat::_narrow(t);
    os << (CORBA::is_nil(tmstat) ? "Thermometer:" : "Thermostat:")
       << endl;

    CCS::ModelType_var model = t->model();
    CCS::LocType_var location = t->location();
    os << "\tAsset number: " << t->asset_num() << endl;
    os << "\tModel       : " << model << endl;
    os << "\tLocation    : " << location << endl;
    os << "\tTemperature : " << t->temperature() << endl;

    if (!CORBA::is_nil(tmstat))
        os << "\tNominal temp: " << tmstat->get_nominal() << endl;
    return os;
}
```

I test su tmstat servono a controllare se il dispositivo è un termostato. Altre ridefinizioni dell'operatore di uscita, che qui non mostriamo, sono usate per mostrare le informazioni associate alle eccezioni.

La funzione per impostare la temperatura di un termostato è la seguente:

```
static void
set_temp(CCS::Thermostat_ptr tmstat, CCS::TempType new_temp)
{
    if (CORBA::is_nil(tmstat))
        return;

    CCS::AssetType anum = tmstat->asset_num();
    try {
        cout << "Setting thermostat " << anum
             << " to " << new_temp << " degrees." << endl;
        CCS::TempType old_nominal = tmstat->set_nominal(new_temp);
        cout << "Old nominal temperature was: "
             << old_nominal << endl;
        cout << "New nominal temperature is: "
```

```
             << tmstat->get_nominal() << endl;
    } catch (const CCS::Thermostat::BadTemp& bt) {
        cerr << "Setting of nominal temperature failed." << endl;
        cerr << bt.details << endl;
    }
}
```

Possiamo ora esaminare il programma principale:

```
int
main(int argc, char* argv[])
{
    try {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
        if (argc != 2) {
            cerr << "Usage: client IOR_string" << endl;
            throw 0;
        }
        CORBA::Object_var obj = orb->string_to_object(argv[1]);
        if (CORBA::is_nil(obj)) {
            cerr << "Nil controller reference" << endl;
            throw 0;
        }
        CCS::Controller_var ctrl;
        try {
            ctrl = CCS::Controller::_narrow(obj);
        } catch (const CORBA::SystemException& se) {
            cerr << "Cannot narrow controller reference: "
                 << se << endl;
            throw 0;
        }
        if (CORBA::is_nil(ctrl)) {
            cerr << "Wrong type for controller ref." << endl;
            throw 0;
        }

        CCS::Controller::ThermometerSeq_var list = ctrl->list();
        CORBA::ULong len = list->length();
        cout << "Controller has " << len << " device";
        if (len != 1)
            cout << "s";
        cout << "." << endl;
        if (len == 0)
            return 0;
        for (CORBA::ULong i = 0; i < list->length(); i++)
            cout << list[i];
        cout << endl;

        CCS::AssetType anum = list[0]->asset_num();
        cout << "Changing location of device "
             << anum << "." << endl;
        list[0]->location("Earth");
```

```
        cout << "New details for device "
             << anum << " are:" << endl;
        cout << list[0] << endl;

        CCS::Thermostat_var tmstat;
        for (CORBA::ULong i = 0;
             i < list->length() && CORBA::is_nil(tmstat);
             i++) {
            tmstat = CCS::Thermostat::_narrow(list[i]);
        }
        if (CORBA::is_nil(tmstat)) {
            cout << "No thermostat devices in list." << endl;
        } else {
            set_temp(tmstat, 50);
            cout << endl;
            set_temp(tmstat, -10);
        }

        cout << "Looking for devices in Earth and HAL." << endl;
        CCS::Controller::SearchSeq ss;
        ss.length(2);
        ss[0].key.loc(CORBA::string_dup("Earth"));
        ss[1].key.loc(CORBA::string_dup("HAL"));
        ctrl->find(ss);
        for (CORBA::ULong i = 0; i < ss.length(); i++)
            cout << ss[i].device.in();
        cout << endl;

        cout << "Increasing thermostats by 40 degrees." << endl;
        CCS::Controller::ThermostatSeq tss;
        for (CORBA::ULong i = 0; i < list->length(); i++) {
            tmstat = CCS::Thermostat::_narrow(list[i]);
            if (CORBA::is_nil(tmstat))
                continue;
            len = tss.length();
            tss.length(len + 1);
            tss[len] = tmstat;
        }
        try {
            ctrl->change(tss, 40);
        } catch (const CCS::Controller::EChange& ec) {
            cerr << ec;
        }
    } catch (const CORBA::Exception& e) {
        cerr << "Uncaught CORBA exception: " << e << endl;
        return 1;
    } catch (...) {
        return 1;
    }
    return 0;
}
```

B Il profilo CORBA UML

NB: Questa sezione non è aggiornata.

Il profilo UML per le specifiche CORBA è un insieme di stereotipi e convenzioni che permettono di rappresentare in UML delle specifiche IDL. In questa sezione esporremo, molto schematicamente, alcune delle sue caratteristiche principali.

I moduli IDL sono rappresentati da package con lo stereotipo ≪CORBAModule≫. Per indicare che uno spazio di nomi (modulo, interfaccia, o tipo strutturato) è contenuto in un altro, si può usare la relazione di *contenimento di spazio di nomi* (*namespace containment*), rappresentata da una linea con una croce cerchiata all'estremità corrispondente allo spazio di nomi contenitore. Per i moduli si può usare anche l'inclusione grafica.

I tipi base IDL sono rappresentati da tipi di dato UML stereotipati come ≪CORBAPrimitive≫.

I tipi strutturati sono rappresentati come classi legate ai tipi dei componenti attraverso relazioni di composizione, eccetto per componenti di tipo interfaccia (cioè riferimento a oggetti), per i quali si usa l'associazione semplice.

Le interfacce IDL sono rappresentate da classi con lo stereotipo ≪CORBAInterface≫. Non si possono usare le interfacce UML, poiché le interfacce CORBA possono avere attributi e possono essere all'origine di associazioni navigabili, pertanto sono concettualmente diverse dalle interfacce UML.

La Fig. 13 mostra la rappresentazione UML della seguente interfaccia:

```
interface TestInterface {
    struct TestStruct {
        string member1;
    };
    attribute string myStringAttr;
    attribute TestStruct myStructAttr;
    void myOp1(in string s, inout TestStruct t);
    boolean myOp2(inout TestStruct t);
};
```

La proprietà `IDLOrder` serve a preservare l'ordine originale delle dichiarazioni nel codice IDL.

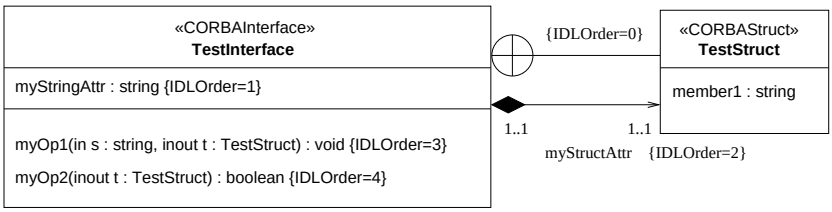

Figura 13: Interfaccia

I tipi definiti mediante dichiarazioni `typedef` hanno lo stereotipo ≪CORBATypedef≫, e la loro relazione col tipo rinominato viene rappresentata da una generalizzazione.

Le strutture sono rappresentate da classi con lo stereotipo ≪CORBAStruct≫. I membri di tipo semplice si rappresentano come attributi della classe, mentre i membri di tipo strutturato si rappresentano per mezzo di relazioni di composizione o di associazione.

Le enumerazioni si rappresentano con classi stereotipate da ≪CORBAEnum≫, i cui attributi, di tipo `CORBA::short` hanno lo stesso nome degli enumeratori.

Le eccezioni IDL sono rappresentate da eccezioni UML con lo stereotipo ≪CORBAException≫. Per specificare che un'operazione solleva delle eccezione, si associa all'operazione la proprietà (o *tagged value*) `raises`, il cui valore è la lista dei nomi delle eccezioni sollevate.

Le sequenze corrispondono a classi stereotipate da ≪CORBASequence≫ o ≪CORBAAnonymousSequence≫. Queste classi sono legate al tipo degli elementi attraverso associazioni qualificate, il cui qualificatore `index` è di tipo `CORBA::long`. La Fig. 14 mostra la rappresentazione della seguente sequenza non limitata:

```
typedef sequence<short> MySeq
```

Similmente, gli array si rappresentano con classi di stereotipo ≪CORBAArray≫ o ≪CORBAAnonymousArray≫, associate al tipo dei componenti con tanti qualificatori quante sono le dimensioni dell'array.

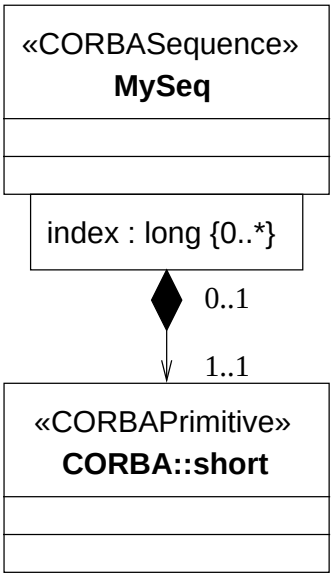

Figura 14: Sequenza

Riferimenti bibliografici

[1] ACE web site. `http://www.cs.wustl.edu/~schmidt/ACE.html`.

[2] CIAO web site. `http://www.cs.wustl.edu/~schmidt/CIAO.html`.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison–Wesley, 1999.

[5] Object Management Group, Framingham, MA, USA. *The Common Object Request Broker: Architecture and specification. Revision 2.6*, 2001. `www.omg.org/technology/documents/specifications.htm`.