UNIVERSITÀ DI PISA

Facoltà di Ingegneria
Corso di Laurea Specialistica in
Ingegneria Informatica per la Gestione d'Azienda

Tesi di laurea specialistica

Una piattaforma a servizi per l'inserimento di ordini d'acquisto in SAP R/3 da sistemi eterogenei remoti

Relatori

Ch.mo Prof. Francesco Marcelloni

Ch.ma Prof.ssa Beatrice Lazzerini

Dott. Giorgio Mauri

Candidato

Alessio Botta

Anno Accademico 2004 – 2005

Nei mesi di studio dedicati a questa tesi ho avuto la fortuna
di incontrare alcune persone che, in maniera diversa,
mi hanno sostenuto in questo lavoro impegnativo.

Ringraziamenti doverosi vanno quindi, in rigoroso ordine sparso,
all'Ing. Mingozzi per l'aiuto nella prima fase di ricerca dell'azienda
presso cui svolgere il tirocinio e per l'interesse suscitatomi
nei confronti della programmazione Java;
all'Ing. De Murtas per il supporto nell'utilizzo del sistema SAP;
al Prof. Casarosa per i preziosi consigli sull'architettura dei servizi web;
al Dott. Iannantuono, al Dott. Mariotti ed ai ragazzi dello Sviluppo Software
della TD Group S.p.A per l'accoglienza in azienda;
al Dott. Dell'Aquila per il tempo dedicatomi nello sviluppo
della versione del Process Engine utilizzato nella tesi
e per i consigli architetturali ed implementativi sul sistema Zlatan;
ad Antonio con il quale ho condiviso non solo l'esperienza
del tirocinio ma anche quella dei cinque anni di studio.

Desidero ringraziare inoltre gli altri miei compagni
e tutti coloro che in questi anni mi hanno incoraggiato e sopportato,
a partire dalla mia famiglia e da Ilaria.

Riassunto analitico

L'obiettivo finale della tesi è l'implementazione di un sistema distribuito che permetta di inserire un ordine di acquisto in SAP R/3 utilizzando protocolli di ingresso differenti (HTTP, SMTP/POP3, SMS e RMI).

Per fare questo, vengono prima studiate ed analizzate tre tecnologie che, in maniera diversa, contribuiscono alla realizzazione di una piattaforma con architettura *service-oriented*: il sistema di messaggistica Java *JMS*, l'applicazione XML *BPEL*, che permette di coordinare il flusso di esecuzione di più servizi web creando veri e propri *processi di business*, ed il connettore *SAP JCo*, per far comunicare un sistema SAP con un programma Java.

Nella seconda parte della tesi, vengono dettagliati i requisiti del sistema, l'architettura e l'implementazione: il modulo Java che gestisce l'ingresso multicanale, i servizi web che compongono il processo, la descrizione BPEL del processo stesso e l'interfaccia grafica.

Il sistema è stato progettato e realizzato nel corso di uno stage effettuato presso la *TD Group S.p.A.* come dimostrazione applicativa dell'utilizzo dell'interprete BPEL *GlobalBiz Process Engine* sviluppato dall'azienda.

Indice

Indice delle figure

Indice dei listati

1 Introduzione

1.1 Scopo del progetto

Per anni, nell'informatica, il focus nell'implementazione del software aziendale è stato puntato sulle *funzionalità* offerte da un determinato prodotto. Diversi fattori hanno nel tempo indebolito questa scelta, tra questi:

- l'ampiezza crescente di soluzioni hardware, software e/o integrate offerte dal mercato;
- la necessità sempre più frequente di integrare i *sistemi legacy* già in possesso dell'azienda;
- la proliferazione di piattaforme, servizi e canali diversi e non sempre compatibili;
- l'aumento crescente di domanda di qualità nei servizi, imposta dalle regole di un mercato sempre più competitivo.

Il focus si sta spostando sempre di più dalla visione funzionale verso quella a *processi*: il problema principale di chi acquista prodotti informatici non è più "*quali funzioni mi offre questo software*", ma piuttosto "*come può inserirsi nel mio sistema e quali vantaggi può apportare alle mie logiche di business*": diventa quindi fondamentale la capacità di *integrare* e *comporre* processi di business a partire da software di varia natura (vedi [IBMSOA]).

Il problema ha almeno due aspetti: uno di tipo architetturale, che trova completo supporto nella SOA e negli standard XML che la implementano, ed uno di tipo puramente tecnologico, che consiste nello studio delle soluzioni intraprese a basso livello per rendere effettiva l'integrazione delle applicazioni.

SOA (*Service Oriented Architetture*, [MSSOA]) è un modello per sistemi distribuiti nel quale i moduli software vengono visti come *servizi.* Un *servizio* può essere una qualsiasi funzione, applicativa o di sistema, semplice o composta a sua volta da altri servizi, interna od esterna all'ente che la utilizza, sviluppata ad hoc o fornita da terze parti. I servizi vengono esposti attraverso una *interfaccia*, scoperti tramite un sistema di *catalogazione* ed acceduti attraverso un *protocollo di trasporto* a livello applicativo.

La figura 1 illustra l'interazione tra gli attori di una SOA.

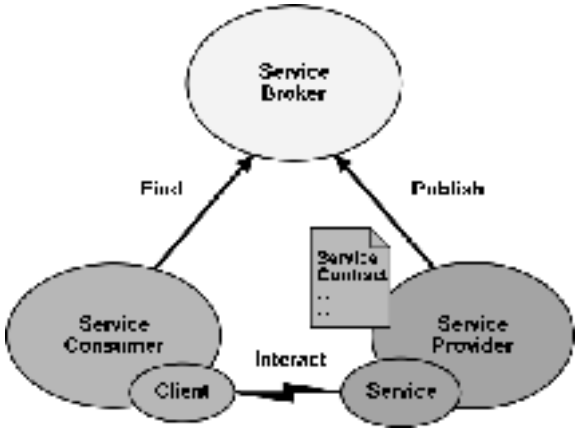

Figura 1: interazione tra attori nel modello SOA

Attualmente, SOA è implementata con la tecnologia dei *web service*: il linguaggio *WSDL* (*Web Service Description Language*, [WSDL]) serve per la descrizione dell'interfaccia del servizio (il *Service Contract* della figura), *UDDI* (*Universal Description, Discovery and Integration*, [UDDI]) è lo standard per la pubblicazione e la scoperta dei servizi sul web, mentre *SOAP* (vedi [SOAP]) è il protocollo applicativo di trasporto dei dati.

Il problema principale dell'aspetto tecnologico risiede nella maturità del modello e delle sue implementazioni: trattandosi di una tecnologia giovane e non completamente definita, è soggetta a interpretazioni e a personalizzazioni che, nella maggior parte dei casi, portano all'effetto contrario rispetto a quello voluto, ovvero ad una specializzazione della tecnologia piuttosto che ad una generalizzazione nell'ottica dell'integrazione.

Per fare un esempio, si pensi a come sia possibile scrivere una applicazione distribuita basata su socket, disinteressandosi completamente del formato dei pacchetti che transitano in rete, proprio perché la tecnologia che ne sta alla base – il protocollo TCP/IP – è matura e ben collaudata. Al contrario, non è pensabile nel breve termine scrivere un'applicazione complessa basata su web service non curandosi alla stessa maniera del formato dei messaggi SOAP scambiati, a causa dei troppi requisiti ancora non definiti, della molteplicità delle implementazioni e, come già accennato, della eccessiva personalizzazione.

Scopo di questa tesi è approfondire l'argomento dell'interoperabilità e dell'integrazione basata su web service, utilizzando un approccio empirico, ovvero tramite la scrittura di un sistema distribuito che abbia una qualche rilevanza pratica e che utilizzi una serie di tecnologie orientate all'integrazione.

In particolare, verranno esaminate tre di queste tecnologie che non sono state oggetto di studio approfondito nei vari corsi della Laurea Specialistica:

- *JMS*, un servizio offerto da J2EE per creare un meccanismo affidabile di scambio di messaggi asincroni tra processi distribuiti;
- *SAP JCo*, una libreria Java scritta da SAP come primo tentativo per rendere disponibili le risorse di una installazione R/3 verso processi Java;
- *BPEL*, una applicazione XML che permette di scrivere processi di business coordinando l'esecuzione di un insieme di web service.

Nella tesi, oltre alle tecnologie citate, si farà largo uso di *Java* per la programmazione dei moduli software, del *WSDL* per la descrizione dei web service, di alcuni tool di sviluppo, di librerie e di altro software che andremo a descrivere più avanti.

1.2 Contenuto del documento

I capitoli della tesi sono così organizzati:

- il capitolo 1 offre una panoramica del progetto e della tesi;
- il capitolo 2 prende in esame le tre tecnologie principali utilizzate, andando a descrivere per ognuna di esse il modello di riferimento, le applicazioni, gli strumenti, gli sviluppi futuri ed altre informazioni importanti;
- il capitolo 3 si occupa del progetto e della sua implementazione, partendo dagli aspetti funzionali e non, passando per la definizione dell'architettura e dei dati scambiati tra i vari attori fino a giungere alla descrizione analitica dei vari moduli software coinvolti. Nell'ultimo paragrafo vengono presentati anche due esempi completi di utilizzo dell'applicazione;
- il capitolo 4 riassume le conclusioni e gli spunti di riflessione nati dal lavoro svolto.

In appendice si trovano una introduzione all'uso di *Apache Axis*, il software utilizzato per la realizzazione dei web service, e la guida all'installazione del progetto realizzato.

1.3 Convenzioni e notazioni

Per una maggiore leggibilità del codice nei listati, si sono utilizzate le seguenti convenzioni:

- i commenti sono stati riportati in *corsivo*;
- alcune parti poco significative del codice sono state tagliate per leggibilità, quando tale mancanza può compromettere la comprensione del codice, viene riportata la parola `cut` in un commento o tre punti di sospensione `...`;
- nel codice Java, sono stati evidenziati in **grassetto** i frammenti di codice più significativi;
- nel codice XML, sono stati evidenziati in **grassetto** i nomi non qualificati degli elementi e sono stati sottolineati i valori degli attributi;
- nel codice XML, per ragioni di leggibilità talvolta non sono state riportate le definizioni dei namespace ed il nome qualificato di alcuni elementi.

Nel testo, i costrutti Java ed XML sono riportati in `carattere monospace`. Gli elementi XML sono anche `<racchiusi tra parentesi>`. I metodi Java sono seguiti da una coppia di parentesi tonde `()`, talvolta lasciate vuote per leggibilità anche nel caso in cui siano previsti argomenti.

I riferimenti bibliografici sparsi nel testo sono riportati tra parentesi quadre con un breve sigla identificativa (es. [J2EETUT] per il tutorial J2EE).

2 Le tecnologie

2.1 Java Message Service

JMS è un sistema utilizzabile per lo scambio di messaggi tra processi Java. La comunicazione via JMS è affidabile ed asincrona, poiché i processi comunicanti sfruttano un provider di servizi che si occupa del recapito dei messaggi.

La caratteristica principale di JMS è che i processi comunicanti sono in genere debolmente accoppiati (*loosely coupled*), ovvero:

- non devono necessariamente condividere interfacce, stub o oggetti;
- non devono essere necessariamente attivi nello stesso momento.

La differenza con altre tecnologie che permettono la programmazione distribuita (ad esempio *Java RMI*) è notevole.

J2EE include nella sua versione 1.4 tutte le interfacce e le classi per l'utilizzo di JMS; inoltre l'application server *Sun Java System AS 8* offre le funzionalità di provider a cui si è accennato.

In questo paragrafo prima verrà analizzato nel dettaglio il modello di riferimento di Java Message Service, quindi si passerà ad elencare i vari attori coinvolti nello scambio dei messaggi, si mostrerà la composizione del package `javax.jms` contenente le classi per l'utilizzo del servizio ed infine si accennerà ad alcune possibili applicazioni del protocollo.

Per un maggior approfondimento dell'argomento si rimanda a [J2EETUT].

2.1.1 Il modello

JMS è basato su un modello in cui sono previsti due pattern per lo scambio di messaggi: il pattern *peer-to-peer* (PTP o P2P) ed il pattern *publish/subscribe* (PUB/SUB). Le API di JMS forniscono un supporto diretto per questi pattern di comunicazione, ma l'affidabilità e la flessibilità del protocollo rendono facile l'implementazione di altri pattern.

Il pattern P2P (fig. 2) è il più semplice: uno o più client inviano messaggi su una *coda*, nella quale rimangono fino a quando non vengono consumati o scadono. Un solo client alla volta può consumare i messaggi dalla coda, sfruttando un meccanismo di *acknowledgement* per segnalare al mittente l'avvenuto consumo. È un pattern di comunicazione *molti-a-uno*, nel quale si può pensare alla coda di messaggi come ad un oggetto privato del client ricevente, accedibile a tutti per il solo invio: il paragone più immediato è quello con una casella di posta elettronica.

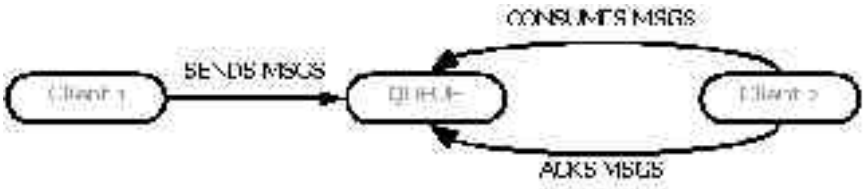

Figura 2: rappresentazione grafica del pattern JMS PTP

L'altro pattern, il PUB/SUB (fig. 3), prevede la presenza di un *topic*, ovvero di un'area dove un client, dopo essersi preventivamente registrato, è in grado di leggere i messaggi pubblicati da tutti gli altri client registrati. Questo pattern *molti-a-molti* è a tutti gli effetti quello utilizzato su *Usenet* per i newsgroup: difatti, tutti i messaggi pubblicati vengono "recapitati" a tutti i client iscritti. La dipendenza tra i client in questo caso è leggermente più forte che nel pattern P2P, poiché un client, una volta registrato, deve rimanere attivo e può leggere soltanto i messaggi inviati dopo la propria registrazione.

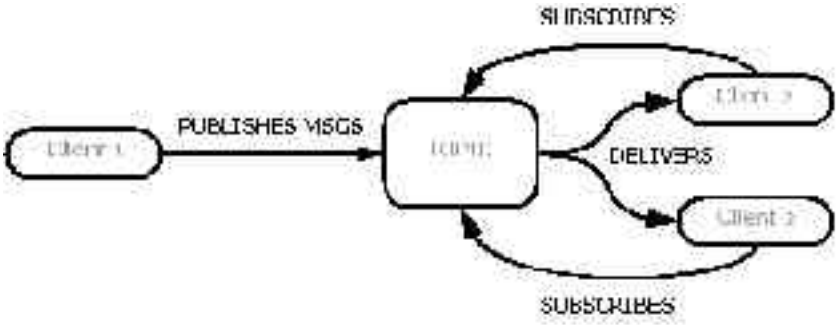

Figura 3: rappresentazione grafica del pattern JMS PUB/SUB

2.1.2 Gli attori

Abbiamo già accennato nei paragrafi precedenti al *provider* ed ai *client*. Nella terminologia JMS, un client è un qualunque processo che spedisca o consumi messaggi, senza distinzioni particolari. Il provider invece è un server che si occupa di:

- gestire le varie azioni previste del protocollo (invio e consumo di messaggi, gestione automatica degli acknowledgement, registrazione a topic, ecc...);
- gestire la materializzazione fisica delle code e dei topic in oggetti;
- fornire gli oggetti per l'accesso alle code e ai topic amministrati.

Gli oggetti del sistema si dividono in *messaggi* ed *oggetti amministrati*. I messaggi sono creati dai client a tempo di esecuzione, mentre gli oggetti amministrati vengono configurati staticamente e sono completamente gestiti dal provider. Gli oggetti amministrati sono di due tipi:

- le *connection factory* che, seguendo un *design pattern* ben noto in ambiente Java, permettono la generazione a run-time di connessioni JMS;
- le *destinazioni*, ovvero la materializzazione fisica delle code e dei topic a cui si è già accennato.

Tipicamente, gli oggetti amministrati sono riferibili all'interno del contesto del provider attraverso un nome *JNDI*, in maniera tale che i client possano identificarli univocamente.

2.1.3 Il package javax.jms

Il package `javax.jms`, presente, tra gli altri, nel file `j2ee.jar`, contiene tutte le classi che servono per utilizzare JMS. Il modello di programmazione proposto da JMS è simile a quello usato in altre tecnologie Java (*JDBC*, *JavaMail*, ecc...) e quindi facilmente comprensibile per chi abbia confidenza con il linguaggio. Nelle figure da 4 a 7 è riportato il *class diagram* delle principali interfacce.

Figura 4: class diagram di JMS con interfaccia base Connection

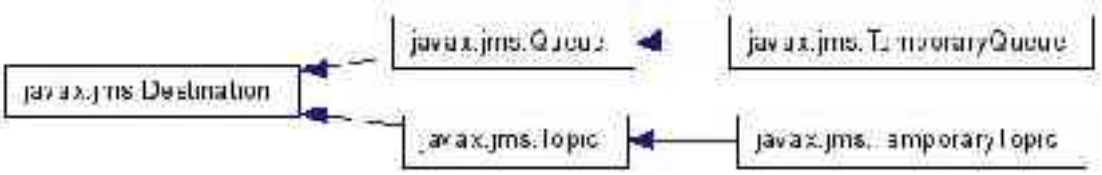

Figura 5: class diagram di JMS con interfaccia base Destination

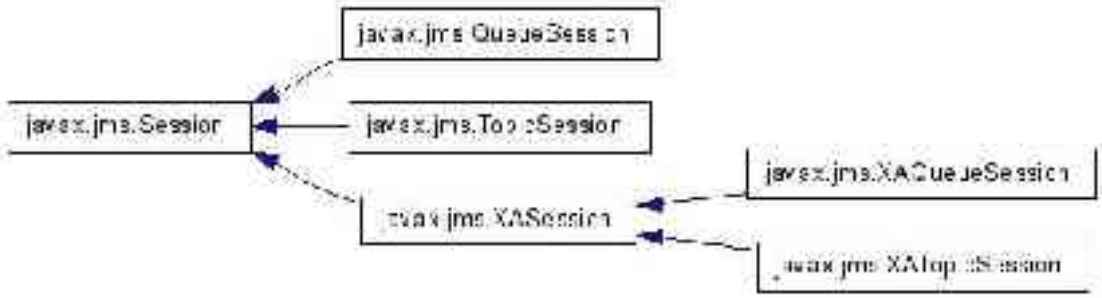

Figura 6: class diagram di JMS con interfaccia base Session

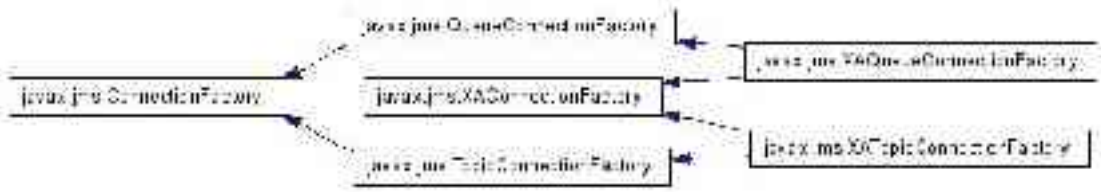

Figura 7: class diagram di JMS con interfaccia base ConnectionFactory

Come primo passo per l'utilizzo delle classi del package è necessario ottenere una `Connection` che rappresenti la connessione al provider JMS: le connessioni sono generate dalle *connection factory* impostate a tempo di configurazione. Dalla `Connection` è poi possibile ottenere una o più `Session`, oggetti che permettono di creare messaggi, produttori e consumatori.

Un messaggio è rappresentato semplicemente da un oggetto di tipo `Message`: i messaggi JMS sono formati, come in molti protocolli di trasporto, da uno *header*

contenente informazioni di controllo e da un *body* contenente il messaggio vero e proprio. Il tipo `Message` si specializza in:

- `BytesMessage` quando il messaggio da inviare è in formato binario;
- `MapMessage` quando il messaggio è composto da coppie (chiave, valore);
- `ObjectMessage` quando il messaggio contiene un qualunque oggetto Java serializzabile;
- `StreamMessage` quando il messaggio contiene tipi Java primitivi;
- `TextMessage` quando il messaggio contiene soltanto caratteri ed è interpretabile come stringa.

Per inviare e ricevere i messaggi, JMS mette a disposizione due classi il cui nome è esplicativo: `MessageProducer` e `MessageConsumer`. I *producer* al momento della creazione possono essere legati o meno ad una destinazione, mentre i *consumer* devono essere necessariamente collegati alla destinazione dalla quale desiderano ricevere i messaggi.

Un `MessageProducer` ha un metodo `send()` che permette di inviare il messaggio, un `MessageConsumer` dispone invece di diverse modalità per la ricezione:

- può invocare il metodo `receive()`, che blocca il processo fino a quando non è disponibile un nuovo messaggio nella destinazione cui è associato;
- può invocare il metodo `receive(int timeout)`, che blocca il processo fino a quando non è disponibile un nuovo messaggio o fino a quando scatta il `timeout`;
- può invocare lo stesso metodo `receive(int timeout)` con valore `timeout = 0`: in questa maniera, viene effettuato un controllo immediato della destinazione, ritornando il valore `null` se nessun messaggio è disponibile;
- può istanziare un oggetto di una classe che implementi l'interfaccia `MessageListener`: in questa maniera, ogni qualvolta verrà ricevuto un messaggio, verrà eseguito in maniera concorrente il metodo `onMessage()` della classe implementata.

JMS definisce come *sincrone* le prime tre modalità di ricezione e come *asincrona* l'ultima. È da sottolineare che questi aggettivi sono riferiti esclusivamente alla modalità di consumo dei messaggi da parte del client, non al protocollo di trasporto che in sé può senza dubbio essere definito completamente asincrono.

Riassumendo, il modello di programmazione previsto da JMS prevede:

- la creazione di una connessione da una connection factory;
- la creazione di riferimenti alle destinazioni accedute;
- la creazione di una o più sessioni dalla connessione ottenuta in precedenza;
- la generazione di client collegati alle destinazioni da una sessione;
- la creazione di messaggi da una sessione;
- l'invio di messaggi tramite i produttori e/o la lettura tramite i consumatori.

Come già accennato nel § 2.1.2, gli oggetti amministrati (connection factory e destinazioni), dai quali è possibile generare a cascata tutti gli altri oggetti, vengono creati a tempo di configurazione nel pannello di amministrazione del provider e possono essere riferiti all'interno dei programmi Java attraverso un nome JNDI assegnato loro: è quindi sufficiente ottenere un contesto JNDI ed eseguire un lookup per avere i riferimenti a tali oggetti.

La complessità nella creazione degli oggetti è una conseguenza diretta dell'architettura di JMS. Come già accennato infatti, le classi che implementano le interfacce del package sono fornite dallo sviluppatore del provider di messaggistica: per garantire portabilità al codice, le specifiche sconsigliano l'istanziazione diretta di oggetti di tali classi (con il costrutto `new`) ma prevedono la possibilità di usare i metodi alternativi visti per la creazione a cascata degli oggetti.

Nel listato 1 è riportato un semplice esempio di utilizzo del package `javax.jms`: il classico *"Hello World!"*, dove la stringa viene inviata e ricevuta via JMS dallo stesso processo.

```
public class JMSHelloWorld {

  static public void main(String args[]) {

    try {

      // recupera il contesto JNDI

      javax.naming.Context jndiContext =
        new javax.naming.InitialContext();

      // lookup JNDI degli oggetti amministrati

      javax.jms.ConnectionFactory cf = (ConnectionFactory)
        jndiContext.lookup("jms/MyConnFactory");

      javax.jms.Destination queue = (Queue)
        jndiContext.lookup("jms/MyQueue");

      // crea connessione e sessione

      javax.jms.Connection connection   = cf.createConnection();
      javax.jms.Session    session      =
        connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

      // crea un produttore associato alla destinazione

      javax.jms.MessageProducer mp = session.createProducer(queue);

      // crea un messaggio di testo e lo riempie

      javax.jms.TextMessage out    = session.createTextMessage();
      out.setText("Hello World!");

      // invia il messaggio
```

```
        mp.send(out);
        mp.close()

        // crea un consumatore associato alla destinazione

        javax.jms.MessageConsumer mc = session.createConsumer(queue);

        // attiva la ricezione dei messaggi

        connection.start();

        // crea un messaggio per la ricezione

        javax.jms.Message in         = session.createMessage();

        // riceve il messaggio e ne stampa il contenuto

        in = mc.receive();
        if (in instanceof javax.jms.TextMessage)
           System.out.println(((TextMessage) in).getText());

        // chiude la connessione

        mc.close();
        connection.close();

      } catch (Exception e) {
        e.printStackTrace();
      }
    }
}
```

Listato 1: codice di JMSHelloWorld

2.1.4 Le applicazioni

2.1.4.1 Comunicazione tra processi Java

La comunicazione tra processi Java è l'applicazione nativa di JMS. I pattern di comunicazione implementabili grazie all'utilizzo di JMS non si limitano ai soli PTP e PUB/SUB: componendo questi due, è possibile creare protocolli anche complessi per rispondere ad esigenze diverse.

Supponiamo ad esempio di avere la necessità di scrivere un server che gestisca un numero indeterminato di richieste concorrentemente. Tipicamente, la soluzione per un problema di questo tipo è l'utilizzo di un *pool di thread*: un *dispatcher* interno al server si occupa di inoltrare le richieste ricevute ai vari thread, in maniera da distribuire il carico il più possibile.

Questa soluzione ha molti lati positivi, ma può non essere quella giusta nel caso in cui si vogliano distribuire gli stessi thread del server su più macchine distinte: JMS permette di implementare in maniera semplice un "server distribuito" in cui i moduli dell'elaborazione delle richieste non devono necessariamente risiedere sulla stessa macchina.

I processi necessari per il funzionamento di un server di questo tipo sono:

1. un provider JMS per ogni macchina;
2. un unico dispatcher che si occupi dell'inoltro delle richieste;
3. uno o più moduli (corrispondi ai thread serventi) per l'elaborazione delle richieste.

Una situazione esemplificativa è rappresentata in fig. 8.

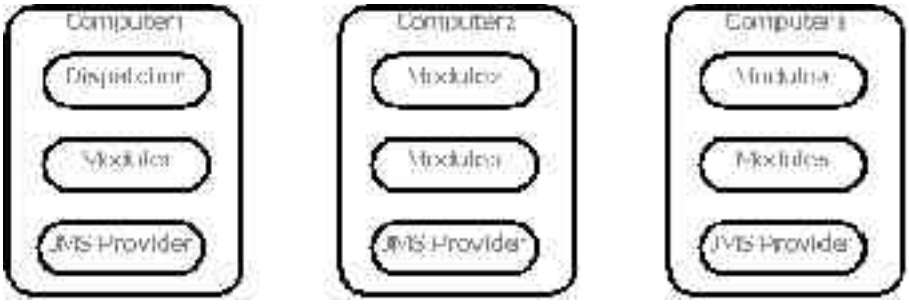

Figura 8: esempio di composizione dei processi di un server distribuito

Il dispatcher deve essere il primo processo ad essere attivato: deve esporre verso l'esterno una interfaccia di qualche tipo per la ricezione delle richieste, oltre ad avere accesso in lettura ad una coda JMS amministrata.

Dopo il dispatcher, vengono lanciati uno o più moduli. Le azioni intraprese da un modulo all'avvio sono:

1. creazione di una coda JMS temporanea privata;
2. notifica della propria attivazione al dispatcher;

3. attesa, consumo ed elaborazione delle richieste sulla propria coda privata.

In sostanza, la coda amministrata serve per far comunicare i moduli con il dispatcher: inviando un messaggio su tale coda, il modulo notifica la propria presenza e la propria disponibilità all'elaborazione delle richieste. La coda privata creata dal modulo deve essere invece utilizzata in senso inverso, ovvero dal dispatcher verso il modulo, per l'inoltro delle richieste di elaborazione giunte al dispatcher attraverso l'interfaccia pubblica.

Il protocollo JMS prevede l'utilizzo di un campo `ReplyTo` nell'header del messaggio: quando un modulo si registra, invia il messaggio sulla coda amministrata, settando il valore di `ReplyTo` con l'indirizzo della coda temporanea da esso creata, in maniera tale da renderla nota, e quindi utilizzabile, al dispatcher.

Questo modello può essere reso complesso a piacere: ad esempio, si possono prevedere algoritmi sofisticati per la scelta del modulo a cui inoltrare una richiesta e si possono introdurre altri messaggi di controllo oltre a quello per la registrazione (la notifica della chiusura del modulo, un ack in risposta ad una richiesta di elaborazione). Ovviamente, un server distribuito implementato con questa soluzione genera un certo overhead, dovuto allo scambio dei messaggi JMS, alla serializzazione degli oggetti contenuti nei messaggi ed alla presenza dei provider JMS su ogni macchina; gli aspetti positivi riguardano invece la semplicità di implementazione e la possibilità di generare molte configurazioni diverse a seconda delle esigenze. In figura 9 è riportato un esempio di funzionamento di un server distribuito.

Figura 9: schema di funzionamento del server distribuito

2.1.4.2 SOAP-over-JMS

Introducendo JMS si è accennato al fatto che si tratta di una tecnologia dove i processi comunicanti sono *loosely coupled*, ovvero devono conoscere poco l'uno dell'altro e comunicano in una modalità sostanzialmente asincrona. Non è un caso che questa definizione ricorra anche quando si parla di web service e di architetture orientate ai servizi.

Per motivi storici (il grande successo del web) e pratici (la diffusione dei server web e la penetrabilità dei firewall sulla porta 80), il protocollo di riferimento per i web service è HTTP. Si tratta tuttavia di un controsenso, perché HTTP ha due caratteristiche fondamentali che contrastano pesantemente con il modello architetturale proposto da SOA: è *sincrono* ed è *poco affidabile* (vedi anche § 2.2.3.2). Gli sforzi nella direzione della creazione di un HTTP più adatto all'uso nell'ambito dei web service hanno portato alla definizione delle specifiche del protocollo *HTTP-R*, dove "R" sta per *Reliable*, ovvero "affidabile" (vedi [HTTPR]).

JMS può a tutti gli effetti essere considerato un protocollo di trasporto, con caratteristiche che si adattano perfettamente all'utilizzo in una SOA. Per questo, è stato proposto l'utilizzo di SOAP con JMS (*SOAP-over-JMS*). Il *W3C* ed il *WS-I Consortium* non hanno ancora standardizzato il binding tra SOAP e JMS come già avvenuto per HTTP e SMTP (vedi anche [SOAP0] e [WSIBP]), ma si trovano già prodotti che sfruttano questa tecnologia. Lo stesso Axis (vedi Appendice A) fornisce nelle sue ultime versioni il supporto per l'utilizzo di JMS come protocollo di trasporto, oltre a HTTP e SMTP: il package `org.apache.axis.transport.jms` contiene le classi per gestirlo. I modelli previsti sono due (vedi [AXIS]): nel primo, JMS viene proposto come protocollo di trasporto nell'ambiente privato della LAN aziendale, mentre ad HTTP viene riservato il ruolo di protocollo di trasporto Internet; nel secondo, si ipotizza una più complicata comunicazione via JMS attraverso la stessa Internet (vedi fig. 10).

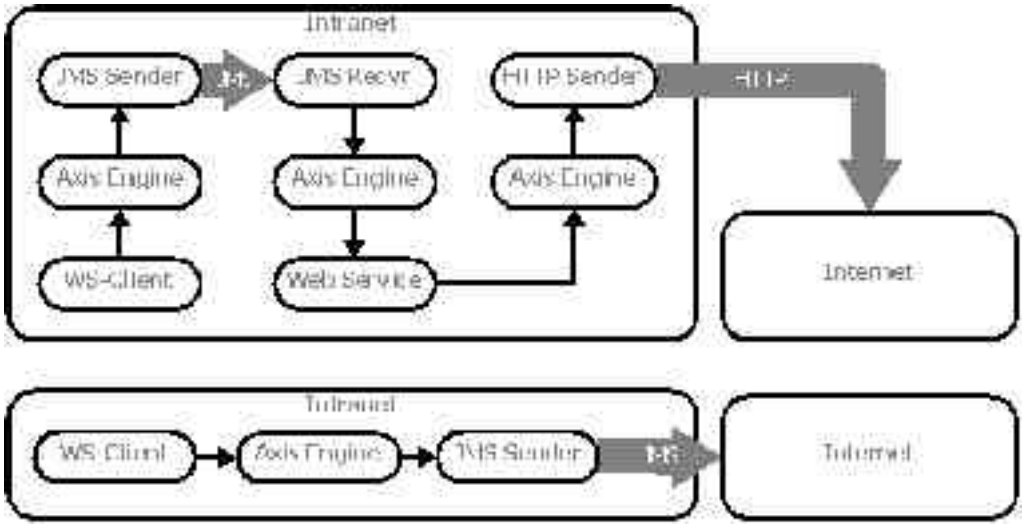

Figura 10: i modelli di utilizzo di SOAP-over-JMS proposti da Axis

2.1.4.3 Integrazione con altri prodotti

Da anni nel mercato dell'EAI sono presenti prodotti di messaggistica asincrona, appartenenti alla categoria del cosiddetto *MOM* (*Message-Oriented Middleware*), cui appartiene anche JMS: tra i più famosi, *IBM MQ Series* (oggi *WebSphere MQ*, vedi [IBMWMQ]) e *MSMQ* di Microsoft (vedi [MSMQ]).

Tipicamente, ognuno di questi prodotti dispone di una API proprietaria per l'utilizzo del proprio sistema di messaggistica nelle applicazioni sviluppate da terzi: una soluzione efficiente ma molto personalizzata, non portabile su piattaforme diverse e soggetta ad una esplosione di complessità ogni qualvolta si voglia aggiornare il sistema o produrre una libreria per un nuovo linguaggio.

Come visto nei paragrafi precedenti, JMS introduce un livello di astrazione tra le classi che implementano il sistema di messaggistica e le applicazioni che lo usano, definendo soltanto le interfacce visibili al programmatore ed utilizzando JNDI per accedere agli oggetti amministrati. Il tipico meccanismo Java del *binding* a *run-time* tra interfacce e classi permette di mascherare la vera implementazione, dipendente dal provider, della libreria utilizzata. Questa soluzione, mirata alla portabilità, permette anche di integrare le applicazioni JMS con sistemi di messaggistica legacy (vedi [IBMJMS]).

Due meccanismi sono utilizzabili per far comunicare JMS con un MOM diverso: l'utilizzo di appositi *bridge* che facciano da traduttori tra un protocollo e l'altro oppure la modifica delle API del sistema legacy per adattarle alle specifiche JMS.

Un esempio del primo caso è *FioranoMQ Bridge* ([FMQ]). *FioranoMQ* è un provider JMS alternativo a *SJSAS8* (vedi § 2.1.5.1) che comprende un tool per trasformare messaggi JMS in ingresso o in uscita in analoghi messaggi in formato MQ Series o MSMQ.

MQ Series classes for JMS (vedi [IBMMQSJMS]) è invece un esempio del secondo meccanismo: una estensione prodotta nel 2001 da IBM per il proprio sistema MQ Series che implementa le specifiche JMS e rende disponibile tramite le interfacce distribuite dalla Sun il sistema di messaggistica preesistente (in seguito, IBM ha creato nuove versioni di MQ Series integrate nell'application server WebSphere, con una architettura diversa ed un supporto Java più ampio).

2.1.5 Gli strumenti

2.1.5.1 Sun Java System Application Server 8

Sun Java System Application Server 8 è l'application server Java distribuito gratuitamente con la versione 1.4 di J2EE, scaricabile presso [J2EE]. *SJSAS8* è stato utilizzato come provider JMS e web server nel progetto di cui è oggetto la seconda parte di questa tesi (§ 3).

SJSAS8 permette di avere in un unico software le funzionalità di:

- un server web;
- un container per web application e JSP;
- un provider JMS;
- un container EJB;
- un gestore di pool di connessioni a database;
- un server JNDI;
- ...

Inoltre, SJSAS8 viene distribuito insieme a diversi tool, tra i quali:

- un database manager;
- una console GUI per l'amministrazione (vedi fig. 11);
- un tool per il packaging ed il deploy delle applicazioni;
- un tutorial completo e diverse applicazioni di esempio.

SJSAS8 è un prodotto sovradimensionato per un uso limitato alle funzionalità del provider JMS. Per questo, Sun distribuisce gratuitamente anche *Sun Java System Message Queue Platform Edition*, un server decisamente più leggero destinato soltanto al *messaging* JMS.

Figura 11: il pannello di amministrazione di SJSAS8

2.2 Business Process Execution Language

Business Process Execution Language for Web Services (abbreviato come BPEL4WS o più spesso semplicemente BPEL, vedi [BPEL]) è una iniziativa promossa da un gruppo di aziende comprendente IBM, Microsoft e SAP per guidare e garantire l'interoperabilità nella descrizione e nell'esecuzione di processi di business basati su web service (vedi [SAPBPEL]).

BPEL è nato nel 2002 dalla fusione tra due linguaggi specificati entrambi l'anno precedente ed in seguito non più supportati: *Web Service Flow Language* di IBM (vedi [WSFL]) e *XLANG* di Microsoft (vedi [XLANG]). La volontà delle due aziende di cooperare ed integrare i due linguaggi in un unico per farlo diventare standard *de facto* nel settore è molto significativa proprio nell'ottica di interazione tra piattaforme diverse.

In primo luogo, BPEL è una *applicazione XML.* Questo permette a BPEL di integrarsi facilmente con altre applicazioni XML utilizzate nell'ambiente dei web service (in particolar modo WSDL) tramite meccanismi come quello dei namespace.

BPEL ha un duplice obiettivo: da una parte, fornisce una *descrizione*, generalizzata a vari livelli, di un processo di business; dall'altra, in maniera simile ad un linguaggio procedurale, *specifica la sequenza delle chiamate* a web service da effettuare per un determinato processo.

Il primo aspetto di BPEL può essere compreso meglio se si pensa al BPEL come ad un WSDL pensato per una scala più grande: nel WSDL, gli elementi "atomici" da cui si parte per la descrizione sono i messaggi scambiati tra client e service, che si compongono in operazioni ed interfacce e si materializzano in messaggi SOAP. Nel BPEL invece gli elementi atomici sono i web service stessi, che rappresentano *partner* di interazioni complesse.

Il secondo aspetto di BPEL invece è più vicino ai linguaggi di programmazione: non solo viene descritto il ruolo del servizio nel processo e le interazioni con gli altri partner, ma vengono anche specificate quali sono le variabili di ingresso e di uscita, la modalità sincrona o asincrona della chiamata, la sequenza delle operazioni preliminari da effettuare e come devono essere gestite le eccezioni.

Curiosamente, può essere fatto un paragone tra BPEL ed SQL ([PBBLOG]). In effetti, il duplice aspetto di BPEL è lo stesso che ritroviamo in SQL: da una parte, si *descrive* il formato delle tabelle e dei dati, dall'altra, si specifica quali comandi debbano essere *eseguiti* per la manipolazione dei dati stessi. Inoltre, come SQL, BPEL non ha vita propria: necessita infatti di un *interprete* che compia a livello applicativo le operazioni descritte nel codice XML.

L'idea che sta alla base di BPEL è la stessa che si ritrova in altri linguaggi o protocolli basati su XML: *generalizzare* il più possibile i concetti, lasciando spazio alla possibilità di *espandere* il linguaggio, senza occuparsi degli *aspetti implementativi* di basso livello, che sono lasciati all'interprete.

Questo atteggiamento è produttivo nel momento della definizione delle specifiche, quando, astraendosi dalla realtà, si riesce a creare un linguaggio estremamente pulito, flessibile ed elegante anche nella rappresentazione concettuale. Quando però si passa all'utilizzazione del linguaggio, ovvero alla rappresentazione di processi o alla

scrittura di interpreti, ci si scontra inevitabilmente con problemi “reali” che nel momento della specifica non potevano essere previsti.

Un fenomeno analogo è già avvenuto con WSDL e SOAP: la differenza di interpretazione delle specifiche, al momento della scrittura dei compilatori WSDL e dei moduli che gestiscono SOAP, ha portato ad una tale differenziazione dei linguaggi nei prodotti di diverse aziende, che è stato necessario ricorrere alla definizione del *WS-I Basic Profile* [WSIBP], ovvero a un insieme di regole chiarificatrici e restrittive sull’interpretazione di WSDL e SOAP, in maniera tale da limitare le potenzialità dei linguaggi ma consentendone una maggiore omogeneità al momento dell’implementazione.

In altri termini, la creazione di un modello completamente generale ed astratto, fatta estraniandosi dagli aspetti implementativi al fine di una totale interoperabilità, può portare all’effetto opposto, ovvero ad una necessità di interpretazione talmente dettagliata da perdere qualsiasi caratteristica di portabilità non solo del codice ma del modello stesso.

L'utilizzo del BPEL non è ancora molto diffuso: soltanto al momento della scrittura di questo documento stanno nascendo le prime versioni degli interpreti e di conseguenza stanno venendo alla luce i problemi implementativi.

Prima di passare ad una sintetica analisi dei costrutti, delle potenzialità e dei limiti del linguaggio, è necessario chiarire alcuni aspetti generali.

Nella terminologia BPEL, come già accennato, un *processo* è composto dalla rappresentazione delle interazioni tra i partner e dalla descrizione della sequenza di attività da svolgere: si tratta quindi di un oggetto “statico”. Una *istanza di processo* è invece un oggetto “dinamico”: viene creata dall'interprete a *run-time*, nel momento in cui un cliente scatena la sequenza delle attività, ed è composta dalla materializzazione del processo in variabili istanziate, spazio di memoria, messaggi SOAP, ecc. La relazione tra processo ed istanza di processo può essere comparata alla relazione che passa tra classe ed oggetto in Java.

Un processo BPEL tipicamente si espone verso l'esterno con un `<portType>`, presentandosi ai partner come se fosse un servizio web a tutti gli effetti. Per questo, negli esempi dei prossimi paragrafi, saranno sempre presenti le descrizioni WSDL del servizio che viene implementato dall'interprete. Ogni volta che questo riceve un messaggio di attivazione sulla `<portType>` definita per un processo, crea ed attiva una nuova istanza del processo stesso.

Questo meccanismo offre risvolti interessanti poiché, se un processo si espone all'esterno con l'interfaccia di un web service, è possibile definire processi di business complessi andando a comporre altri processi più semplici descritti in precedenza, creando così sistemi molto modulari e scalabili in grado di mappare perfettamente i processi reali di una azienda.

2.2.1 Il linguaggio

2.2.1.1 Struttura

Nel listato 2 è riportato un primo semplice esempio di processo BPEL, utile per comprendere quale sia la struttura del codice. Il processo descritto non fa altro che l'echo di una stringa (vedi anche [IBMBPEL3]).

```
<process name="echoProcess"
targetNamespace="http://www.echo.org"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
xmlns:tns="http://www.echo.org">

  <partnerLinks>
    <partnerLink  name="client"
                  partnerLinkType="tns:echoServicePLT"
                  myRole="service" />
  </partnerLinks>

  <variables>
    <variable   name="echoString"
                messageType="tns:echoMessage" />
  </variables>

  <sequence>
    <receive  name="receive1"
              partnerLink="client"
              portType="tns:echoServicePT"
              operation="echo"
              variable="echoString"
              createInstance="yes" />
    <reply    name="reply1"
              partnerLink="client"
              portType="tns:echoServicePT"
              operation="echo"
              variable="echoString" />
  </sequence>

</process>
```

Listato 2: processo BPEL echoProcess

Le strutture minime sono la root `<process>`, una sezione `<partnerLinks>` che serve a descrivere le interazioni tra i web service, una serie di variabili definite dentro `<variables>` ed una sequenza (`<sequence>`) di attività da svolgere, che in questo caso consistono semplicemente nella ricezione di un messaggio e nel suo inoltro senza modifiche allo stesso mittente.

2.2.1.1.1 Integrazione con il WSDL

BPEL utilizza ampiamente costrutti propri ed estensioni al linguaggio del WSDL: i costrutti originali sono usati normalmente per conoscere dettagli relativi ai web service da invocare (ad esempio interfaccia e formato dei messaggi in ingresso ed in uscita), mentre le estensioni vengono usate nella definizione astratta del processo.

Nel listato precedente sono stati utilizzati attributi che fanno riferimento ad elementi definiti in un file WSDL. Il listato 3 riporta un possibile esempio di file WSDL coerente con le definizioni riferite nel BPEL.

```
<definitions targetNamespace="http://www.echo.org">

  <message  name="echoMessage">
    <part  name="echoString"  type="xs:string" />
  </message>

  <portType      name="echoServicePT">
    <operation  name="echo">
      <input    message="echoMessage" />
      <output   message="echoMessage" />
    </operation>
  </portType>

  <binding  name="echoServiceBinding"
            type="echoServicePT">
    <!-- binding standard a SOAP -->
    ...
  </binding>

  <service  name="echoService">
    <!-- endpoint del servizio -->
    ...
  </service>

  <plt:partnerLinkType  name="echoServicePLT">
    <plt:role           name="service">
      <plt:portType     name="echoServicePT" />
    </plt:role>
  </plt:partnerLinkType>

</definitions>
```

Listato 3: definizioni WSDL per il processo echoProcess

Nell'esempio si nota nettamente la differenza tra i costrutti propri del WSDL e l'estensione `<partnerLinkType>` preceduta da un apposito *namespace*. Prima di analizzare in dettaglio l'uso dei vari riferimenti, verranno riprese le nozioni fondamentali sul WSDL.

Un documento WSDL è diviso in due parti: le definizioni *astratte* e le *definizioni concrete*. Le definizione astratte comprendono tutto ciò che riguarda la descrizione "ad alto livello" di uno o più web service, in particolare:

- i tipi XML utilizzati come ingresso ed uscita dai web service;
- il formato dei messaggi scambiati;
- l'insieme delle operazioni che è possibile effettuare su ciascun web service.

Le definizioni concrete comprendono invece la cosiddetta *materializzazione* del servizio, ovvero il binding al protocollo di trasporto (tipicamente *SOAP-over-HTTP*) e l'URL al quale è possibile contattare una *istanza* del servizio.

Un documento WSDL ha tipicamente la forma riportata nel listato 4.

```
<definitions>

  <types>
    <xs:schema>
      <xs:complexType name="aType">
        ...
      </xs:complexType>
      <xs:complexType name="anotherType">
        ...
      </xs:complexType>
      <!-- altri tipi complessi o semplici -->
    </xs:schema>
  </types>

  <message name="aMessage">
    <part name="aPart" type="aType" />
    <part name="anotherPart" type="anotherType" />
  </message>
  <message name="anotherMessage">
    <part name="anotherPart" type="anotherType" />
  </message>
  <!-- altri message -->

  <portType      name="myPortType">
    <operation  name="myFirstOperation">
      <input    message="aMessage" />
      <output   message="anotherMessage" />
    </operation>
    <operation  name="mySecondOperation">
      <input    message="anotherMessage" />
    </operation>
  </portType>
  <!-- altri portType -->

  <binding name="myBinding" type="myPortType">
    <soap:binding style="rpc" />
```

```
    <operation name="myFirstOperation>
      <soap:operation soapAction="" />
      <input><soap:body use="literal" /></input>
      <output><soap:body use="literal" /></output>
    </operation>
    <operation name="mySecondOperation>
      <soap:operation soapAction="" />
      <input><soap:body use="literal" /></input>
    </operation>
  </binding>
  <!-- altri binding -->

  <service  name="myService">
    <port  name="myServicePort" binding="myBinding">
      <soap:address location="http://myservice.com" />
    </port>
  </service>
  <!-- altri service -->

</definitions>
```

Listato 4: struttura di base di un documento WSDL

È evidente la separazione tra le definizioni astratte, puramente descrittive, e le definizioni concrete, dove si va a specificare quale debba essere il formato SOAP del messaggio inviato. Questo meccanismo rispecchia la filosofia di *generalizzazione della descrizione* propria degli standard W3C. Un altro aspetto che emerge con evidenza dal listato è la *riusabilità* degli elementi definiti nel documento, ad esempio:

- un `<type>` può essere riferito da più `<message>`;
- diverse `<operation>` possono avere lo stesso `<message>` come `<input>` o come `<output>`;
- più `<binding>` possono fare riferimento allo stesso `<portType>`, magari usando protocolli o formati diversi;
- più `<service>` possono descrivere endpoint diversi per uno stesso `<binding>` (*replicabilità del servizio*).

BPEL utilizza questi costrutti che fanno parte del "*core*" di WSDL nella seguente maniera:

- le variabili BPEL *devono* avere come tipo un messaggio (vedi § 2.2.1.2.2);
- la gestione delle eccezioni *può* (ma non necessariamente *deve*) fare riferimento a *SOAP fault* definiti in un `<binding>` (vedi § 2.2.1.2.4);
- quando si invoca o quando di attende una chiamata da un web service, *devono* essere specificati `<portType>` ed `<operation>` (vedi § 2.2.1.2.1).

Le due estensioni più importanti che vengono introdotte da BPEL per descrivere le interazioni tra i web service sono i *partner link* e le *proprietà*.

Quando due web service interagiscono tra loro (oppure quando un interprete BPEL interagisce con un web service), l'interazione viene modellata tramite un <partnerLink>. Ogni <partnerLink> a sua volta può essere considerato una specifica istanza di una classe di <partnerLink> con caratteristiche simili: tale classe prende il nome di <partnerLinkType>. Comprendere il significato dei <partnerLink> è essenziale per capire le potenzialità descrittive e di generalizzazione del BPEL.

Un esempio può chiarire meglio:

```
<!-- nel WSDL -->
<partnerLinkType name="myPartnerLinkType">
  <role  name="service">
    <portType name="aServicePortType" />
  </role>
  <role  name="client">
    <portType name="aClientPortType" />
  </role>
</partnerLinkType>

<!-- nel BPEL -->
<partnerLinks>
  <partnerLink  name="myClient"
                partnerLinkType="myPartnerLinkType"
                myRole="client" />
  <partnerLink  name="myService"
                partnerLinkType="myPartnerLinkType"
                partnerRole="service" />
</partnerLinks>
...
<invoke   name="anInvoke"
          partnerLink="myService"
          operation="myOperation"
          portType="aServicePortType"
          inputVariable="myInputVariable"
          outputVariable="myOutputVariable" />
```

Listato 5: utilizzo dei <partnerLink> in BPEL

Il <partnerLinkType> myPartnerLinkType modella a livello astratto una classe di interazioni tra un servizio ed un client. I due partner coinvolti hanno rispettivamente come interfaccia aServicePortType e aClientPortType. I <partnerLink> definiti nel BPEL istanziano la relazione assegnando il ruolo di client all'interprete BPEL ed il ruolo di servizio ad un web service esterno (myRole="client" e partnerRole="service"). Un riferimento al <partnerLink> viene poi usato al momento dell'invocazione del servizio (vedi § 2.2.1.2.1).

Sui <partnerLink> ci sono dei problemi molto rilevanti, discussi approfonditamente nel § 2.2.3.1.

Si può definire una regola empirica per la definizione dei <partnerLink> e dei <partnerLinkType>, che può essere d'aiuto per evitare errori:

- per ogni interazione tra due web service, definire un `<partnerLinkType>`;
- per ogni `<portType>` coinvolto nell'interazione, definire un `<role>` nel `<partnerLinkType>`;
- per ogni ruolo definito in un `<partnerLinkType>`, definire un `<partnerLink>`.

Quindi ad esempio per una chiamata sincrona ad un web service da parte del processo BPEL si dovrà:

- definire un `<partnerLinkType>`;
- definire un solo ruolo: dal momento che la chiamata è sincrona, per spedire indietro la risposta verrà usato lo stesso socket aperto dalla richiesta. È quindi coinvolto un solo `<portType>`, quello del servizio chiamato;
- definire un unico `<partnerLink>` al quale farà riferimento la `<invoke>` sincrona.

Per i dettagli sulla modalità di chiamata sincrona, si rimanda al § 2.2.3.2.

L'altro costrutto introdotto dal BPEL nel WSDL è quello delle *proprietà*. Una proprietà è un alias per più variabili BPEL di uno stesso tipo (§ 2.2.1.2.2) e viene usata per riferire tramite un unico nome diversi oggetti. L'utilità delle proprietà risulta evidente nel momento in cui vengono introdotti i *correlation set* (vedi § 2.2.1.2.3).

```
<property         name="myProperty"
                  type="xs:string" />
<propertyAlias    propertyName="myProperty"
                  messageType="aMessage"
                  part="aPart"
                  query="/aPartElement" />
<propertyAlias    propertyName="myProperty"
                  messageType="anotherMessage"
                  part="anotherPart"
                  query="/anotherPartElement" />
```

Listato 6: definizione di proprietà in BPEL

Ovviamente, gli elementi `aPartElement` e `anotherPartElement` del listato 6 dovranno essere di tipo `xs:string`. In questa maniera, a seconda del contesto (messaggio) a cui si fa riferimento, l'interprete è in grado di mappare il valore della proprietà su quello di uno dei due elementi.

2.2.1.2 Costrutti principali

2.2.1.2.1 Interazione con servizi web

BPEL è un linguaggio che permette di coordinare il flusso di chiamate a web service: è naturale che i costrutti principali di tale linguaggio siano quelli per l'interazione con i servizi stessi.

L'attività `<invoke>`, già vista nel listato 5, è usata per chiamare una operazione di un web service. La chiamata può essere effettuata in maniera sincrona o asincrona: la distinzione tra i due casi avviene in maniera implicita e non specificando un attributo o un elemento particolare. Quando si chiama un servizio, è necessario infatti fornire sempre la variabile di ingresso; se viene specificata anche una variabile di uscita, la chiamata sarà sincrona e quindi bloccante. Quindi, una `<invoke>` che presenta sia una `inputVariable` che una `outputVariable` causerà un blocco dell'istanza di processo fino a quando non verrà ricevuta una risposta.

Nel caso in cui non sia presente una variabile di output, ci sono due possibilità: o il web service è di tipo *one-way* e quindi non fornisce una risposta in uscita, oppure il web service viene chiamato in maniera asincrona e quindi il costrutto per ricevere la risposta verrà specificato più avanti nel processo. Per una analisi più approfondita delle modalità di invocazione di un servizio, si rimanda al § 2.2.3.2.

Ad una `<invoke>` asincrona possono seguire due costrutti diversi per gestire la risposta: una `<receive>` o una `<pick>`. La `<receive>` è l'operazione standard per ricevere un messaggio da un web service. Tipicamente, una `<receive>` è posta all'inizio del processo, che viene così attivato da un qualche client attraverso l'invio di un messaggio SOAP. In questo caso, è necessaria la presenza dell'attributo `createInstance="yes"` che segnala all'interprete l'attivazione di una nuova istanza del processo. La `<receive>` ha un formato come quello del listato 2: si nota come, analogamente alla `<invoke>`, nella `<receive>` si debba specificare in maniera univoca da quale web service e da quale operazione ci si aspetta che provenga il messaggio.

Il costrutto `<pick>` si differenzia invece perché è possibile scegliere il mittente del messaggio tra più web service o tra più `<operation>`. Quando si trova una `<pick>`, si apre una corsa tra i diversi servizi abilitati all'invio: soltanto il primo messaggio ricevuto viene processato e si prosegue nell'elaborazione come se ci si trovasse all'interno del costrutto `<switch>` (vedi anche § 2.2.1.2.5).

Un esempio di `<pick>` è riportato nel listato 7.

```
<pick>
  <onMessage  partnerLink="aClient"
              portType="aPortType"
              operation="anOperation"
              variable="myVariable">
    <!-- serie di attività -->
    ...
  </onMessage>
  <onMessage  partnerLink="anotherClient"
```

```
                portType="anotherPortType"
                operation="anotherOperation"
                variable="myVariable">
      <!-- serie di attività -->
      ...
    </onMessage>
</pick>
```

Listato 7: utilizzo di <pick> in BPEL

La `<pick>` risulta particolarmente utile per la gestione degli errori nelle chiamate asincrone (vedi § 2.2.3.2).

Un altro costrutto utile è la `<reply>` già vista nel listato 2: la sua funzione è quella di rispondere ad un partner dal quale si è ricevuto un messaggio e per il quale è disponibile una connessione HTTP ancora aperta. Tipicamente, questo costrutto si utilizza nei casi in cui l'istanza di processo si comporta come un servizio, ad esempio per fornire la risposta al client che ha attivato il processo.

2.2.1.2.2 Variabili

Le variabili in BPEL hanno una importanza particolare, in quanto rappresentano l'implementazione nell'istanza di processo dei messaggi astratti definiti nel WSDL. Una variabile può essere quindi vista come una stringa XML il cui schema è definito dal `<message>` a cui è associata.

Le variabili vengono dichiarate all'inizio del processo o internamente ad uno `<scope>` (vedi § 2.2.1.2.4), come nell'esempio del listato 8.

```
<!-- nel WSDL -->
<types>
  <xs:schema>
    <xs:complexType  name="aComplexType">
      <xs:sequence>
        <xs:element name="firstElement"
                    type="xs:string" />
        <xs:element name="firstElement"
                    type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
</types>

<message  name="myMessage">
  <part  name="firstPart"
         type="xs:string" />
  <part  name="secondPart"
         type="aComplexType" />
</message>

<!-- nel BPEL -->
<process>
  <partnerLinks>...</partnerLinks>
```

```
  <variables>
    <variable   name="aGlobalVariable"
                messageType="myMessage" />
  </variables>
  <sequence>
    <scope>
      <variables>
        <variable   name="aLocalVariable"
                    messageType="myMessage" />
      </variables>
    ...
    </scope>
  </sequence>
</process>
```

Listato 8: definizione di variabili in BPEL

Le variabili vengono utilizzate nelle `<invoke>`, nelle `<receive>` e nelle `<pick>` come contenitori per i dati inviati o ritornati dai web service; un costrutto speciale `<assign>` permette di copiare valori da una variabile all'altra, utilizzando il linguaggio *XPath* per indirizzare i vari elementi della stringa XML (vedi listato 9).

```
<assign     name="anAssign">
  <copy>
    <from   expression="Received" />
    <to     variable="aGlobalVariable"
            part="firstPart" />
  </copy>
  <copy>
    <from   variable="aLocalVariable"
            part="firstPart" />
    <to     variable="aGlobalVariable"
            part="secondPart"
            query="/secondElement" />
  </copy>
</assign>
```

Listato 9: assegnazione di valori alle variabili BPEL

Un altro utilizzo delle variabili è quello che ne viene fatto nel *fault handling*: possono infatti servire per incapsulare i SOAP fault lanciati da un web service (vedi § 2.2.1.2.4).

2.2.1.2.3 Correlazione

Quando una istanza di processo invoca un web service in modalità asincrona, tipicamente rimane in attesa della ricezione del messaggio di ritorno. Poiché in ogni momento possono essere attive più istanze di uno stesso processo, si pone il problema di determinare a quale fra queste debba essere recapito il messaggio di risposta entrante, ovvero quale sia l'istanza che ha invocato il servizio per quello specifico messaggio di risposta.

Questo è un aspetto del più ampio problema della *correlazione*, comune a molti protocolli di comunicazione. La correlazione è la capacità di associare messaggi tra loro in relazione logica in maniera da formare una *conversazione*. I messaggi di posta elettronica, ad esempio, sono identificati da un `Message-ID` univoco: quando si risponde ad un messaggio ricevuto, viene inserito nell'header della risposta il campo `In-Reply-To` con valore pari al `Message-ID` del messaggio a cui si fa riferimento: questo meccanismo consente ai client di posta elettronica di raggruppare i messaggi in *thread* utilizzando catene di riferimenti ai `Message-ID`.

Anche *WS-Addressing*, l'applicazione XML specificata da W3C come supporto agli header SOAP per l'indirizzamento ed il messaging, definisce un meccanismo di correlazione simile, generalizzando il concetto da `Message-ID` a `URI` (vedi [WSA]).

La filosofia di BPEL è però di non occuparsi di aspetti di basso livello ma soltanto della descrizione del processo. Questa scelta, per quanto riguarda la correlazione, è supportata da almeno due motivi:

1. la composizione dei messaggi in uscita è affidata all'interprete, non ci sono legami tra BPEL e SOAP e non è quindi possibile intervenire nell'header del messaggio per definire la correlazione;
2. eventualmente, non sarebbe possibile sapere se il meccanismo di correlazione è supportato o meno dal servizio invocato (è inutile e dannoso utilizzare il `Message-ID` nella chiamata se il servizio non supporta l'inserimento di `In-Reply-To` nella risposta).

La soluzione elegante proposta da BPEL comporta lo spostamento del problema dal livello di trasporto al livello applicativo, senza introdurre overhead e ridondanze. L'idea è quella di definire una sorta di *chiave primaria* per un processo: i valori di tali chiavi sono mappati su valori degli elementi dei messaggi scambiati, per cui non è necessario aggiungere informazioni di controllo.

Ad esempio, in un processo che riguarda l'acquisto di un libro via Internet, la chiave primaria potrebbe essere formata dal codice ISBN del libro e dallo username dell'acquirente, come in listato 10.

```
<!-- nel WSDL -->
<message name="buyBookRequest">
  <part name="ISBN"        type="xs:string" />
  <part name="clientEmail" type="xs:string" />
</message>
<message name="buyBookResponse">
```

```
  <part  name="status"      type="xs:string" />
  <part  name="ISBNRef"     type="xs:string" />
  <part  name="usernameRef" type="xs:string" />
</message>
<portType     name="BuyBookPT">
  <operation  name="buy">
    <input    message="buyBookRequest" />
  </operation>
</portType>
<portType     name="BuyBookCallbackPT">
  <operation  name="buyCallback">
    <output   message="buyBookResponse" />
  </operation>
</portType>
...
<property        name="userID"    type="xs:string" />
<property        name="ISBNID"    type="xs:string" />
<propertyAlias   propertyName="userID"
                 messageType="buyBookRequest"
                 part="clientEmail"
                 query="/" />
<propertyAlias   propertyName="userID"
                 messageType="buyBookResponse"
                 part="usernameRef"
                 query="/" />
<propertyAlias   propertyName="ISBNID"
                 messageType="buyBookRequest"
                 part="ISBN"
                 query="/" />
<propertyAlias   propertyName="ISBNID"
                 messageType="buyBookResponse"
                 part="ISBNRef"
                 query="/" />

<!-- nel BPEL -->
<correlationSet  name="bbcs" properties="userID ISBNID" />
...
<invoke   name="asyncInvoke"
          portType="BuyBookPT"
          operation="buy"
          ...>
    <correlations>
      <correlation  set="bbcs"
                    initiate="yes"
                    pattern="out" />
    </correlations>
</invoke>
<receive  name="asyncReceive"
          portType="BuyBookCallbackPT"
          operation="buyCallback"
          ...>
```

```
    <correlations>
      <correlation  set="bbcs"
                    initiate="no"
                    pattern="in" />
    </correlations>
</receive>
```

Listato 10: definizione ed uso dei correlation set in BPEL

Nel listato, la `<invoke>` inizializza (`initiate="yes"`) il `<correlationSet>` con i valori contenuti negli elementi `<clientEmail>` e `<ISBN>` del messaggio in uscita (`pattern="out"`): l'interprete sa che, quando riceverà un messaggio contenente gli stessi valori rispettivamente negli elementi `<username>` e `<ISBNRef>`, dovrà inoltrare tale messaggio all'istanza di processo che ha richiesto la correlazione. Per maggiori dettagli sul pattern di chiamata asincrona con callback utilizzato nel listato si rimanda al § 2.2.3.2.

In questo esempio si capisce la funzione delle `<property>` già descritte nel § 2.2.1.1.1: lo `userID` per il servizio chiamante è un indirizzo email e per il servizio chiamato uno username, mentre il codice ISBN viene riferito nei messaggi con due nomi diversi. L'introduzione del livello di astrazione delle proprietà consente proprio l'utilizzo di nomi differenti per i campi della chiave primaria nei vari messaggi (notare però che il tipo deve essere sempre lo stesso).

L'unica controindicazione di un meccanismo del genere è la necessità di avere riportati in tutti i messaggi coinvolti nella conversazione elementi che fanno riferimento alle chiavi primarie: si tratta di un problema di poco conto se il processo è costituito da servizi costruiti ad hoc, più complesso diventa invece nel caso in cui il processo sia composto da servizi eterogenei preesistenti.

I pattern di scambio dei `<correlationSet>` possono diventare anche molto complessi: ad esempio, una `<receive>` può allo stesso tempo inizializzare un nuovo `<correlationSet>` e fare riferimento ad uno precedentemente inizializzato. I casi d'uso tipici del meccanismo sono però sostanzialmente due:

- la chiave primaria viene utilizzata in tutto il processo e quindi inizializzata nella prima `<receive>` che attiva l'istanza di processo;
- la chiave primaria viene utilizzata per correlare richiesta e risposta di un servizio asincrono come visto nel listato 10.

2.2.1.2.4 Visibilità, gestione degli errori e compensazione

Uno degli aspetti del BPEL che maggiormente lo avvicinano ad un linguaggio di programmazione procedurale è la presenza del costrutto `<scope>`, che introduce livelli di visibilità diversi per variabili e `<correlationSet>`, esattamente come avviene nei linguaggi *C-like* con i blocchi racchiusi tra parentesi graffe. Persino la terminologia è simile: si parla di *variabili locali* e *globali*, e le variabili locali possono mascherare variabili definite al livello superiore tramite l'omonimia (non è presente però un costrutto che faccia da risolutore della visibilità, ovvero un analogo dell'operatore `::` del C++).

BPEL mette inoltre a disposizione un meccanismo *throw-catch* per la gestione degli errori, concettualmente molto simile a quello presente nei linguaggi di programmazione più evoluti. Ogni blocco `<catch>` è legato ad uno `<scope>`: se un fault non viene catturato all'interno dello `<scope>` dove è stato creato, la gestione passa al blocco di livello superiore.

All'interno di questo meccanismo vengono mappati anche i SOAP fault eventualmente lanciati dai web service invocati nel processo. Tali fault devono però essere definiti nel `<portType>` e nel `<binding>` del servizio.

Nel listato 11 è presente un esempio di vari utilizzi di `<scope>` e gestione dei fault.

```
<!-- nel WSDL -->
<portType      name="aService">
  <operation   name="anOperation">
    <input     message="anInputMessage" />
    <ouput     message="anOutputMessage" />
    <fault     message="myFaultMessage" />
  </operation>
</portType>

<!-- nel BPEL -->
<process>
  <variables>
    <variable   name="aGlobalVariable"
                messageType="anInputMessage" />
  </variables>

  <sequence>
    ...
    <assign>
      <copy>
        <from   expression="'a sample text'" />
        <to     variable="aGlobalVariable"   ... />
      </copy>
    </assign>
    <scope>
      <variables>
        <variable   name="aLocalVariable"
                    messageType="anOutputMessage" />
```

```
      </variables>
      <faultHandlers>
        <catch faultName="myFault">
          <sequence>
            ...
          </sequence>
        </catch>
        <catch faultName="anotherFault">
          <sequence>
            ...
          </sequence>
        </catch>
      </faultHandlers>
      <invoke   name="myInvoke"
                portType="aService"
                operation="anOperation"
                inputVariable="aGlobalVariable"
                outputVariable="aLocalVariable" />
      <switch>
        <case  condition="...">
          <throw faultName="anotherFault" />
        </case>
      </switch>
    </scope>
  </sequence>
</process>
```

Listato 11: uso di <scope> e meccanismo di gestione dei fault in BPEL

Nel listato sono definite due variabili, una a livello globale, e quindi visibile ed utilizzabile nell'intero processo, ed una privata allo `<scope>`. Sono riportati anche entrambe le modalità di lancio di un fault: attraverso la mappatura con un SOAP fault definito nel WSDL (nel caso di `myFault`) ed attraverso l'utilizzo esplicito di `<throw>`. Notare come `anotherFault` sia definito implicitamente al momento dell'utilizzo e non in precedenza.

L'elemento `<faultHandlers>` può anche essere specificato internamente ad una singola `<invoke>` o `<receive>` invece che a livello di `<scope>`. In questa maniera, si definisce un blocco implicito contenente la sola attività di livello superiore al gestore del fault (vedi listato 12).

```
<invoke   name="myInvoke"
          portType="aService"
          operation="anOperation"
          inputVariable="aGlobalVariable"
          outputVariable="aLocalVariable">
        <faultHandlers>
          <catch faultName="myFault"> ... </catch>
        </faultHandlers>
</invoke>
```

Listato 12: esempio di uso di <faultHandlers> internamente ad una <invoke> in BPEL

Il meccanismo di gestione dei fault viene esteso e generalizzato nel BPEL anche ad altri tipi di eventi, tramite l'utilizzo di un blocco `<eventHandlers>` simile al `<faultHandlers>`. Gli eventi possono essere di due tipi: `<onMessage>` alla ricezione di un messaggio (usato in maniera identica a quanto viene fatto nella `<pick>`, vedi § 2.2.1.2.1) oppure `<onAlarm>` quando scade un timeout. È interessante sottolineare che la durata dei timeout può anche essere calcolata a *run-time* sulla base di valori di tipo `xs:duration` presenti nei messaggi.

L'ultimo costrutto associato alla visibilità analizzato in questo paragrafo è `<compensationHandler>`: questo tipo di gestore permette di definire una serie di operazioni considerate di *rollback* o di *undo* per uno `<scope>`. Il rollback tipicamente viene utilizzato nel caso in cui si ha un fallimento e, quindi, l'attività `<compensate>`, che attiva la compensazione per uno specifico blocco, si trova di solito all'interno di un `<catch>` o di uno `<switch>`. Un esempio è riportato nel listato 13.

```
<process>
  ...
  <faultHandlers>
    <catch faultName="myFault">
      <compensate scope="firstScope" />
    </catch>
  </faultHandlers>
  ...
  <scope name="firstScope">
    <compensationHandler>
      <!-- attività per il rollback -->
    </compensationHandler>
    ...
  </scope>
</process>
```

Listato 13: utilizzo della compensazione in BPEL

Maggiori informazioni sui blocchi di visibilità e sugli eventi possono essere trovate nelle specifiche BPEL (vedi [BPEL]) ed in [IBMBPEL6], [IBMBPEL7] e [IBMBPEL8].

2.2.1.2.5 Controllo di flusso

BPEL ha alcuni costrutti che permettono di gestire il flusso del processo in maniera molto efficiente, potente e performante. In questo paragrafo viene dato soltanto un accenno ai vari elementi e qualche esempio.

Un blocco di azioni può essere eseguito in sequenza o in parallelo. Nel primo caso tali operazioni sono all'interno di un elemento `<sequence>`, nel secondo caso di un elemento `<flow>`. Nel caso in cui le azioni (`<invoke>`, `<receive>`, `<pick>`, `<assign>` e le altre viste) siano eseguite in parallelo, è possibile determinare alcuni punti di sincronizzazione attraverso l'espressione di dipendenze temporali con i costrutti `<link>`, `<target>` e `<source>`. L'utilizzo di blocchi di sequenze parallelizzate è particolarmente utile nel caso di chiamate asincrone per minimizzare il tempo di esecuzione del processo.

I costrutti `<switch>` e `<while>` sono del tutto analoghi agli omonimi costrutti dei linguaggi di programmazione: `<switch>` permette l'esecuzione di codice condizionale mentre `<while>` consente la ripetizione di una serie di azioni. Interessante è la possibilità di utilizzare valori interni alle variabili (calcolati a tempo di esecuzione) per il calcolo delle espressioni condizionali, in maniera simile ai linguaggi di programmazione imperativi e a oggetti. Anche il costrutto `<pick>`, già descritto nel § 2.2.1.2.1, può essere considerato un costrutto per il controllo del flusso.

In figura 12 e nel listato 14 è riportato un esempio di utilizzo dei vari elementi citati. La freccia tratteggiata rappresenta una dipendenza di sincronizzazione temporale tra due `<invoke>` parallele.

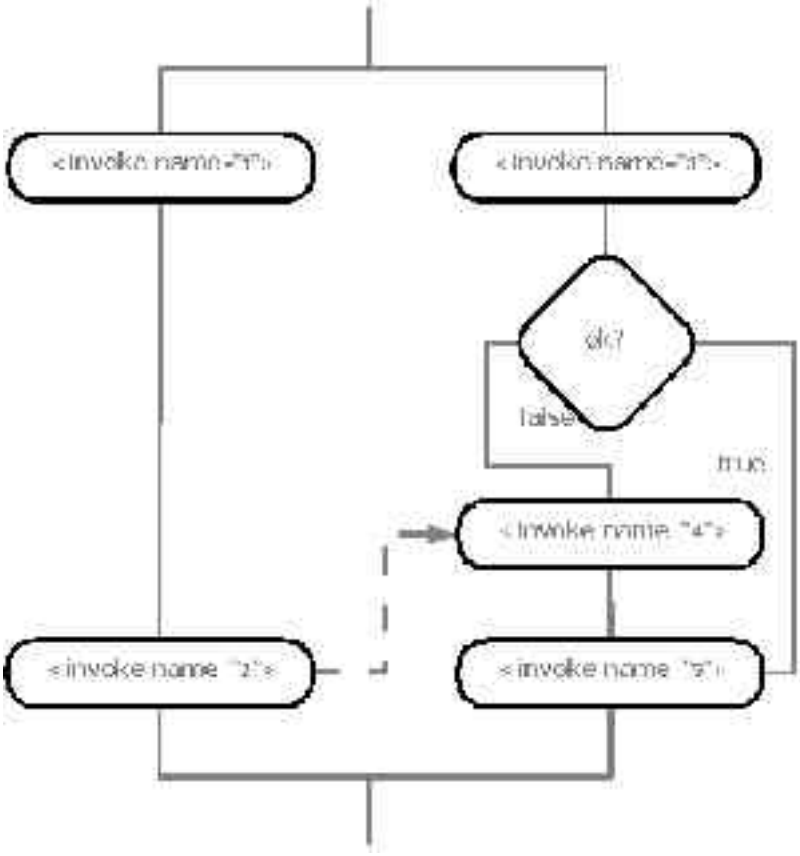

Figura 12: rappresentazione grafica di un flusso di processo BPEL

```
<flow>
  <link name="2-to-5" />
  <sequence>
    <invoke  name="1" />
    <invoke  name="2">
      <source  linkName="2-to-5" />
    </invoke>
  </sequence>
  <sequence>
    <invoke  name="3" />
    <switch>
      <case
        condition="getVariableData('ok', 'text') = 'true'">
        <empty />
      </case>
      <case
        condition="getVariableData('ok', 'text') = 'false'">
        <sequence>
          <invoke  name="4">
            <target  linkName="2-to-5" />
          </invoke>
        </sequence>
      </case>
    </switch>
    <invoke  name="5" />
  </sequence>
</flow>
```

Listato 14: utilizzo dei costrutti di controllo BPEL

2.2.2 Le applicazioni

2.2.2.1 Sviluppo di applicazioni ad hoc in ambiente distribuito

L'applicazione più semplice del BPEL è il suo utilizzo come *framework* per lo sviluppo di applicazioni distribuite. *Zlatan*, il progetto analizzato nel § 3, dimostra come un interprete BPEL possa essere inserito a far parte di una applicazione progettata *ad hoc*. In particolare, in questi casi vengono sfruttate le capacità dell'interprete di gestire istanze multiple del processo e di mantenere uno stato associato ad ognuna di queste istanze grazie alla presenza delle variabili e dei correlation set.

L'utilizzo di un interprete per coordinare le varie fasi del processo obbliga a progettare una applicazione modulare e quindi a generare codice riusabile; viene inoltre ridotto il tempo di produzione dell'applicazione grazie alle funzionalità già disponibili nell'interprete che non devono essere implementate da zero. Inoltre, scegliendo un unico ambiente di sviluppo per i servizi che compongono l'applicazione, preferibilmente un ambiente pienamente compatibile con *WS-I*, si riducono i problemi dovuti all'interoperabilità di piattaforme diverse, concentrandosi sulle problematiche applicative piuttosto che su quelle di comunicazione.

2.2.2.2 Integrazione di sistemi legacy

Un'altra interessante possibilità di applicazione del BPEL e delle altre tecnologie che implementano il SOA è quella dell'integrazione di *sistemi legacy* (vedi [IBMLEG]).

Tipicamente, una azienda possiede delle applicazioni software e delle banche dati di vecchia concezione che non possono essere facilmente trasportate su nuove piattaforme per problemi tecnologici, organizzativi o più spesso economici. Il SOA permette di recuperare ed integrare le funzionalità di queste applicazioni in maniera sicura: la soluzione consiste nell'applicare una interfaccia coerente con l'architettura orientata ai servizi e di connettere tale interfaccia al programma che esegue la funzione.

Gli strumenti software che implementano questa soluzione sono detti *adapter*. Un adapter è un tool che permette di:

1. generare la descrizione WSDL di una applicazione legacy;
2. esporre tale interfaccia sul web;
3. intercettare i messaggi SOAP diretti all'applicazione legacy;
4. attivare l'applicazione legacy inizializzandola con i valori contenuti nel messaggio SOAP;
5. attendere la fine del task di esecuzione dell'applicazione legacy;
6. parserizzare l'output dell'applicazione legacy e tradurlo in formato SOAP;
7. inviare la risposta al client chiamante.

Gli adapter stanno avendo un successo crescente nel mercato dell'EAI (*Enterprise Application and Integration*): aziende come IBM, Tibco e IONA producono adapter molto sofisticati, mentre Sun ha recentemente definito le specifiche per creare l'implementazione di adapter (*connector* nella terminologia Java) che connettano direttamente J2EE a sistemi legacy o di terze parti (vedi [JCA]), tra cui SAP (vedi § 2.3).

La suite *GlobalBiz* dispone di un adapter integrato con il *Process Engine* per utilizzare come servizi web procedure batch ed eseguibili (vedi § 2.2.4.1).

2.2.2.3 Evoluzione di processi di business da servizi esistenti

Uno degli obiettivi della SOA è quello di creare processi di business distribuiti su Internet integrando servizi offerti da partner diversi.

Un esempio tipico è quello della prenotazione di un biglietto aereo: un processo che comporta la collaborazione tra cliente, agenzia di viaggi, compagnia aerea, aeroporto ed altri partner (es. per ricerca disponibilità parcheggio, noleggio auto o prenotazione hotel). Ognuno degli attori del sistema offre servizi semplici o composti in processi, implementati su piattaforme e con tecnologie eterogenee, tutti comunque esposti come web service. BPEL consente di coordinare tali servizi in un unico processo che appare al cliente finale come una *transazione atomica*, mascherando completamente la complessità gestionale, organizzativa e tecnologica del sistema.

Una delle caratteristiche di BPEL che supportano meglio questa possibilità è la gestione delle cosiddette *long-run transaction*, ovvero delle transazioni che hanno durata di scala molto maggiore rispetto ai tempi dell'elaborazione *on-line*. Ad esempio, un servizio potrebbe prevedere che la conferma di una prenotazione venga autorizzata a mano da un utente del sistema partner: i tempi di risposta al cliente in questo caso potrebbero diventare ore o addirittura giorni.

Le specifiche del BPEL prevedono che gli interpreti siano in grado di gestire queste situazioni, mantenendo i processi nello stato attivo a tempo indeterminato, e comprendono anche l'uso di eventi scatenati da timeout, come visto nel § 2.2.1.2.4, per garantire soglie di tempi massimi nell'elaborazione del processo.

L'altro meccanismo particolarmente utile è la `<compensate>`, che permette di implementare un sistema di *rollback* modulare e a cascata per rendere veramente atomica una transazione composta da più servizi.

2.2.3 Limiti attuali e sviluppi futuri

2.2.3.1 Attributi ridondanti ed attributi mancanti

All'applicazione pratica del BPEL, come già detto, emergono problemi rilevanti che derivano da una progettazione mirata troppo alla generalizzazione descrittiva e poco alle problematiche implementative.

Un esempio lampante proviene dall'utilizzo dei `partnerLink` nelle `<invoke>`. Come già visto, una `<invoke>` è caratterizzata dalla presenza, tra gli altri attributi, di `partnerLink`, `portType` ed `operation`. Dal `partnerLink` è possibile risalire al `<partnerLinkType>` e, tramite l'attributo `partnerRole`, al `<portType>` del partner: non è possibile però risalire ad un `<binding>` né tanto meno ad un `<service>` e quindi ad una URL alla quale sia contattabile il servizio. In sostanza, l'interprete conosce tutti i dettagli astratti sul web service partner, ma non ne conosce l'indirizzo e non può quindi inviare il messaggio SOAP.

Inoltre, c'è una evidente ridondanza nel momento in cui nella `<invoke>` viene specificato il `<portType>`: questo deve essere necessariamente uguale al `<portType>` definito per relativo `<partnerLinkType>`; in caso contrario, l'effetto potrebbe essere addirittura dannoso e generare un comportamento inaspettato da parte dell'interprete.

Il *Process Engine* di *GlobalBiz* (PE - vedi § 2.2.4.1) e l'interprete di *ActiveBPEL* (§ 2.2.4.3) risolvono il problema dell'indirizzamento in due maniere completamente opposte da un punto di vista filosofico.

PE introduce una estensione al linguaggio per dichiarare, nella definizione del `<partnerLink>`, qual è la `<port>` WSDL (ovvero la materializzazione del web service) a cui fare riferimento. In questa maniera non si introducono altre ridondanze, perché la URL del servizio viene definita una sola volta (nel WSDL), e si rispetta la struttura del BPEL, sfruttando la possibilità, prevista dalle specifiche XML, di inserire estensioni ai linguaggi mappandole su appropriati namespace.

ActiveBPEL introduce invece un terzo documento, detto *Process Deployment Descriptor*, che contiene, tra le altre informazioni, gli *endpoint* dei servizi web. Questo meccanismo è più flessibile rispetto al precedente (un endpoint, ad esempio, può essere specificato attraverso *WS-Addressing*), ma compromette pesantemente la portabilità del processo BPEL generato e, allontanandosi dalle specifiche *W3C* del WSDL e da quelle del BPEL, introduce possibilità di ridondanza e di errori. Nei listati 15 e 16 sono riportati esempi di risoluzione dell'endpoint nei due interpreti.

```
<!-- nel WSDL -->
<portType   name="myPortType">...</portType>
<binding    name="myBinding" type="myPortType">...</binding>
<service    name="myService">
  <port  name="myServicePort"
         binding="myBinding">
    <soap:address location="http://myexample.com/myservice" />
  </port>
```

```
</service>
<plt:partnerLinkType    name="myPLT">
  <plt:role name="myRole">
    <plt:portType    name="myPortType" />
  </plt:role>
</plt:partnerLinkType>

<!-- nel BPEL -->
<partnerLink  name="myPL"
              partnerLinkType="myPLT"
              partnerRole="myRole"
              bpelext:partnerPort="myServicePort" />
```

Listato 15: risoluzione dell'endpoint in GlobalBiz PE

```
<!-- nel WSDL -->
<portType   name="myPortType">...</portType>
<plt:partnerLinkType    name="myPLT">
  <plt:role name="myRole">
    <plt:portType    name="myPortType" />
  </plt:role>
</plt:partnerLinkType>

<!-- nel BPEL -->
<partnerLink  name="myPL"
              parterLintType="myPLT"
              partnerRole="myRole" />

<!-- nel file myexample.pdd -->
<process  name="myExampleProcess">
  <partnerLinks>
    <partnerLink  name="myPL">
      <partnerRole  endpointReference="static">
         http://myexample.com/myservice
      </partnerRole>
    </partnerLink>
  </partnerLinks>
</process>
```

Listato 16: risoluzione dell'endpoint in ActiveBPEL

2.2.3.2 Web service asincroni

Il WSDL prevede di poter specificare a livello astratto quattro diversi modelli di scambio dei messaggi per l'invocazione di una operazione di un servizio web (vedi [WSDL]):

- *Request-response*: un messaggio di input seguito da un messaggio di output;
- *One-way*: un messaggio di input senza output;
- *Solicit-response*: un messaggio di output seguito da un messaggio di input;
- *Notification*: un messaggio di output senza input.

Questi modelli non sono tuttavia completamente supportati al momento della definizione concreta: il binding SOAP, a causa delle limitazione imposta dal *WS-I Basic Profile*, permette di utilizzare soltanto i primi due modelli (vedi [WSIBP]).

Un errore comune è quello di credere che le operazioni di tipo *request-response* siano sincrone e quelle di tipo *one-way* siano asincrone: questi modelli specificano soltanto il pattern di scambio dei messaggi, non il comportamento del servizio al momento della chiamata. In Microsoft .NET, ad esempio, un web service definito come request-response può essere invocato in entrambe le modalità: sincrona bloccante oppure asincrona, tramite l'utilizzo di due metodi `beginOperation()` ed `endOperation()` generati automaticamente dal compilatore WSDL di .NET per ogni operazione.

Né WSDL né BPEL dispongono di meccanismi chiari per l'identificazione della modalità di chiamata (sincrona o asincrona): ai problemi ed alle difficoltà interpretative generate da questi deficit dei linguaggi, si aggiunge la difficoltà di utilizzo di SOAP sopra un protocollo sincrono e non affidabile come HTTP (vedi anche § 2.1.4.2).

HTTP è infatti basato su un modello richiesta-risposta nel quale la connessione TCP/IP rimane aperta fino a quando la comunicazione non è conclusa. Il problema nasce quando un protocollo con queste caratteristiche viene utilizzato come trasporto nel mondo *loosely coupled* dei web service. A questo proposito, le specifiche del *WS-I Basic Profile* riportano che "*an INSTANCE SHOULD use either a "200 OK" or "202 Accepted" HTTP status code for a response message that does not contain a SOAP envelope but indicates the successful outcome of a HTTP request*": in altre parole, un web service one-way, che quindi non prevede risposta, dovrebbe comunque spedire indietro un messaggio HTTP vuoto con codice 200 o 202 per permettere al client di chiudere la connessione.

Axis (vedi Appendice A) non rispetta queste specifiche. Quando il tool di compilazione *WSDL2Java* genera le classi per la realizzazione dei servizi, questi vengono sempre implementati in maniera da essere chiamati soltanto in modalità sincrona. Le operazioni che nel WSDL erano dichiarate come one-way vengono tradotte in metodi `void` e forniscono sempre una risposta vuota a livello SOAP. Axis genera anche un nuovo WSDL per ogni servizio

implementato, in maniera da allineare la descrizione del servizio al comportamento reale.

Anche a causa di questi problemi interpretativi, è nato un pattern di descrizione dei web service asincroni ormai molto diffuso, basato essenzialmente sul noto meccanismo delle chiamate *callback*. In questo pattern si descrive il web service chiamato come one-way e si definisce nel `<portType>` del chiamante una operazione di callback invocata dal chiamato al momento della risposta asincrona. Il WSDL appare simile a quello di seguito:

```
<!-- portType del chiamato -->
<portType      name="asynchService">
  <operation   name="asyncOperation">
    <input     message="aMessage" />
  </operation>
</portType>

<!-- portType del chiamante -->
<portType      name="client">
  <operation   name="anOperation">
    <input     message="anotherMessage" />
    <output    message="aThirdMessage" />
  </operation>
  <operation   name="asyncOperationCallback">
    <input     message="aMessageCallback" />
  </operation>
</portType>
```

Listato 17: definizioni WSDL per una chiamata asincrona con callback

In questa maniera, il servizio chiamante (tipicamente, il processo BPEL) attiva l'operazione `asyncOperation` con una chiamata one-way sul servizio chiamato, il quale notificherà la fine delle operazioni ed invierà il risultato in maniera asincrona invocando l'operazione `asyncOperationCallback` sul `<portType>` del chiamante. Lo svantaggio principale di questo pattern è che entrambi i partner devono avere un `<portType>` definito ed essere implementati come servizi web.

Si possono quindi classificare nel BPEL almeno tre tipologie di chiamate a servizi:

- *servizi request-response chiamati in maniera sincrona*;
- *servizi one-way* `void`;
- *servizi request-response chiamati in maniera asincrona con callback*.

I servizi del primo tipo sono concettualmente i più semplici da comprendere e praticamente i più facili da implementare. Un esempio è fornito nel listato 18.

```
<!-- nel WSDL -->
<portType      name="myPortType">
  <operation   name="myOperation">
    <input     message="myInputMessage" />
    <output    message="myOutputMessage" />
```

```
  </operation>
</portType>

<!-- nel BPEL -->
<invoke   partnerLink="myPartnerLink"
          portType="myPortType"
          operation="myOperation"
          inputVariable="myInputVariable"
          outputVariable="myOutputVariable" />
```

Listato 18: invocazione sincrona di un servizio in BPEL

In questo caso, l'istanza del processo BPEL invia il messaggio SOAP di richiesta contenente `myInputVariable` e si blocca in attesa di ricevere una risposta a livello SOAP contenente `myOutputVariable`. La risposta viene fornita sulla stessa connessione HTTP servita per inviare la chiamata.

Per i servizi del secondo tipo, non c'è distinzione nella modalità di chiamata: l'istanza di processo invia `myInputVariable` ed attende la chiusura della connessione a livello HTTP.

Per i servizi del terzo tipo, il meccanismo è più complicato: è necessario innanzi tutto definire il *correlation set* per distinguere l'istanza di processo che deve elaborare la risposta ricevuta dal web service (vedi § 2.2.1.2.3), oltre a dover utilizzare una `<invoke>` per la chiamata ed una `<receive>` o una `<pick>` (vedi § 2.2.1.2.1) per la risposta. Un esempio è riportato nel listato 19.

```
<!-- nel WSDL -->
<portType      name="myPortType">
  <operation   name="myOperation">
    <input     message="myInputMessage" />
  </operation>
</portType>
<portType      name="myCallbackPortType">
  <operation   name="myCallbackOperation">
    <input     message="myInputMessage" />
  </operation>
</portType>

<!-- nel BPEL -->
<invoke   partnerLink="myPartnerLink"
          portType="myPortType"
          operation="myOperation"
          inputVariable="myInputVariable">
  <correlation   set="myCorrSet"
                 initiate="yes"
                 pattern="out" />
</invoke>

<receive  partnerLink="myPartnerLink"
          portType="myCallbackPortType"
          operation="myCallbackOperation"
          variable="myOutputVariable">
```

```
  <correlation  set="myCorrSet"
                initiate="no"
                pattern="in" />
</receive>
```

Listato 19: invocazione asincrona con callback in BPEL

Le operazioni che l'interprete BPEL deve effettuare sono:

- istanziare un `<correlation>` per l'istanza di processo attiva, inizializzandolo con i valori contenuti in `myInputVariable` in base al mapping definito con le proprietà (vedi § 2.2.1.1.1 e § 2.2.1.2.3);
- invocare l'operazione `myOperation` sul servizio web, compilando il messaggio SOAP con i valori della variabile `myInputVariable`;
- attendere la chiusura delle connessione HTTP da parte del servizio chiamato;
- mettersi in attesa di una risposta asincrona da parte del servizio chiamato e, al momento della ricezione, assegnare alla variabile `myOutputVariable` i valori contenuti nel messaggio di risposta;
- chiudere la connessione HTTP con il servizio chiamato.

L'utilizzo delle operazioni di callback per la chiamata dei web service asincroni comporta dei problemi nella gestione delle eccezioni. Come visto nel § 2.2.1.2.4, i SOAP fault generati dai web service vengono riconosciuti dall'interprete BPEL. I fault sono considerati nel WSDL come messaggi di output: il pattern asincrono con callback definisce soltanto messaggi di input, quindi i SOAP fault non sono utilizzabili.

Per simulare il lancio di eccezioni si può però usare un meccanismo simile:

1. definire, per ogni fault che può essere generato dal servizio chiamato, una `<operation>` sul `<portType>` contenente l'operazione di callback;
2. utilizzare una `<pick>` al posto della `<receive>` per la ricezione della risposta, inserendo un *event handler* per la callback ed uno per ciascuna `<operation>` associata ad un fault;
3. differenziare il flusso del processo a seconda dell'operazione invocata: è possibile anche lanciare una eccezione non definita nel WSDL da dentro l'*event handler*, in maniera da continuare a sfruttare il meccanismo delle `<catch>`.

Un esempio di utilizzo di questo metodo è riportato in listato 20.

```
<!-- nel WSDL -->
<portType     name="myPortType">
  <operation  name="myOperation">
    <input    message="myInputMessage" />
  </operation>
</portType>
<portType     name="myCallbackPortType">
  <operation  name="myCallbackOperation">
```

```
    <input    message="myInputMessage" />
  </operation>
  <operation  name="anException">
    <input    message="myInputMessage" />
  </operation>
</portType>

<!-- nel BPEL -->
<invoke   partnerLink="myPartnerLink"
          portType="myPortType"
          operation="myOperation"
          inputVariable="myInputVariable">
  <correlation  set="myCorrSet"
                initiate="yes"
                pattern="out" />
</invoke>

<pick>
  <onMessage  partnerLink="myPartnerLink"
              portType="myCallbackPortType"
              operation="anException"
              variable="myOutputVariable">
    <correlation  set="myCorrSet"
                  initiate="no"
                  pattern="in" />
    <throw    faultName="anExceptionFault" />
  </onMessage>
  <onMessage  partnerLink="myPartnerLink"
              portType="myCallbackPortType"
              operation="myCallbackOperation"
              variable="myOutputVariable">
    <correlation  set="myCorrSet"
                  initiate="no"
                  pattern="in" />
    <empty />
  </onMessage>
</pick>
```

Listato 20: invocazione asincrona con callback e gestione degli errori in BPEL

L'utilizzo di questo metodo non è indolore: ovviamente, l'implementazione del web service deve essere in grado di effettuare chiamate ad operazioni diverse a seconda dell'esito delle attività svolte. Questa funzione potrebbe anche essere svolta da un *handler* posto tra il servizio chiamato ed il chiamante: la funzione dell'handler sarebbe quella di intercettare i SOAP fault indirizzati verso l'istanza del processo BPEL e trasformarli nella relativa chiamata all'`<operation>` legata a tale fault.

La disparità di trattamento tra le eccezioni lanciate in modalità sincrona ed asincrona è evidente, da imputarsi in questo caso alle mancanze del WSDL.

2.2.3.3 Integrazione con altri standard W3C

Nel paragrafo precedente è stato analizzato il modello di chiamata asincrona con callback. Questo modello presenta un problema di indirizzamento piuttosto rilevante: il servizio chiamato non ha infatti nessuna informazione su dove contattare nuovamente l'interprete BPEL per invocare l'operazione di callback.

Una soluzione *quick and dirty* potrebbe essere quella di inserire un campo `<Reply-to>` nel formato dei dati scambiati, mischiando il livello applicativo con quello di comunicazione in maniera poco pulita. D'altra parte, SOAP fornisce il meccanismo degli *header* per lo scambio di informazioni di controllo, ed il W3C ha definito lo standard *WS-Addressing*, già citato nel § 2.2.1.2.3 e nel § 2.2.3.2, che si integra nei SOAP header proprio per informazioni quali l'indirizzo per la risposta. Il problema in questo caso è che né il WSDL né BPEL permettono di esprimere una azione o un concetto di tipo "*risolvi l'URL dell'istanza di processo corrente ed aggiungi al messaggio in uscita un header contenente il campo Reply-to in formato WS-Addressing*": soltanto l'interprete BPEL può decidere, in base alla propria implementazione e se lo ritiene necessario, di aggiungere l'intestazione oppure di permettere l'integrazione nel processo di handler o filtri per la gestione degli header.

Oracle BPEL Manager (vedi § 2.2.4.2), ad esempio, riconosce il pattern delle chiamate asincrone con callback e si occupa di inserire un campo appropriato nel messaggio SOAP in uscita.

L'utilizzo di WS-Addressing in maniera selvaggia può portare però a dei conflitti con la definizione del processo BPEL. Come visto nel § 2.2.1.2.3, il BPEL non attribuisce un ID alle istanze di processo, ma le distingue a livello applicativo, utilizzando come chiave primaria un set di valori interni ai messaggi SOAP scambiati. WS-Addressing ha invece un proprio sistema di correlazione, basato su ID univoci. Questi due meccanismi, se usati correttamente, non dovrebbero provocare conflitti ma, nel caso migliore, creano una inutile ridondanza e mischiano le competenze tra il protocollo di trasporto ed il livello di descrizione del processo.

WS-Addressing ha un rapporto molto stretto con un altro standard che nel tempo potrebbe essere integrato in BPEL in maniera più significativa, ovvero *WS-Security* (vedi [WSS]). La sicurezza è un argomento che non viene trattato in questa tesi né nel progetto sviluppato, ma che rimane un problema centrale nella realizzazione di qualsiasi sistema o processo distribuito. A questo proposito, il *Public Key Infrastructure Technical Committee* del gruppo *OASIS* ha pubblicato in [PKI] un *action plan* nel quale vengono evidenziati i principali problemi che ostacolano la realizzazione di una vera infrastruttura di sicurezza interoperabile, tra i quali i costi troppo elevati, il basso supporto fornito dalle varie piattaforme e la concentrazione sugli aspetti tecnologi piuttosto che su quelli applicativi.

Un'altra tecnologia W3C utilizzata nel BPEL è *XPath* (vedi [XPATH]), già citata per l'uso che viene fatto nell'attributo `query` di `<copy>`, `<from>` e `<propertyAlias>`. XPath, nato originariamente per l'uso con XSLT, permette di specificare espressioni molto potenti per accedere a elementi, attributi o valori contenuti in un documento o in un frammento XML: viene quindi anche usata per indirizzare i singoli elementi delle variabili BPEL. Il frammento

XML considerato in questi casi non è però il `<message>` di cui la variabile è istanza, ma una unica `<part>` del messaggio stesso: vengono cioè mischiati due diversi meccanismi di accesso agli elementi, il primo basato sulla scomposizione di messaggi in parti, l'altro più legato ad XML ed alla sua rappresentazione DOM.

Sarebbe auspicabile che nelle prossime versioni questo aspetto venisse chiarito e definito meglio. In effetti, la divisione di un messaggio in parti è motivata dal fatto che le operazioni, in quasi tutti i linguaggi di programmazione, possono accettare più di un parametro in ingresso, ma ha poco senso per quanto riguarda i messaggi di output, poiché tipicamente il valore di *return* è un unico oggetto strutturato, modellato meglio dalla definizione di un tipo XSD piuttosto che dall'unione di più parti. Il problema quindi non si limita all'uso di XPath in BPEL ma coinvolge anche la rappresentazione dei messaggi nel WSDL.

2.2.4 Gli strumenti

2.2.4.1 GlobalBiz Process Engine

GlobalBiz Process Engine è un interprete BPEL sviluppato da una azienda toscana, la *TD Group S.p.A.*, in collaborazione con la quale è stato realizzato, come dimostrazione di utilizzo dell'engine, il progetto *Zlatan* oggetto del § 3.

Si tratta di un prodotto commerciale facente parte della suite *GlobalBiz.* Le sue caratteristiche principali sono:

- *non necessita di un ambiente di esecuzione particolare* (ad esempio, application server di terze parti): il server web è integrato con l'interprete BPEL, caratteristica che lo rende particolarmente performante in termini di prestazioni;
- *non necessita di packaging per i file contenenti i documenti WSDL e BPEL*: la definizione di servizi e processi avviene in maniera diretta, sottomettendo i documenti all'interprete da una interfaccia utente;
- *fornisce una rappresentazione grafica in tempo reale dello stato del processo*: si può scegliere tra una rappresentazione in formato JPG (generata utilizzando *GraphWiz*) o una in formato SVG (vedi figura 13 per un esempio);
- *rispetta le specifiche e la filosofia di fondo di BPEL*: non sono state introdotte nel linguaggio modifiche pesanti come file di configurazione o definizione di nuovi elementi, la personalizzazione è minima e mirata alla soluzione dei limiti del BPEL e del WSDL, alcuni dei quali citati nei paragrafi precedenti. Particolare attenzione è stata posta nel rispetto dell'utilizzo dei costrutti `<correlation>` e `<scope>`;
- *comprende un adapter* per esporre come web service procedure batch DOS, processi Windows e shell Unix.

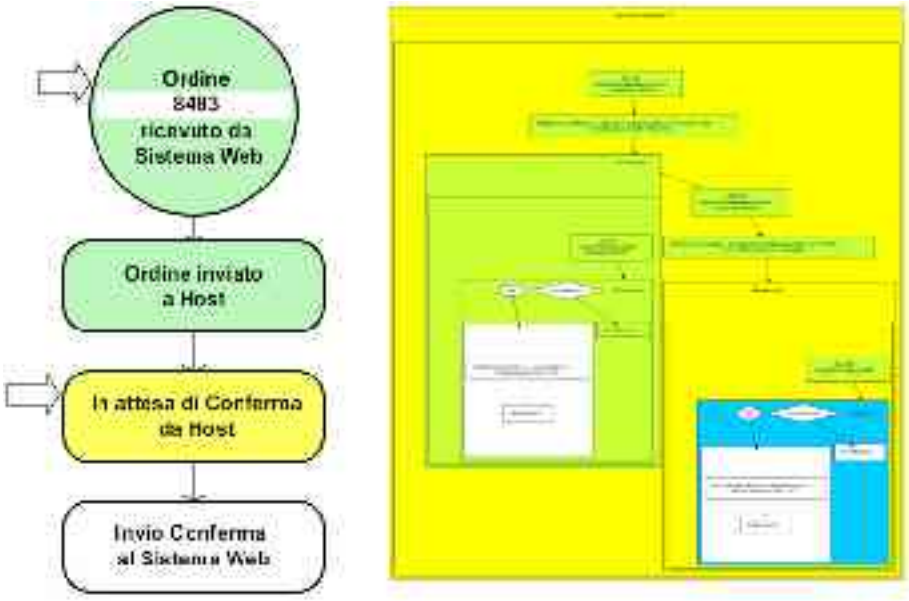

Figura 13: esempi di SVG e di JPEG generati dal PE per lo stesso processo

Insieme al PE è disponibile in GlobalBiz *ECO* (*Enterprise Context Organizer*): ECO è un *EII* (*Enterprise Information Integration*), ovvero una applicazione in grado di integrare dati provenienti da fonti diverse. ECO permette di definire formule, dizionari di traduzione, relazioni e trasformazioni tra tabelle di database diversi, in maniera da offrire una vista logica omogenea dei dati distribuiti. ECO utilizza XML per rappresentare gli schemi logici e fisici dei dati e *MathML* per la definizione delle formule.

Un'altra applicazione presente nella suite è il demone che si occupa della gestione degli eventi generati dai vari attori del sistema (servizi, clienti ed applicazioni della suite). Le applicazioni possono registrare un evento presso una base di dati condivisa: il demone accede al database e trasforma gli eventi registrati in messaggi SOAP per l'attivazione dei processi BPEL o per l'invocazione di servizi web.

L'utilizzo di GlobalBiz permette quindi di effettuare l'integrazione di software aziendale sia a livello di *applicazioni*, attraverso il Process Engine, sia a livello di *dati*, attraverso ECO.

2.2.4.2 Oracle BPEL Process Manager

Oracle BPEL Process Manager (vedi [ORABPEL]) è un plug-in per la piattaforma *Eclipse* (vedi [ECL]) che permette di scrivere ed eseguire processi BPEL. Il tool è al momento utilizzabile liberamente: i suoi punti di forza risiedono nell'interfaccia utente e nella presenza di un *designer* di processi BPEL in cui è stato particolarmente curato l'aspetto grafico. Il linguaggio BPEL è stato esteso per supportare l'attivazione diretta di un processo da una JSP e per interagire con J2EE; viene inoltre usato *WS-Addressing* per la correlazione e l'indirizzamento dei messaggi.

Gli svantaggi di questo software sono la necessità di avere un application server Java a disposizione, l'eccessiva personalizzazione delle specifiche BPEL che rendono il codice generato poco portabile, il supporto limitato ai correlation set e la mancanza di un tool per la visualizzazione grafica in tempo reale dello stato del processo.

2.2.4.3 ActiveBPEL

ActiveBPEL (vedi [ACTBPEL]) è un motore BPEL sviluppato con un progetto *open-source* sponsorizzato da *Active Endpoints*. L'azienda commercializza un ambiente per il design ed il deploy dei processi compatibile con il motore. Tra le caratteristiche più interessanti di ActiveBPEL vi sono, oltre alla disponibilità open-source, il supporto nativo alla creazione di handler e la presenza di una API per generare ed eseguire processi BPEL all'interno delle proprie applicazioni. ActiveBPEL non è semplice da usare: il packaging dei file da fornire all'interprete è complesso e deve essere fatto a mano ed il motore deve essere installato su una piattaforma Java comprensiva di Tomcat ed Axis (al momento della stesura di questo documento, il supporto per gli altri application server è minimo).

2.3 SAP Java Connector

SAP Java Connector (*JCo*) è la tecnologia sviluppata da SAP per permettere l'interazione bidirezionale tra programmi Java ed il sistema R/3. La distribuzione di JCo è composta da un file `jar` contenente le classi Java e da alcune `dll` con il codice per interfacciarsi (anche da remoto) al sistema R/3: il collegamento tra le classi Java e le librerie è gestito utilizzando *Java Native Interface*.

Questo paragrafo introduce le principali tecnologie di connessione tra SAP ed il mondo esterno, con particolare riferimento a quelle Java. Verrà poi analizzato il contenuto del package `com.sap.mw.jco` e l'uso delle principali classi, quindi si passerà ad elencare le applicazioni realizzabili e gli sviluppi futuri delle tecnologie di connessione SAP.

Per maggiori informazioni su SAP JCo si rimanda a [ARAJCO].

2.3.1 Il modello

I tipi di comunicazione possibili da e verso un sistema SAP sono due:

- *comunicazione omogenea*: è quella che avviene tra due sistemi SAP, utilizzando tecnologie proprietarie consolidate negli anni;
- *comunicazione eterogenea*: è quella che avviene tra un sistema SAP ed uno non-SAP. Si basa su tecnologie miste, di proprietà SAP o sviluppate da terze parti.

Le prime tecnologie implementate in ordine cronologico sono state ovviamente quelle che permettono la comunicazione omogenea tra due sistemi SAP: tra queste, *RFC*, *BAPI* e *IDoc*. In seguito, con la nascita di Java e della piattaforma .NET e visto il crescente interesse del mercato nei confronti delle tematiche dell'integrazione di applicazioni di classe *enterprise*, sono nati connettori per la comunicazione eterogenea sempre più sofisticati e potenti.

RFC (*Remote Function Call*) è una tecnologia che realizza nella piattaforma ABAP il ben noto concetto di chiamata di procedura remota (*Remote Procedure Call*). Le RFC sono tuttora utilizzate per la programmazione ABAP e si sono sviluppate nel tempo, aggiungendo nuove modalità di chiamata (*transazionale*, *a coda* e *veloce*) all'unica modalità sincrona disponibile originariamente. BAPI (*Business API*) è una libreria di funzioni ABAP che sfrutta le RFC e che ha una impostazione ad oggetti per rendere semplice ed automatica la scrittura di programmi.

Le principali tecnologie per la comunicazione eterogenea sono:

- *SAP BC* (*Business Connector*): si tratta di una tecnologia che permette l'integrazione con diversi tipi di sistemi. Utilizza HTTP come protocollo di trasporto ed XML per la rappresentazione dei dati scambiati;
- *SAP JCo* (*Java Connector*): come accennato, è la prima versione delle librerie per connettere Java e SAP. Di fatto è un package Java che, utilizzando *JNI*, rende disponibili nell'ambiente Java le funzioni contenute nelle BAPI.
- *SAP JRA* (*Java Resource Adapter*) è una estensione delle JCo che rende disponibili gli oggetti SAP come *risorse J2EE*, secondo quanto richiesto dalle specifiche Sun ([JCA]);

- *SAP .NET Connector*: una serie di tool per connettere SAP con la piattaforma Microsoft .NET. Le funzionalità sono più vicine a quelle di WAS che non a JCo o a JRA;
- *SAP WAS* (*Web Application Server*) è un vero e proprio application server che può gestire moduli J2EE di qualsiasi tipo, integrandoli in SAP grazie a JCo ed alle altre tecnologie. Tra le altre funzionalità, permette l'esposizione di procedure BAPI come web service.

Le tecnologie di connessione a Java possono essere rappresentate con una pila, in quanto a tutti gli effetti ogni nuova tecnologia si appoggia su quella precedente e ne espande le funzionalità e la connettività, ma in realtà l'interfaccia con il sistema R/3 è sempre la stessa, basata su BAPI ed RFC (vedi fig. 14).

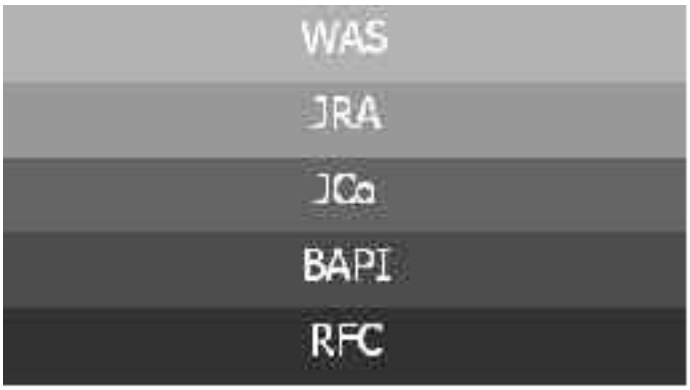

Figura 14: pila delle interfacce SAP-Java

2.3.2 Il package com.sap.mw.jco

Le funzioni principali della libreria sono offerte in realtà da un'unica classe, `com.sap.mw.jco.JCO`, che contiene al suo interno diverse *inner class* pubbliche. In questo paragrafo verrà descritto l'uso del package per l'applicazione più immediata, ovvero la scrittura di un client Java che effettua delle operazioni su R/3. Nel § 2.3.3.4 si evidenzieranno le differenze nel caso in cui si voglia invece scrivere un server Java contattabile da R/3.

La classe `JCO.Connection` rappresenta una connessione ad un sistema SAP. Tale classe è astratta: le sue specializzazioni istanziabili sono `JCO.Client` e `JCO.Server`. Per ottenere un client, si possono usare due metodi statici di `JCO`: `JCO.createClient()` o `JCO.getClient()`, il cui uso è discriminato dalla presenza di una connessione diretta o di un pool di connessioni (vedi § 2.3.3.1 e § 2.3.3.2).

Una volta ottenuta una connessione sotto forma di client, è necessario conoscere i *metadati* della funzione BAPI da invocare. Senza entrare nei dettagli sul significato dei metadati, viene riportato nel listato 21 il codice, piuttosto standard, per ottenere il riferimento ad una funzione (nell'esempio, la `BAPI_COMPANYCODE_GETLIST`, che riporta l'elenco dei codici ed i nomi delle aziende registrate nel sistema R/3):

```
JCO.Client connection = JCO.createClient(
                "800",      // numero del client del sistema
                "username", // username
                "password", // password
                "EN",       // lingua
                "host",     // nome del server SAP
                "00");      // system number
connection.connect();       // effettua la connessione

// recupera una funzione dal repository
JCO.Repository     r = new JCO.Repository("MyRep", connection);
IFunctionTemplate ft =
  r.getFunctionTemplate ("BAPI_COMPANYCODE_GETLIST");
JCO.Function   funct = new JCO.Function(ft);
```

Listato 21: creazione di una funzione in SAP JCo

L'esecuzione di una funzione avviene per mezzo della chiamata al metodo `execute()`: curiosamente, nonostante sia possibile collegare un oggetto funzione alla connessione che l'ha indirettamente creato, tale metodo non fa parte della classe `JCO.Function` ma di `JCO.Client`.

Le BAPI non prevedono soltanto parametri di input e di output, ma tre diversi tipi:

- *parametri import*: parametri di solo input, possono essere composti da *campi*, che contengono semplici informazioni scalari, oppure da *strutture*, che sono collezioni di campi ordinati ed accessibili attraverso una chiave (sono equivalenti alle *mappe* nella terminologia Java);
- *parametri export*: parametri di solo output, hanno un formato simile a quello dei parametri di import;

- *parametri table*: parametri sia di input che di output. Hanno la forma di una tabella di un database, ovvero sono ordinati in righe e colonne: le colonne sono accessibili attraverso l'uso del nome del campo, mentre le righe sono accessibili in ordine sequenziale. I parametri tabella non sono in alcuna maniera collegati alle tabelle del database di SAP, ma servono soltanto per il passaggio di vettori di elementi strutturati tra il chiamante e la funzione chiamata.

Prima di chiamare una funzione, è possibile settarne i parametri import e table utilizzando i metodi getter e setter `getImportParameterList()`, `getTableParameterList()`, `setImportParameterList()` e `setTableParameterList()`. Ci sono due usi equivalenti di tali metodi: si possono usare i metodi getter per recuperare il riferimento all'oggetto `JCO.ParameterList` da settare in seguito oppure si può prima creare l'oggetto e poi settarlo come parametro per una funzione. Questo secondo metodo è più complesso: nel listato 22 viene riportato un semplice esempio riguardante il primo (l'esempio è relativo ad un'altra funzione, `BAPI_COMPANYCODE_GETDETAIL`, perché `BAPI_COMPANYCODE_GETLIST` non ha parametri import).

```
JCO.ParameterList importPars = funct.getImportParameterList();
importPars.setValue("COMP_CODE", "0001");
connection.execute(funct);
```

Listato 22: parametri di import ed esecuzione di una funzione JCo

Analogamente a quanto accade con i parametri import, si può accedere ai parametri export ritornati dalla BAPI. Nell'esempio del listato 23, si accede alla struttura predefinita `RETURN` ritornata da tutte le BAPI:

```
connection.execute(funct);
JCO.ParameterList exportPars = funct.getExportParameterList();
JCO.Structure     returnPars = export.getStructure("RETURN");
```

Listato 23: parametri di export e struttura RETURN in JCo

`RETURN` contiene un messaggio di testo ed un carattere che indica la tipologia del messaggio (`E` per `Error`, `W` per `Warning` e `S` per `Success`). Sia il messaggio che il tipo sono campi della struttura, accessibili attraverso i nomi di campo `TYPE` e `MESSAGE` (listato 24).

```
JCO.Field message = return.getField("MESSAGE");
JCO.Field    type = return.getField("TYPE");
if (type.getString().equals("E"))
  throw new Exception("SAP Exception");
if (type.getString().equals("W"))
  System.out.print("WARNING: ");
System.out.println(message.getString());
```

Listato 24: accesso ai campi di una struttura JCo

È possibile anche accedere direttamente al valore dei campi dalla struttura, come dall'esempio 25 equivalente a quello precedente.

```
String message = returnPars.getString("MESSAGE");
String    type = returnPars.getString("TYPE");
if (type.equals("E"))
  throw new Exception("SAP Exception");
if (type.equals("W"))
  System.out.print("WARNING: ");
System.out.println(message);
```

Listato 25: accesso diretto ai campi di una struttura JCo

La gestione delle tabelle è molto intuitiva e simile a quanto avviene con JDBC: si usano tra gli altri i metodi `nextRow()`, `getFirstRow()` e `getLastRow()` per navigare tra le righe ed i metodi `getXXX(int index)` e `getXXX(String colName)` per accedere ai record (`XXX` è il tipo dei dati contenuti nel record).

Nel listato 26 è riportato un esempio di accesso ad una tabella i cui valori sono ritornati dalla funzione BAPI `BAPI_COMPANYCODE_GETLIST`.

```
JCO.ParameterList pl = funct.getTableParameterList();
JCO.Table      codes = pl.getTable("COMPANYCODE_LIST");

if (codes.getNumRows() > 0) {
  codes.firstRow();
  do {
     System.out.println(codes.getString("COMP_CODE") +
       "\t" + codes.getString("COMP_NAME"));
  } while(codes.nextRow());
}
```

Listato 26: accesso ai campi di una tabella JCo

2.3.3 Le applicazioni

2.3.3.1 Client con connessione diretta

Le applicazioni client con connessione diretta sono le più semplici che si possano scrivere con SAP JCo. Una connessione diretta usa una coppia `(userid, password)` per connettersi a SAP: la gestione dell'apertura e della chiusura della connessione viene lasciata al programmatore.

Le connessioni dirette non sono ottimizzate per l'uso con thread o istanze multiple e possono in questi casi generare conflitti: sono perciò utilizzate solitamente per applicazioni *desktop* o *stand-alone*.

Nel listato 27 è riportato per intero l'esempio del § 2.3.2, che utilizza una connessione diretta.

```
import com.sap.mw.jco.*;
public class CompanyCodeGetList {

  public static void main (String args[])
  throws Exception {
    JCO.Client connection = JCO.createClient(
      "800", "username", "password", "EN",  "host", "00");
    connection.connect();
    JCO.Repository     r =
      new JCO.Repository("MyRep", connection);
    IFunctionTemplate ft =
      r.getFunctionTemplate("BAPI_COMPANYCODE_GETLIST");
    JCO.Function   funct = new JCO.Function(ft);
    connection.execute(funct);
    JCO.ParameterList  exportPars =
      funct.getExportParameterList();
    JCO.Structure returnPars  =
      exportPars.getStructure("RETURN");
    String message   = returnPars.getString("MESSAGE");
    String type      = returnPars.getString("TYPE");
    if (type.equals("E")) throw new Exception("SAPException");
    if (type.equals("W")) System.out.print("WARNING: ");
    System.out.println(message);
    JCO.ParameterList pl = funct.getTableParameterList();
    JCO.Table      codes = pl.getTable("COMPANYCODE_LIST");
    if (codes.getNumRows() > 0) {
      codes.firstRow();
      do {
        System.out.println(codes.getString("COMP_CODE") +
          "\t" + codes.getString("COMP_NAME"));
      } while(codes.nextRow());
    }
    connection.disconnect();
  }
}
```

Listato 27: esempio di utilizzo di un client JCo con connessione diretta

2.3.3.2 Connection pooling

Le connessioni dirette non sono adatte, come detto, ad applicazioni che utilizzano thread o istanze multiple. In questa classe di applicazioni rientrano anche i web service, il cui pool di istanze è gestito in maniera trasparente al programmatore dall'ambiente di esecuzione.

JCo mette a disposizione per questi casi un meccanismo di *connection pooling*. Il pool di connessioni, identificato da un nome, viene istanziato una sola volta per ogni JVM: quando un thread ha bisogno di una connessione, ne ottiene una tra quelle disponibili e la rilascia prima possibile. Tutte le connessioni del pool condividono la stessa coppia `(username, password)`.

Un esempio di applicazione che utilizza il connection pooling è riportato nel listato 28, con il relativo output nel listato successivo. Il metodo `main()` della classe genera un certo numero di thread, ognuno dei quali acquisisce informazioni su un singolo `CompanyCode` scelto a caso.

```
import com.sap.mw.jco.*;

public class CompanyCodeConnPool extends Thread {

  public  static final String POOL_NAME = "MyPool";
  public  static final String REP_NAME  = POOL_NAME + "Rep";
  private int index;

  public CompanyCodeConnPool(int index) { this.index = index; }

  public void run() {
    try {
      JCO.Client connection = JCO.getClient(POOL_NAME);
      JCO.Repository r =
        new JCO.Repository(REP_NAME, connection);
      IFunctionTemplate ft1 =
        r.getFunctionTemplate("BAPI_COMPANYCODE_GETLIST");
      IFunctionTemplate ft2 =
        r.getFunctionTemplate("BAPI_COMPANYCODE_GETDETAIL");
      JCO.Function   listFunction = new JCO.Function(ft1);
      JCO.Function detailFunction = new JCO.Function(ft2);
      connection.execute(listFunction);
      JCO.Table codes = listFunction.getTableParameterList().
        getTable("COMPANYCODE_LIST");
      if ((codes.getNumRows() > 0) &&
        (index < codes.getNumRows())) {
        codes.firstRow();
        int i = 0;
        while ((i < this.index) && (!codes.isLastRow())) {
          codes.nextRow(); i++;
        }
        detailFunction.getImportParameterList().
          setValue(codes.getString("COMP_CODE"),
          "COMPANYCODEID");
        detailFunction.getExportParameterList().
```

```
            setActive(false, "COMPANYCODE_ADDRESS");
          connection.execute(detailFunction);
          String code    = codes.getString("COMP_CODE");
          String name    = codes.getString("COMP_NAME");
          String country =
            detailFunction.getExportParameterList().
            getStructure("COMPANYCODE_DETAIL").
            getString("COUNTRY");
          System.out.println(this.index  + "\t" + code +
            "\t" + name + "\t" + country);
        } else
          System.out.println(this.index + "\tNon Presente");
        JCO.releaseClient(connection);
      } catch (Exception e) { e.printStackTrace(); }
    }

    public static void main(String args[]) {
      JCO.addClientPool(POOL_NAME, 5,
        "800", "username", "password", "EN", "host", "00");
      for (int i = 0; i < 10; i++) {
        int index = (int) (Math.random() * 50);
        new CompanyCodeConnPool(index).start();
      }
    }
}
```

Listato 28: esempio di utilizzo di un client JCo con pool di connessioni

```
27      8000    IDES Chile              CL
38      R300    IDES Retail INC US      US
49      Non Presente
1       0100    IDES Japan 0100         JP
13      4000    IDES                    US
17      5200    IDES Japan 5200         JP
13      4000    IDES                    US
35      F100    Bankhaus Frankfurt      DE
9       2500    IDES Netherlands        NL
28      AC00    IDES Training AC Gr. 00 DE
```

Listato 29: possibile output generato da CompanyCodeConnPool

2.3.3.3 Classi proxy

Il modello di programmazione delle BAPI è abbastanza semplice; schematizzando al massimo, si tratta di:

1. creare una connessione;
2. ottenere il "riferimento" a una o più funzioni;
3. settare i parametri import e table per l'ingresso;
4. eseguire la funzione;
5. valutare i parametri export e table ritornati;
6. chiudere la connessione o effettuare altre chiamate di funzione.

La vera difficoltà di JCo risiede invece nella necessità di dover conoscere i nomi delle BAPI, le tabelle utilizzate e i nomi e i formati delle strutture e dei campi. In questo può essere d'aiuto l'indice delle interfacce SAP pubblicato in [IFR].

Per quanto riguarda operazioni di routine che sono tra le più classiche di un sistema gestionale come SAP R/3, quali ad esempio:

- consultare un listino;
- inserire o validare un ordine d'acquisto;
- emettere una fattura;
- visualizzare la lista dei clienti;

è sicuramente più utile creare delle classi Java che facciano da proxy alle JCo e nascondano una volta per tutte la complessità della tecnologia. Alcune aziende operanti nel campo delle EAI forniscono collezioni di classi proxy o tool automatici per la creazione di proxy *ad hoc* in base alle proprie esigenze (vedi [ARA]).

Alla luce di quanto detto, la figura 14 può essere modificata nella seguente maniera:

Figura 15: pila delle interfacce SAP-Java modificata

Nel listato 30 è riportato un esempio di classe proxy, in cui viene definito un oggetto `CompanyCode` a cui è possibile applicare due metodi il cui codice è già stato visto in precedenza, `getList()` e `getDetail()`, usando le relative BAPI. La classe utilizza un semplice bean `Company`, il cui codice non è riportato, per contenere i valori ritornati dai metodi.

```
import com.sap.mw.jco.*;

public class CompanyCode {

  private JCO.Client      connection;
  private JCO.Repository  repository;
  private JCO.Function    getDetailFunction;

  private void init() {
    this.repository =
      new JCO.Repository("CCProxy", connection);
    IFunctionTemplate ft =
      repository.getFunctionTemplate(
      "BAPI_COMPANYCODE_GETDETAIL");
    this.getDetailFunction  = new JCO.Function(ft);
  }

  public void setCompanyCodeID(String companyCodeID) {
    getDetailFunction.getImportParameterList().
      setValue(companyCodeID, "COMPANYCODEID");
  }

  public Company getDetail() {
    getDetailFunction.getExportParameterList().
      setActive(false, "COMPANYCODE_ADDRESS");
    connection.execute(getDetailFunction);
    JCO.Structure detail = getDetailFunction.
      getExportParameterList().getStructure(
      "COMPANYCODE_DETAIL");
    Company ret = new Company();
    ret.setCity(detail.getString("CITY"));
    ret.setCode(detail.getString("COMP_CODE"));
    ret.setCountry(detail.getString("COUNTRY"));
    ret.setName(detail.getString("COMP_NAME"));
    return ret;
  }

  public static Company[] getList(JCO.Client connection) {
    JCO.Repository repository =
      new JCO.Repository("CCSProxy", connection);
    IFunctionTemplate ft =
      repository.getFunctionTemplate(
      "BAPI_COMPANYCODE_GETLIST");
    JCO.Function getListFunction  = new JCO.Function(ft);
    connection.execute(getListFunction);
    JCO.Table codes = getListFunction.getTableParameterList().
```

```
      getTable("COMPANYCODE_LIST");
    int nr = codes.getNumRows();
    if (nr > 0) {
      Company[] ret = new Company[nr];
      codes.firstRow();  int cont = 0;
      do {
        ret[cont] = new Company();
        ret[cont].setCode(codes.getString("COMP_CODE"));
        ret[cont].setName(codes.getString("COMP_NAME"));
        cont++;
      } while (codes.nextRow());
      return ret;
    }
    return null;
  }

  public CompanyCode(JCO.Client connection) {
    this.connection = connection;  this.init();
  }

  public CompanyCode(JCO.Client connection,
    String companyCodeID) {
    this.connection = connection;  this.init();
    this.setCompanyCodeID(companyCodeID);
  }

  public static void main(String[] args) {
    JCO.Client connection = JCO.createClient(
      "800", "username", "password", "EN", "host", "00");
    connection.connect();
    Company[] cclist = CompanyCode.getList(connection);
    for (int i = 0; i < cclist.length; i++) {
      CompanyCode cc =
        new CompanyCode(connection, cclist[i].getCode());
      Company temp = cc.getDetail();
      System.out.println(temp.getCode() + "\t" +
        temp.getName() + "\t" + temp.getCountry());
      }
    connection.disconnect();
  }
}
```

Listato 30: esempio di classe proxy JCo per CompanyCode

```
0001    SAP A.G.          DE
0100    IDES Japan 0100   JP
0110    IDES Japan 0110   JP
1000    IDES AG           DE
2000    IDES UK           GB
2100    IDES Portugal     PT
2200    IDES France       FR
2300    IDES España       ES
```

Listato 31: parte dell'output generato dal metodo main() di CompanyCode

2.3.3.4 Applicazioni server

Negli esempi visti fino ad adesso, le classi di JCo sono state sempre usate per scrivere un client che compie una qualche operazione su un server SAP R/3 remoto, utilizzando le BAPI. Questa è sicuramente l'applicazione più diffusa di un connettore come JCo, ma non è l'unica.

Nel package infatti è presente anche una classe – `JCO.Server` – che permette di scrivere un vero e proprio server accessibile da un programma ABAP eseguito in ambiente SAP. Il codice fornito da SAP in JCo gestisce una gamma di problematiche proprie dei server (caduta della connessione, *mapping* delle eccezioni Java in eccezioni ABAP, passaggio dei parametri, ecc...), in maniera da lasciare al programmatore soltanto la definizione degli aspetti applicativi.

Come accennato, un server deve estendere la classe `JCO.Server`. Il codice del server risiede quasi tutto nel metodo `protected void handleRequest()`. Tale metodo accetta come parametro la `JCO.Function` invocata da lato client (SAP); il codice applicativo da eseguire può essere distinto in base al nome della funzione, come dallo schema di listato 32.

Lo sviluppo di applicazioni server JCo comprende la configurazione di R/3 e la scrittura dei client ABAP.

Una applicazione interessante dei server può essere la scrittura di un proxy che permetta alle routine ABAP di accedere ad un sistema EIS esterno utilizzando le BAPI, mappate in realtà sulle chiamate di procedura del sistema esterno tramite l'utilizzo del proxy. Un esempio di questa architettura è riportata in figura 16. Per maggiori informazioni sulle applicazioni server JCo, si rimanda a [ADVJCO].

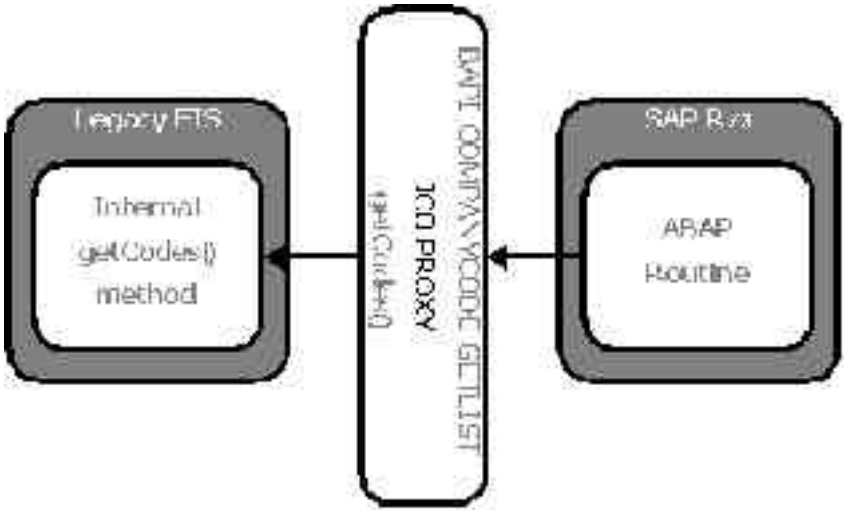

Figura 16: server JCo come proxy tra R/3 ed un EIS legacy

```
protected void handleRequest(JCO.Function function) {

  String fName = function.getName();
  JCO.Structure returnCode =
    function.getExportParameterList().getStructure("RETURN");

  if ("FUNZIONE1".equals(fName)) {
```

```
    // codice FUNZIONE1
    returnCode.setValue("S", "TYPE");
    returnCode.setValue("Info Message 1.", "MESSAGE");
  } else if ("FUNZIONE2".equals(fName)) {
    // codice FUNZIONE2
    returnCode.setValue("S", "TYPE");
    returnCode.setValue("Info Message 2.", "MESSAGE");
  } else {
    // funzione non supportata
    throw new JCO.AbapException(" ... ", " ... ");
  }
}
```

Listato 32: schema del metodo hadleRequest() in un server JCo

2.3.4 Sviluppi futuri

Le figure 14 e 15 illustrano in maniera chiara che l'evoluzione delle interfacce eterogenee da e verso SAP si basa su una filosofia di riuso del codice di più basso livello già sviluppato. L'ingresso dell'architettura *service-oriented* e dei servizi web nel mercato dell'EAI potrebbe provocare nel tempo la fine dell'accrescimento delle pile di tecnologie delle figure: l'interfaccia a servizi potrebbe essere infatti la soluzione definitiva al problema dell'interazione con le altre piattaforme.

Le caratteristiche che la rendono particolarmente adatta allo scopo sono:

- *la natura multi-piattaforma*: SAP non dovrebbe più sviluppare connettori diversi per ambienti Java e .NET, ma un'unica libreria di servizi fornita al più di diversi set di *stub platform-dependant* per facilitare ulteriormente la scrittura dei client;
- *l'integrazione con le tecnologie precedenti*: i web service si possono inserire perfettamente, come visto, in cima alla pila di tecnologie della figura 14;
- *l'uso di standard condivisi*: SAP è membro della maggior parte dei comitati che propongono l'adozione di standard XML nello sviluppo delle applicazioni EAI e non può quindi che trarre vantaggio nell'utilizzare essa stessa tali tecnologie.

3 Il progetto

3.1 Analisi funzionale

L'obiettivo del progetto *Zlatan* è di creare un sistema che permetta l'inserimento di un ordine di acquisto, inviato utilizzando protocolli di trasmissione diversi, in un sistema SAP R/3. Il sistema deve essere inoltre in grado di fornire una o più risposte al cliente che ha inviato l'ordine, utilizzando anche in uscita una serie di protocolli diversi.

In figura 17 è riportata una schematizzazione del sistema dal punto di vista funzionale.

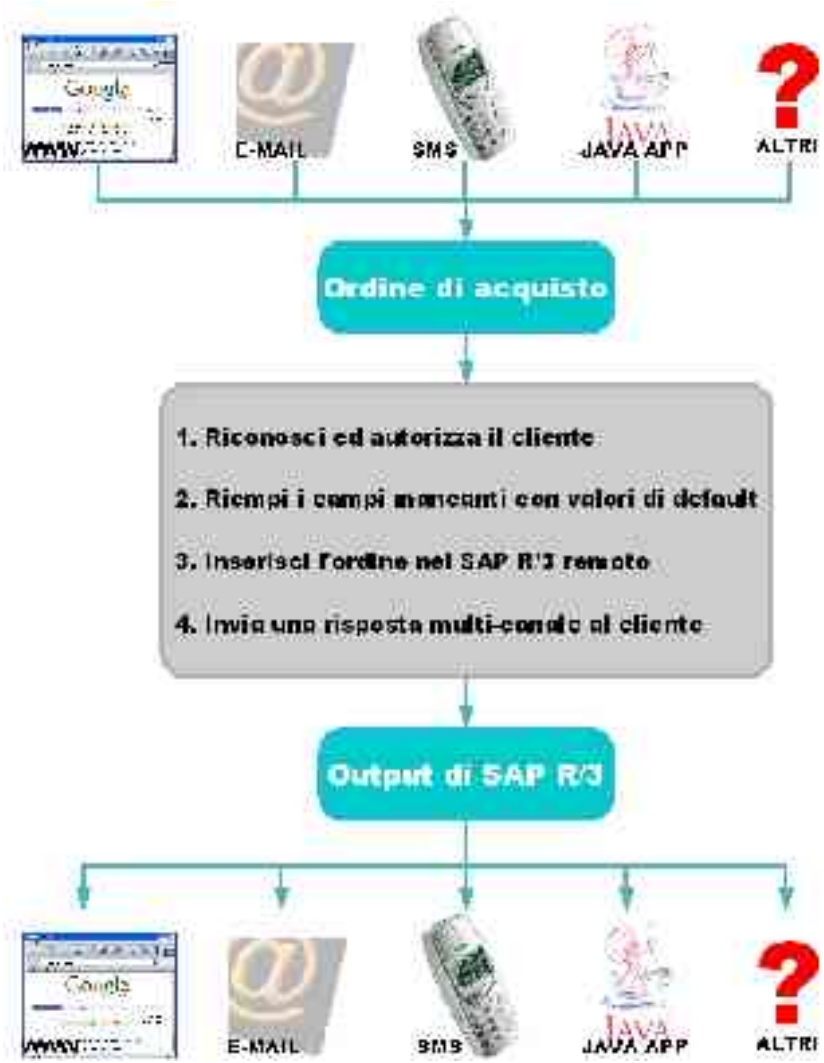

Figura 17: schema funzionale del sistema Zlatan

L'implementazione presentata in questo documento supporta in ingresso i protocolli HTTP, SMTP/POP3, RMI e SMS ed in uscita i protocolli HTTP, SMTP/POP3 e SOAP.

Lo scenario di applicazione di Zlatan può essere sia il *B2C*, ad esempio per l'interazione diretta con un cliente finale in una applicazione di tipo *CRM*, sia il *B2B*, nel caso in cui si voglia far comunicare due *EIS* di aziende diverse.

3.2 Analisi non funzionale

Le caratteristiche non funzionali del sistema sono:

- *espansibilità e modularità*: il sistema deve essere costituito da moduli relativamente semplici, che interagiscano l'uno con l'altro attraverso protocolli diversi in maniera da costituire un *sistema nervoso* di base che possa essere espanso in futuro, in seguito alla richiesta di introduzione di servizi aggiuntivi (es. nuovi protocolli in ingresso o in uscita, gestione della firma digitale);
- *replicabilità e scalabilità*: il sistema deve essere replicabile, ovvero ogni modulo deve poter essere istanziato più di una volta senza comprometterne il funzionamento, e deve contemporaneamente essere scalabile, ovvero all'aumento del numero delle richieste deve poter essere attuabile un proporzionale aumento della capacità del sistema;
- *robustezza, tolleranza agli errori e sicurezza*: il sistema deve essere robusto e tollerante agli errori, oltre a non permettere intrusioni da parte di persone non autorizzate. Questi requisiti sono stati implementati soltanto in parte;
- *prestazioni*: il sistema, pur nel rispetto dell'architettura progettata, deve sfruttare le risorse al massimo e minimizzare l'overhead ed il sovraccarico.

Il concetto di espansibilità e modularità sta alla base dell'architettura stessa del sistema. Come spiegato più avanti, l'architettura è complessa e per certi versi ridondante: questa è una scelta implementativa precisa che ha tra i suoi vantaggi la semplicità nella realizzazione dei vari driver che gestiscono i protocolli di ingresso.

L'utilizzo di tecnologie votate all'architettura SOA e la stessa struttura modulare sono gli elementi che consentono di replicare e scalare il sistema in maniera molto semplice. Il § 3.3.1 illustra come i moduli possano essere replicati o distribuiti su più server.

La robustezza, la sicurezza e le prestazioni, per quanto implementate soltanto in parte a causa della natura dimostrativa e non commerciale del sistema, sono assicurate da una struttura di base che filtra ripetutamente le richieste, evitando per quanto possibile di far elaborare al server SAP richieste incomplete, errate o provenienti da utenti non autorizzati. In altre parole, il carico dell'*elaborazione della richiesta* e dei *controlli sintattici e di merito* è distribuito tra i vari moduli: questo permette di evitare il sovraccarico del server su cui risiede il sistema R/3 e di garantire una certa robustezza, assicurata anche da una precisa implementazione della gestione degli errori.

3.3 Architettura

L'architettura generale del sistema è riportata in figura 18.

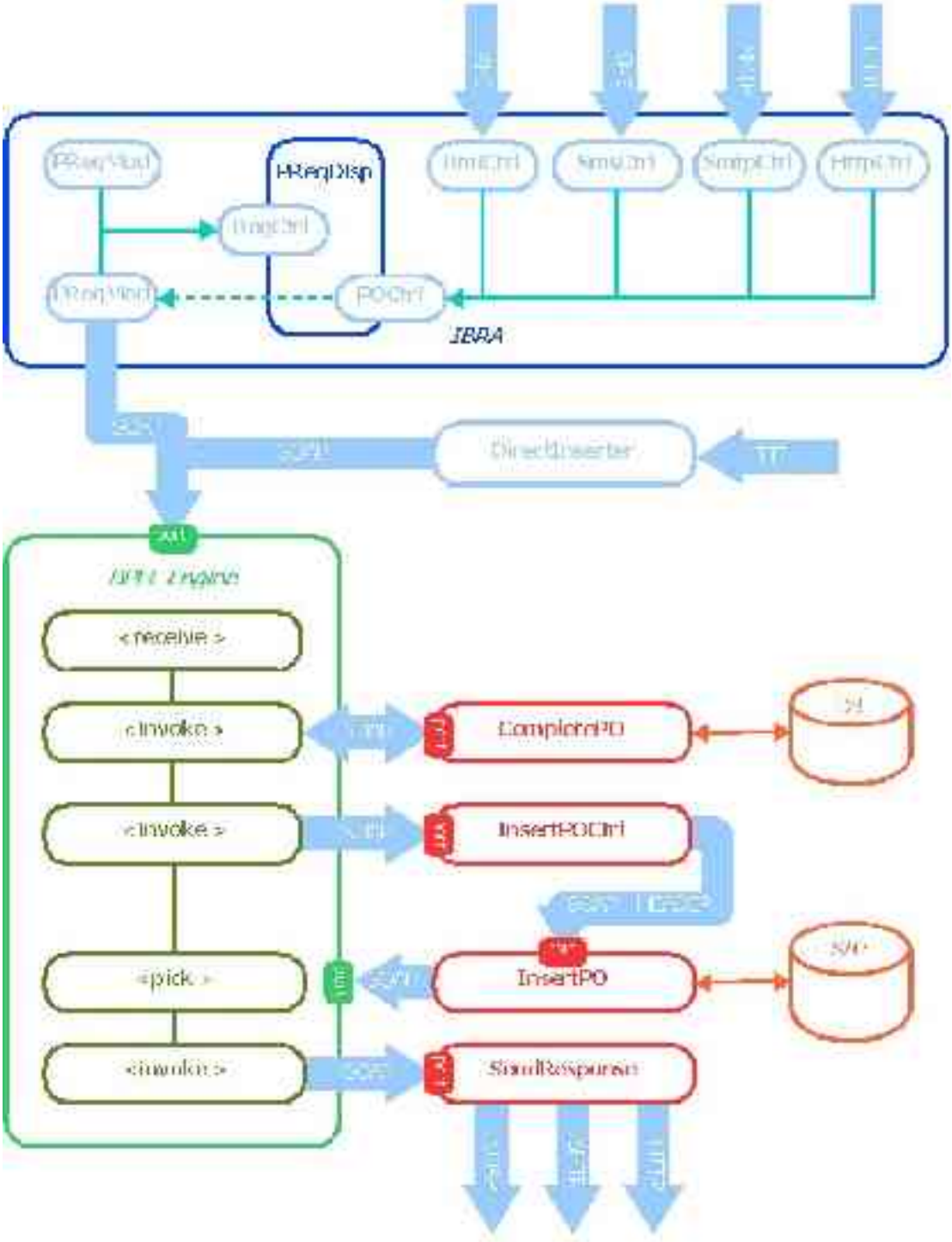

Figura 18: architettura generale del sistema

Nello schema si identificano chiaramente tre "blocchi" distinti: il primo è il modulo *IBRA*, un insieme *pure Java* di processi che gestiscono l'ingresso multicanale.

IBRA, analizzato molto in dettaglio nel § 3.4.1, è composto da:

- *una serie di driver*, ognuno dedicato alla ricezione ed al parsing dei messaggi in uno specifico protocollo. Il compito dei driver è di trasformare il messaggio dal formato proprio del protocollo di trasporto ad un formato intermedio JMS;
- *un server distribuito*, implementato utilizzando JMS con il pattern descritto nel § 2.1.4.1 e composto quindi da un dispatcher e da diverse istanze del modulo di elaborazione che si occupa di attivare il processo BPEL.

Il secondo blocco dello schema è costituito dal codice BPEL che descrive il processo: il codice è interpretato ed eseguito dal *Process Engine* di *GlobalBiz* presentato nel § 2.2.4.1. All'analisi del processo è dedicato il § 3.4.3.

L'ultimo blocco è composto dai web service invocati durante il processo dall'interprete. I web service sono stati implementati utilizzando *Apache Axis* (vedi Appendice A) e sono analizzati a partire dal § 3.4.2.1.

3.3.1 Distribuzione dei moduli del sistema

Zlatan è un sistema altamente distribuibile tra macchine diverse. Questa caratteristica, unita alle proprietà di replicabilità e scalabilità, rendono possibile una vasta gamma di configurazioni diverse.

Il caso più semplice è quello in cui IBRA risiede su una macchina, il PE su un'altra ed i web service su una terza. IBRA può essere replicato in blocco oppure scomposto, configurato, distribuito e replicato: può essere dedicata, ad esempio, una macchina al dispatcher e alle istanze dei moduli per l'elaborazione, una macchina a più repliche di uno stesso driver ed una macchina per i restanti driver.

Anche il PE può essere replicato, a patto che venga creata un'altra istanza di IBRA per l'attivazione della nuova copia del processo o che vengano usati dei dispatcher HTTP per l'inoltro delle richieste. La gestione del pool di istanze dei servizi web è invece demandata all'application server all'interno del quale vengono eseguiti.

3.3.2 Formato dei dati

Una delle caratteristiche più interessanti dell'intero sistema Zlatan è che il formato dei dati dell'ordine d'acquisto, dal momento in cui questo viene parserizzato da uno dei driver, rimane sempre lo stesso. Ciò è stato possibile grazie a:

1. una preliminare identificazione univoca dei dati da fornire ai vari attori del sistema;
2. la descrizione del formato dei dati attraverso *XML Schema Definition*;
3. l'implementazione della classe Java che rappresenta l'ordine d'acquisto sulla base della definizione XSD;
4. la possibilità di usare serializzatori Java distinti nelle varie fasi di trattamento dell'oggetto (un serializzatore binario per JMS ed uno XML per SOAP).

L'identificazione dei dati è partita dall'analisi dei campi richiesti dalla routine BAPI che inserisce l'ordine dentro SAP R/3 (`BAPI_SALESORDER_CREATEFROMDAT2`, vedi anche § 2.3). I dati fondamentali richiesti dalla routine sono riportati in tabella 1.

Campo	***Obbl***	***Spiegazione***	***Tipologia***
`DOC_TYPE`	Sì	Tipologia dell'ordine d'acquisto	Default
`SALES_ORG`	Sì	ID dell'organizzazione dell'azienda ricevente	Default
`DISTR_CHAN`	Sì	ID del canale distributivo dell'azienda ricevente	Default
`DIVISION`	Sì	ID della divisione dell'azienda ricevente	Default
`PARTN_ROLE`	Sì	Ruolo del cliente nel sistema SAP	Default
`PART_NUMB`	Sì	Identificatore del cliente nel sistema SAP	Facoltativo
`PURCH_NO_C`	No	Identificatore dell'ordine nel SIA del cliente	Facoltativo
`MATERIAL`	No	Codice SAP del materiale ordinato	Obbligatorio
`REQ_QTY`	No	Quantità di materiale ordinato	Obbligatorio

Tabella 1: Campi richiesti dalla routine ABAP per l'inserimento di un ordine di acquisto

La colonna *Obbl* della tabella fa riferimento alla necessità o meno di fornire tale campo alla funzione BAPI.

I campi `SALES_ORG`, `DISTR_CHAN`, `DIVISION`, `DOC_TYPE` e `PARTN_ROLE` hanno significato soltanto internamente al sistema R/3: il cliente non dovrebbe avere l'onere di specificarli ogni volta che compila un ordine di acquisto. Per questo, vengono configurati una volta per tutte con dipendenza dal `PARTN_NUMB` e sono definiti come "*default*". I loro valori sono conservati in un database locale a Zlatan, che potrebbe anche essere un estratto del database di SAP R/3.

Il `PARTN_NUMB` a sua volta può essere ricavato in base al mittente del messaggio: questo metodo da un lato richiede la presenza di una tabella con tutte le corrispondenze `(ADDRESS, PARTN_NUMB)`, dall'altro fornisce un semplice meccanismo per effettuare un primo controllo di sicurezza e filtrare l'accesso al sistema soltanto agli utenti registrati in tabella. L'altro campo facoltativo è `PURCH_NO_C`: non è richiesto da R/3 ed eventualmente può essere generato in automatico da Zlatan.

In definitiva, l'unica informazione essenziale che deve essere fornita dal cliente è la lista dei codici materiale da ordinare e le relative quantità.

Possiamo quindi esprimere due semplici dipendenze funzionali che ci permettono di modellare il database per il completamento dei dati dell'ordine di acquisto.

```
PROTOCOL, FROM_ADDRESS → PARTN_NUMB
PARTN_NUMB → DOC_TYPE, PARTN_ROLE, SALES_ORG, DISTR_CHAN, DIVISION
```

Le due dipendenze determinano la presenza nel database delle tabelle `SENDERS` e `DEFAULT_VALUES`.

La struttura dati necessaria al sistema non si limita però alle informazioni richieste da R/3 e componenti l'ordine di acquisto: una terza tabella (`REPLIES`) è necessaria per conoscere a quali indirizzi un cliente vuole ricevere la conferma della ricezione dell'ordine. Tale tabella è composta dai campi `(PROTOCOL, TO_ADDRESS, PARTN_NUMB)` ed è *tuttochiave*, poiché un cliente può chiedere l'invio di più messaggi di conferma a indirizzi/protocolli diversi e più clienti possono richiedere l'invio dei messaggi allo stesso indirizzo come informazione.

Riassumendo, un ordine d'acquisto in Zlatan è composto dai campi elencati in tabella 2.

Campo	***Molteplicità***	***Spiegazione***
`From`	1	Indirizzo di provenienza del `PurchaseOrder`
`Protocol`	1	Protocollo del canale di provenienza
`PartnerNumber`	0...1	ID del cliente nel sistema SAP
`PurchaseOrderNumber`	0...1	Numero dell'ordine di acquisto
`PartnerRole`	0...1	Ruolo del cliente nel sistema SAP
`DocumentType`	0...1	Tipo del `PurchaseOrder`
`DistributionChannel`	0...1	ID del canale di distribuzione dei materiali
`SalesOrganization`	0...1	ID dell'organizzazione di vendita dei materiali
`SalesDivision`	0...1	ID della divisione di vendita dei materiali
`Material`	1...n	Coppia di valori `(Code, Quantity)`

Tabella 2: Formato di un PurchaseOrder

Il cliente che invia l'ordine è tenuto a compilare soltanto i campi `From`, `Protocol` e a fornire almeno una coppia di valori `(Material.Code, Material.Quantity)`; eventualmente, se ne è a conoscenza, può fornire il `PartnerNumber` ed il `PurchaseOrderNumber`: un secondo attore del sistema si occuperà di completare l'ordine nelle sue parti mancanti, recuperando i dati dal database formato dalle tre tabelle descritte sopra (vedi anche § 3.4.2.2).

3.4.1 Il modulo IBRA

In questo paragrafo si analizza in dettaglio la struttura e l'implementazione di IBRA.

Il dispatcher, i moduli ed i driver, con l'eccezione del driver HTTP, una volta avviati appaiono all'utente sotto forma di icona nella *system tray* (vedi figura 19). Cliccando sull'icona con il tasto destro, si accede ad un semplice menu, dal quale è possibile tra le altre cose visualizzare la console dell'applicazione.

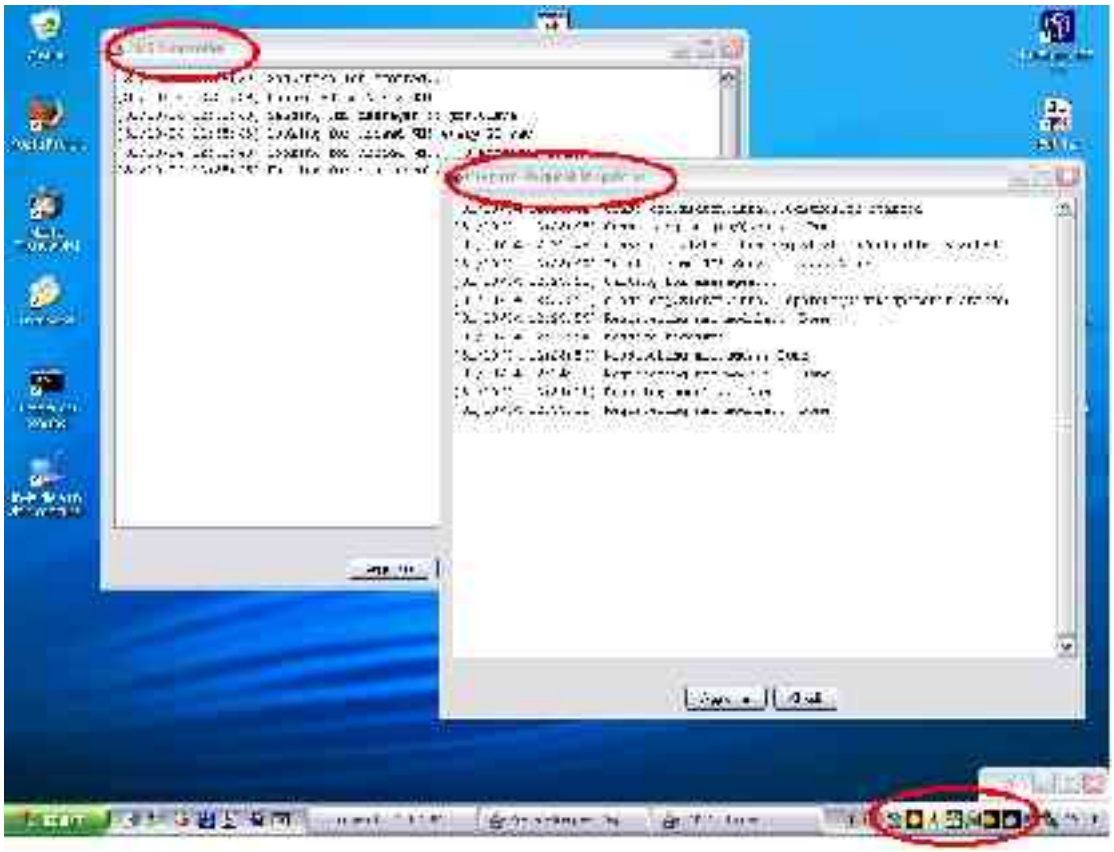

Figura 19: icone dei moduli nella system tray ed esempio di due console

Alternativamente, è possibile configurare le applicazioni per la visualizzazione in modalità "*console*" (vedi fig. 25 a pag. 99), dove l'output viene inviato sull'uscita standard (es. per ambienti non Windows).

3.4.1.1 HttpController

`HttpController` è il driver che si occupa della gestione dell'ingresso via HTTP. Essendo implementato con una servlet, necessita di un container (es. Apache Tomcat) o di un application server Java per l'esecuzione.

Per contattare il driver è necessario postare un messaggio HTTP con un contenuto simile a quello del listato 33.

```
POST /zlatan/HttpController HTTP/1.1
Host: 127.0.0.1
[cut]
Content-Length: 107

partnerNumber=1177&PONumber=PO-1177-181004-1&mat1Code=R-
1100&mat1Qty=10&mat2Code=R-1103&mat2Qty=20&matNum=2
```

Listato 33: post di esempio per il driver HTTP

I dati ricevuti vengono utilizzati per creare un oggetto di tipo `PurchaseOrderType` (vedi § 3.4.2.1). Questo oggetto viene quindi fornito ad un `PurchaseOrderSender` privato all'istanza della servlet, che si occupa di:

1. serializzare il `PurchaseOrderType` in formato binario;
2. incapsularlo in un messaggio JMS;
3. inviarlo al dispatcher.

Il post HTML deve contenere obbligatoriamente almeno il `partnerNumber` ed un materiale: nel caso in cui non venga specificato un numero ordine d'acquisto, la servlet ne genera uno univoco in automatico.

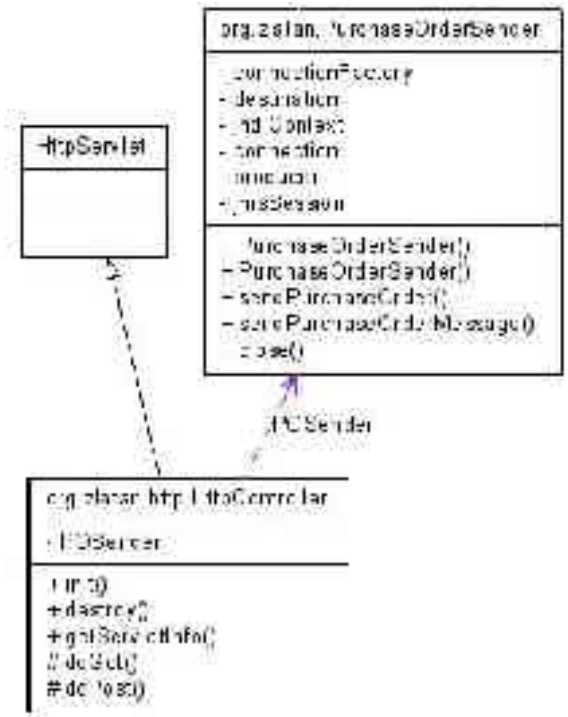

Figura 20: class diagram di HttpController

Il campo `Protocol` dell'ordine di acquisto contiene la stringa `http`, mentre il `From` è nel formato `//partnerNumber@host:port/servlet`.

Come gli altri driver, `HttpController` è configurabile. Trattandosi di una servlet, questo driver dispone già di un meccanismo di configurazione ed inizializzazione, basato sui valori dei parametri presenti nel file `web.xml`. I parametri configurabili sono i nomi JNDI della coda JMS per l'invio dei messaggi al `PrepareRequestDispatcher` e la relativa connection factory.

Nel listato 34 sono riportati i pezzi di codice più significativi della servlet.

```
protected void doPost(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.write("<html><head>\n\t<title>Zlatan - ... </head>");

  String partnerNumber =
    request.getParameter("partnerNumber").trim();
  if (partnerNumber.equals("")) {
    out.write("<p><b>Errore</b></p>");
    // cut
    out.close();
    return;
  }

  String PONumber = request.getParameter("PONumber").trim();
  int matNum = Integer.parseInt(
    request.getParameter("matNum").trim());

  // se non è stato specificato un numero interno dell'ordine
  // dal cliente, ne viene assegnato uno dal sistema

  if (PONumber.equals(""))
  PONumber = PONumber.concat(partnerNumber).concat("-").
    concat(Long.toString(System.currentTimeMillis()));

  try {

    // crea ed inizializza l'ordine di acquisto

    PurchaseOrderType PO = new PurchaseOrderType();
    PO.setPurchaseOrderNumber(PONumber);
    PO.setPartnerNumber(partnerNumber);
    PO.setProtocol("http");
    PO.setFrom("//" + partnerNumber  + "@" +
      request.getRequestURL().toString().substring(7));
    MaterialType[] mat  = new MaterialType[matNum];
    for (int i = 1; i <= matNum; i++) {
      String matCode = request.getParameter("mat" +
        i + "Code");
```

```
        PositiveInteger matQty = new PositiveInteger(
          request.getParameter("mat" + i + "Qty"));
        mat[i - 1] = new MaterialType();
        mat[i - 1].setCode(matCode);
        mat[i - 1].setQuantity(matQty);
      }
      MaterialsType mats = new MaterialsType();
      mats.setMaterial(mat);
      PO.setMaterials(mats);

      // invia l'ordine al dispatcher

      POSender.sendPurchaseOrder(PO);
      out.write("<p>Il messaggio &egrave; " +
        " stato inoltrato.</p>");
      // cut
    }
    catch (JMSException e) {
      // cut
      throw new ServletException("...");
    }
    // cut
    out.close();
}
```

Listato 34: codice di HttpServlet

3.4.1.2 Pop3Controller

`Pop3Controller` è il driver per la gestione dell'ingresso via SMTP: è implementato con un demone che accede ad una casella di posta elettronica utilizzando le librerie *JavaMail* di J2EE. Ad un intervallo di secondi configurabile, il demone controlla la presenza di eventuali messaggi contenenti ordini di acquisto, li parserizza e li invia al dispatcher.

Un ordine d'acquisto spedito via email ha il formato esemplificato nel listato 35.

```
Message-ID: <4173797A.1030700@tdnet.it>
[cut]
To:  zlatan@tele2.it
Subject: Purchase Order
Content-Type: text/plain; charset=ISO-8859-1; format=flowed
Content-Transfer-Encoding: 7bit

ponumber PO-1177-181004-2
material R-1004   100
material R-1105   200
```

Listato 35: messaggio di posta di esempio per il driver POP3

Il messaggio deve contenere le stringhe `Purchase` e `Order`, minuscole o maiuscole, nel subject; deve essere di tipo `text/plain` e può contenere le parole chiave `PartnerNumber`, `PONumber` e `Material`. Nel messaggio possono essere inserite altre stringhe.

`Pop3Controller`, come `HttpController`, spedisce i messaggi utilizzando un `PurchaseOrderSender`. I messaggi spediti vengono cancellati dalla casella di posta elettronica. Il `PurchaseOrderType` creato contiene la stringa `pop3` nel campo `Protocol` e l'indirizzo email del mittente in `From`.

I parametri configurabili di `Pop3Controller` si trovano nel file `pop3controller-config.xml` e comprendono:

- `host`, `username` e `password` della casella di posta elettronica;
- nomi JNDI della connection factory e della code del dispatcher;
- intervallo (in secondi) di attivazione del demone;
- icona da visualizzare nella system tray e nome dell'applicazione.

Uno degli aspetti interessanti di `Pop3Controller` è il meccanismo per marcare i messaggi nuovi trovati nella cartella, dal momento che il protocollo POP3 non supporta nativamente la distinzione tra messaggi già letti o non ancora letti. Questa funzione è riportata insieme ad altre nelle parti di codice del listato 36.

```
public class Pop3Controller {

  class Pop3ControllerThread extends TimerTask {

    private Properties          props;
    private Session             session;
    private Store               store;
    private Folder              folder;
```

```
  private String               lastRead;
  private String               host;
  private String               username;
  private String               password;
  private PurchaseOrderSender  sos;
  private DateFormat           formatter;

  public Pop3ControllerThread(String host, String username,
    String password, String connFact, String destName) {

    // inizializzazione

    this.host = host;
    // cut
    props = new Properties();
    session = Session.getDefaultInstance(props, null);
    lastRead = new String("");
    // cut
    sos = new PurchaseOrderSender(connFact, destName);
    // cut
  }

  // cut

  public void run() {

    try {

      // connessione alla casella di posta

      store = session.getStore("pop3");
      store.connect(host, username, password);
      folder = store.getFolder("INBOX");
      folder.open(Folder.READ_WRITE);
      int msgNumber = folder.getMessageCount();
      MimeMessage current = null;

      // controllo dei nuovi messaggi

      for (int index = msgNumber; index > 0; index--) {
        current = (MimeMessage) folder.getMessage(index);
        if (current.getMessageID().equals(lastRead))
          break;

        // cut
        // elabora messaggio

        String subject = current.getSubject();
        Object content = current.getContent();
        if ((content instanceof String) &&
          (subject.toLowerCase().indexOf("purchase") > -1)
```

```
        && (subject.toLowerCase().indexOf("order") > -1))
      {

        StringTokenizer st =
          new StringTokenizer((String) content);
        PurchaseOrderType PO = new PurchaseOrderType();
        Vector matv = new Vector();
        PO.setProtocol("pop3");
        String from = current.getFrom()[0].toString();
        // cut
        PO.setFrom(from);
        while (st.hasMoreTokens()) {
          String token = st.nextToken();
          if (token.equalsIgnoreCase("PartnerNumber"))
            PO.setPartnerNumber(st.nextToken());
          if (token.equalsIgnoreCase("PONumber"))
            PO.setPurchaseOrderNumber(st.nextToken());
          if (token.equalsIgnoreCase("Material")) {
            MaterialType mat = new MaterialType();
            mat.setCode(st.nextToken());
            mat.setQuantity( ... );
            matv.add(mat);
          }
        }
        MaterialType[] mat =
          new MaterialType[matv.size()];
        for (int i = 0; i < matv.size(); i++) {
          // cut
        }
        MaterialsType mats = new MaterialsType();
        mats.setMaterial(mat);
        PO.setMaterials(mats);

        if (PO.getPurchaseOrderNumber() != null) {
          if (PO.getPurchaseOrderNumber().equals(""))
            PO.setPurchaseOrderNumber(
              current.getMessageID());
        }
        else
          PO.setPurchaseOrderNumber(
            current.getMessageID());
        sos.sendPurchaseOrder(PO);
        current.setFlag(Flags.Flag.DELETED, true);
        // cut
      }
    }

    if (msgNumber != 0)
      lastRead = ((MimeMessage)
        folder.getMessage(msgNumber)).getMessageID();
    else
```

```
            lastRead = new String();
          folder.close(true);
          store.close();
          // cut
        }
        catch (javax.jms.JMSException e){ ... }
        catch (IOException e){ ... }
        catch (MessagingException e){ ... }
      }
    }

    private Timer timer;

    public Pop3Controller(int seconds, String host,
      String username, String password,
      String connFact, String destName) {

      // attiva il thread a tempo

      timer = new Timer();
      timer.scheduleAtFixedRate(new Pop3ControllerThread(
        host, username, password, connFact, destName),
        0, seconds*1000);
    }

    public static void main(String[] args) {

      // cut
      try {
      ConfigFile cf =
        new ConfigFile("pop3controller-config.xml", false);
      // cut
      }
      catch (InvalidDataException e) { ... }
      catch (Exception e) { ... }
      new Pop3Controller(seconds, host, username,
        password, connFact, destName);
      // cut
    }
}
```

Listato 36: codice di Pop3Controller

3.4.1.3 SmsController

`SmsController` è l'equivalente di `Pop3Controller` per la gestione di ordini d'acquisto provenienti da SMS. Questo demone dovrebbe in teoria accedere ad una casella telefonica via Internet, grazie ad un abbonamento presso uno dei provider che offrono tale servizio. L'implementazione realizzata in realtà utilizza come ricevitore per i messaggi una scheda GSM collegata attraverso una porta di comunicazione seriale e, come protocollo, il set di comandi AT supportati dal GSM.

Nel caso specifico è stata utilizzata la porta virtuale *COM1*, mappata attraverso un driver alla porta *IrDA* per collegare via infrarossi un cellulare *Nokia 8310*.

Un ordine d'acquisto inviato via SMS ha un formato simile a quello inviato via email: la differenza è che, mentre nel messaggio di posta elettronica è necessario specificare le parole chiave nel subject, nel caso del messaggio SMS le prime due lettere del body devono essere `PO`.

`SMSController` utilizza le librerie *JavaCOMM* della Sun per gestire la comunicazione tramite le porte seriali e, come accennato, alcuni comandi del protocollo AT GSM per leggere i messaggi.

Il comportamento del driver è simile a `Pop3Controller`: la differenza principale consiste nel fatto che il protocollo AT GSM supporta nativamente la marcatura dei messaggi non letti, e quindi non è necessario tenere traccia dei messaggi già esaminati. Il campo `From` del messaggio inviato contiene il numero di telefono del mittente mentre il campo `Protocol` contiene la stringa `sms`.

Il protocollo di trasferimento dati di IrDA prevede di chiudere la connessione nel caso in cui questa non venga utilizzata per un determinato periodo di tempo. Per questo, indipendentemente dal valore del tempo di attivazione del demone, il processo ogni cinque secondi effettua un ping alla porta per mantenere viva la connessione.

L'implementazione del modulo ha anche la struttura identica a quella di `Pop3Controller`: una *inner class* derivata da `TimerTask` compie l'elaborazione vera e propria nel metodo `run()`. In figura 21 è riportato il diagramma UML della classe interna.

La classe `MobilePhoneConnection` viene utilizzata nella lettura degli SMS mentre nell'invio del messaggio JMS, come per gli altri driver, viene usato un `PurchaseOrderSender`.

I parametri configurabili si trovano in `smscontroller-config.xml` e sono:

- nome della porta alla quale è collegata la scheda GSM;
- nomi JNDI della connection factory e della code del dispatcher;
- intervallo (in secondi) di attivazione del demone;
- icona da visualizzare nella system tray e nome dell'applicazione.

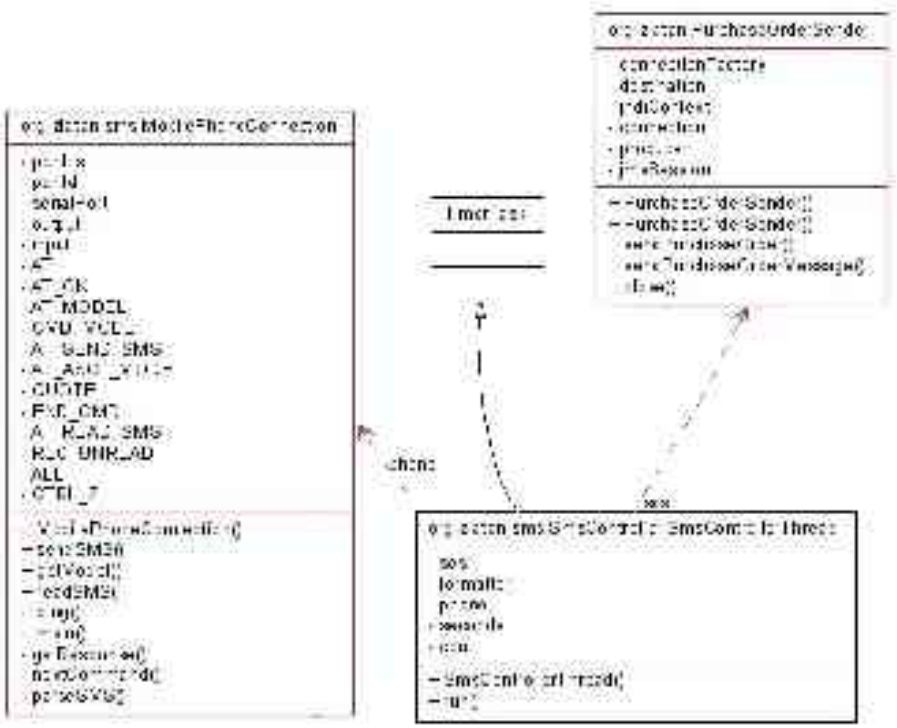

Figura 21: class diagram di SmsControllerThread

Nel listato 37 viene riportato il metodo `run()` della classe `SmsControllerThread`, in quello successivo invece i frammenti principali di `MobilePhoneConnection`, dove è evidenziato l'uso dei comandi AT GSM per la comunicazione con il telefono cellulare.

```
public void run() {

  try {

    if ((cont % seconds) == 0) {

      // preleva dalla scheda GSM i messaggi non letti

      // cut
      Vector unread = phone.readSMS();
      int msgNumber = unread.size();
      SMSMessage current = null;
      for (int index = 0; index < msgNumber; index++) {
        current = (SMSMessage) unread.elementAt(index);
        // cut

        // elabora il messaggio

        if (current.getText().indexOf("PO") == 0) {
          StringTokenizer st =
            new StringTokenizer(current.getText());
          PurchaseOrderType PO = new PurchaseOrderType();
          Vector matv = new Vector();
          PO.setProtocol("sms");
```

```
            PO.setFrom(current.getSender());
            while (st.hasMoreTokens()) {
              // parserizza il messaggio come in SmsController
              // cut
            }
            // cut
            PO.setMaterials(mats);
            // cut
            sos.sendPurchaseOrder(PO);
          }
        }
      } else {
        if ((cont % 5) == 0) { phone.ping(); }
      }
      cont++;
    }
    catch (javax.jms.JMSException e) { ... }
    catch (IOException e){ ... }
}
```

Listato 37: codice di SmsController.SmsControllerThread.run()

```
// cut
public class MobilePhoneConnection {

  // stringhe per il set di comandi AT GSM

  private static final String AT            = "AT";
  private static final String AT_OK         = "OK";
  private static final String AT_MODEL      = "AT+CGMM";
  private static final String CMD_MODE      = "+++";
  private static final String AT_SEND_SMS   = "AT+CMGS";
  private static final String AT_ASCII_MODE = "AT+CMGF=1";
  private static final String QUOTE         = "\"";
  private static final String END_CMD       = "\r";
  private static final String AT_READ_SMS   = "AT+CMGL";
  private static final String REC_UNREAD    = "REC UNREAD";
  private static final String ALL           = "ALL";
  private static final String CTRL_Z        =
    Character.toString((char) 26);

  private Enumeration         portList;
  private CommPortIdentifier  portId;
  private SerialPort          serialPort;
  private OutputStream        output;
  private InputStream         input;

  public MobilePhoneConnection(String portName)
  throws IOException {

    // carica i driver, se non è ancora stato fatto
```

```
    PropsDllLoader pdl = new PropsDllLoader();
    try {
      pdl.loadJavaxCommProperties();
      pdl.loadWin32DriverLibrary();
    } catch (Exception e) { ... }

    // ottiene il controllo della porta

    portList = CommPortIdentifier.getPortIdentifiers();
    while (portList.hasMoreElements()) {
      portId = (CommPortIdentifier) portList.nextElement();
      if (portId.getPortType() ==
        CommPortIdentifier.PORT_SERIAL) {
        if (portId.getName().equals(portName)) {
          try {
            serialPort = (SerialPort)
              portId.open("MobilePhoneConnection", 2000);
          } catch (PortInUseException e) {
            throw new IOException(portName + " is in use.");
          }
          output = serialPort.getOutputStream();
          input  = serialPort.getInputStream();
          try {
            serialPort.setSerialPortParams(19200,
              SerialPort.DATABITS_8,
              SerialPort.STOPBITS_1,
              SerialPort.PARITY_NONE);
          } catch (UnsupportedCommOperationException e) {
            throw new IOException(" ... ");
          }
        }
      }
    }
}

// cut

private String nextCommand(boolean print)
throws IOException {

    StringBuffer ret = new StringBuffer();
    boolean ok = false;
    while (true) {
      int c = input.read();
      ret.append((char) c);
      if (print) System.out.print((char) c);
      if (c == 'K') {
        if (ok) break;
      }
      else {
        if (c == 'O') ok = true; else ok = false;
      }
```

```
    }
    return ret.toString().trim();
  }

  private Vector parseSMS(String response) throws IOException {

    int i, j;
    String line, sms, temp, originator, text;
    Vector msgs = new Vector();
    BufferedReader reader;
    sms = new String("");
    reader = new BufferedReader(new StringReader(response));
    reader.readLine();
    line = reader.readLine();
    while ((line != null) && (!line.equalsIgnoreCase("OK"))) {
      if (line.indexOf("+CMGL") == 0) sms = line;
      else
        if ((line.indexOf("+CMGL") == -1) &&
          (line.length() > 0)) sms = sms + "~" + line;
        else sms = "";
      if ((sms.indexOf("\"REC READ\"") > -1) ||
        (sms.indexOf("\"REC UNREAD\"") > -1)) {
        if (sms.indexOf("~") != -1) {
          i = 6;
          temp = "";
          while (sms.charAt(i) != ',') {
            temp += sms.charAt(i);
            i ++;
          }
          i = sms.indexOf('"') + 1;
          i = sms.indexOf('"', i) + 1;
          i = sms.indexOf('"', i) + 1;
          temp = "";
          while (sms.charAt(i) != '"') {
            temp += sms.charAt(i);
            i ++;
          }
          originator = temp;
          i = sms.indexOf('~') + 1;
          text = sms.substring(i);
          SMSMessage smsTemp = new SMSMessage();
          smsTemp.setSender(originator);
          smsTemp.setText(text);
          msgs.add(smsTemp);
        }
      }
      else if (sms.indexOf("\"STO SENT\"") > -1);
      else if (sms.indexOf("\"STO UNSENT\"") > -1);
      line = reader.readLine();
    }
    reader.close();
    return msgs;
```

```
  }

  public String getModel() throws IOException {

    output.write(new String(MobilePhoneConnection.AT_MODEL +
      MobilePhoneConnection.END_CMD).getBytes());
    output.flush();
    String response = this.nextCommand(false);
    return response.substring(8,
      response.length() - 2).trim();
  }

  public Vector readSMS() throws IOException {

    output.write(
      new String(MobilePhoneConnection.AT_ASCII_MODE +
      MobilePhoneConnection.END_CMD).getBytes());
    output.write(new String(MobilePhoneConnection.AT_READ_SMS
      + "=" + MobilePhoneConnection.QUOTE +
      MobilePhoneConnection.REC_UNREAD +
      MobilePhoneConnection.QUOTE +
      MobilePhoneConnection.END_CMD).getBytes());
    output.flush();
    this.nextCommand(false);
    String response = this.nextCommand(false);
    return this.parseSMS(response);
  }

  public void ping() throws IOException {

    output.write(new String(MobilePhoneConnection.AT +
      MobilePhoneConnection.END_CMD).getBytes());
    this.nextCommand(false);
  }

  // cut
}
```

Listato 38: codice di MobilePhoneConnection

3.4.1.4 RmiController

`RmiController` è il driver architetturalmente e concettualmente più semplice tra tutti quelli implementati. Grazie alla facilità d'uso del protocollo RMI, è stato infatti possibile realizzare una classe che implementa una sola funzione `insertPO()`. Tale funzione si occupa semplicemente di inviare l'ordine di acquisto ricevuto come parametro attraverso un `PurchaseOrderSender`. La funzione `main()` della stessa classe implementa il server che rende disponibile la funzione attraverso la registrazione al `rmiregistry`.

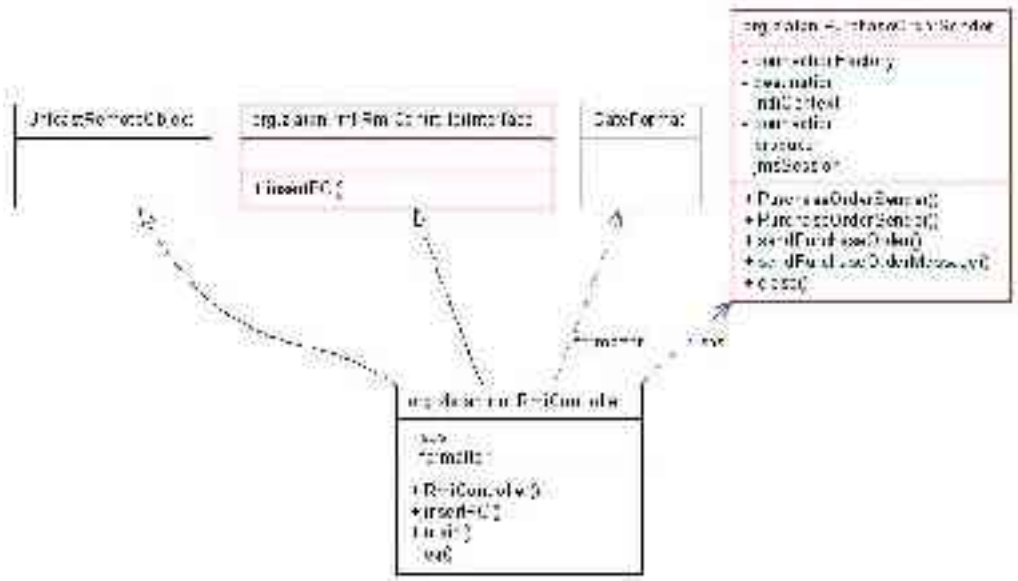

Figura 22: class diagram di RmiController

Nel listato 39 è riportato, ad eccezione del metodo `main()`, simile a quello degli altri driver, il codice della classe.

```
// cut
public class RmiController
extends UnicastRemoteObject
implements RmiControllerInterface {

  private PurchaseOrderSender sos;
  private DateFormat formatter;

  public RmiController(String connFact, String destName)
  throws RemoteException {

    super();
    // cut

    try {
      sos = new PurchaseOrderSender(connFact, destName);
    } catch (Exception e) { ... }
    log(this.getClass().toString() + " started");
  }
```

```
  private void log(String s) {
    System.out.println(formatter.format(new Date()) + s);
  }

  public void insertPO(org.zlatan.wsdl.PurchaseOrderType po)
  throws java.rmi.RemoteException {

    log("PurchaseOrder received");
    if ((po.getPartnerNumber() == null) &&
       (po.getFrom() == null))
         throw new RemoteException(" ... ");
    if (po.getMaterials().getMaterial().length == 0)
      throw new RemoteException(" ... ");
    po.setProtocol("rmi");
    String purchaseOrderNumber = null;
    if (po.getPartnerNumber() != null)
      purchaseOrderNumber = new String(po.getPartnerNumber()).
        concat("-").concat(Long.toString(
        System.currentTimeMillis()));
    else
      purchaseOrderNumber = new String(po.getFrom()).
        concat("-").concat(Long.toString(
        System.currentTimeMillis()));
    if (po.getPurchaseOrderNumber() != null) {
      if (po.getPurchaseOrderNumber().trim().
        equalsIgnoreCase(""))
          po.setPurchaseOrderNumber(purchaseOrderNumber);
    }
    else po.setPurchaseOrderNumber(purchaseOrderNumber);
    try {
      sos.sendPurchaseOrder(po);
    } catch (javax.jms.JMSException e){ ... }
    log("PurchaseOrder sent");
  }

  // cut
}
```

Listato 39: codice di RmiController

L'utilizzo del driver è quello tipico degli oggetti remoti esposti via RMI: sono disponibili una interfaccia remota (`RmiControllerInterface`) ed uno stub (`RmiController_Stub`) che possono essere usati per realizzare un client Java.

Il listato 40 mostra un esempio di client che utilizza tali classi.

```
// cut
public class RmiClient {

  private static PurchaseOrderType preparaPO()
  throws Exception {

    PurchaseOrderType po = new PurchaseOrderType();
```

```
    po.setPartnerNumber("PARTN001");
    po.setFrom("PARTN001");
    MaterialsType mats = new MaterialsType();
    MaterialType[] mat = new MaterialType[2];
    mat[0] = new MaterialType();
    mat[0].setCode("TESTMAT1");
    mat[0].setQuantity(new PositiveInteger(
      new Integer(10).toString()));
    mat[1] = new MaterialType();
    mat[1].setCode("TESTMAT2");
    mat[1].setQuantity(new PositiveInteger(
      new Integer(20).toString()));
    mats.setMaterial(mat);
    po.setMaterials(mats);
    return po;
  }

  public static void main(String[] args) {

    if (System.getSecurityManager() == null)
      System.setSecurityManager(new RMISecurityManager());
    try {
      String name = "rmi://localhost:1099/RMIController";
      RmiControllerInterface rmictrl =
        (RmiControllerInterface) Naming.lookup(name);
      rmictrl.insertPO(preparaPO());
    } catch (Exception e) { ... }
  }
}
```

Listato 40: codice di RmiClient

3.4.1.5 PrepareRequestDispatcher

Il dispatcher di *IBRA* è costituito da tre classi: due classi estendono `Thread` ed una terza contiene il metodo `main()` che implementa il server vero e proprio. Il modello di programmazione di riferimento è quello del *server distribuito* illustrato nel § 2.1.4.1.

In particolare:

- `RegistrationController` si occupa di controllare la coda JMS fissa sulla quale vengono inviate le richieste di registrazione di nuovi moduli di processamento e le eventuali richieste di cancellazione;
- `POController` inoltra i messaggi in arrivo dai driver sulla coda del dispatcher alla coda temporanea di uno dei moduli registrati, utilizzando un algoritmo *round-robin*;
- `PrepareRequestDispatcher` implementa il server che lancia i due thread e possiede l'oggetto condiviso `RegisteredQueueList`, dove si trovano gli indirizzi di tutte le code temporanee dei vari moduli attualmente registrati: i metodi che vanno a lavorare su tale oggetto sono tutti *thread-safe*.

In figura 23 è riportato lo schema UML delle relazioni tra le classi citate.

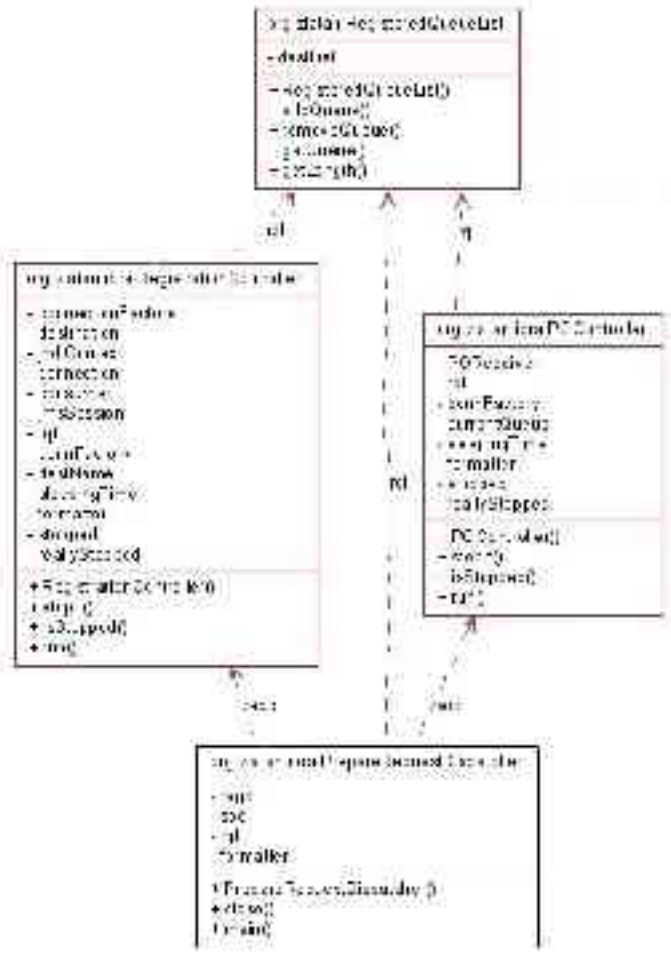

Figura 23: class diagram del dispatcher

Come evidenziato anche dallo schema, `PrepareRequestDispatcher` ha tra i suoi membri una istanza per ciascuno dei due thread ed un `RegisteredQueueList` condiviso il cui riferimento viene passato ai costruttori dei thread.

Anche i due thread interni a `PrepareRequestDispatcher` hanno tempi di attivazione configurabili in base alle necessità ed alle risorse del proprio sistema. La configurazione è effettuabile anche in questo caso utilizzando il file `preparerequestdispatcher-config.xml`, che contiene le seguenti proprietà:

- nomi JNDI della connection factory, della coda del dispatcher e della coda per la registrazione;
- intervallo (in secondi) di attivazione dei due thread;
- icona da visualizzare nella system tray e nome dell'applicazione.

Nel listato 41 e seguenti sono riportati gli estratti di codice più significativi dalle varie classi citate.

```
// cut
public class POController extends Thread {

  private PurchaseOrderReceiver  POReceiver;
  private RegisteredQueueList    rql;
  private String                 connFactory;
  private int                    currentQueue;
  private int                    sleepingTime;
  private DateFormat             formatter;
  private boolean                stopped;
  private boolean                reallyStopped;

  public POController(String connFactory,
    String receiverDestName, int sleepingTime,
    RegisteredQueueList rql) {

      super();
      // cut

      // inizializzazione
      this.currentQueue  = 0;
      // cut
      reallyStopped      = false;
      try {
        POReceiver = new PurchaseOrderReceiver(
          connFactory, receiverDestName);
      } catch (Exception e) { ... }
   }

    // cut

    public void run() {
```

```
      while (!stopped) {
        int maxQueueNumber = rql.getLength();
        PurchaseOrderSender POSender = null;
        if (maxQueueNumber > 0) {

          // algoritmo round-robin

          if (currentQueue >= maxQueueNumber)
            currentQueue = 0;
          try {
            Message pom =
              POReceiver.receivePurchaseOrderMessage();
            if (pom != null) {
              // cut
              // messaggio ricevuto, inoltro

              POSender = new PurchaseOrderSender(
                connFactory, rql.getQueue(currentQueue));
              POSender.sendPurchaseOrderMessage(pom);

              // algoritmo round-robin

              currentQueue = currentQueue + 1;
            }
          } catch (Exception e) { ... }
        }
        try { this.sleep(sleepingTime); }
        catch (InterruptedException e) {}
      }

      // shut down

      try {
        this.POReceiver.close();
      } catch (Exception e) {
        // cut
        System.exit(0);
      }
      reallyStopped = true;
    }
}
```

Listato 41: codice di POController

```
// cut
public class RegistrationController extends Thread {

  private ConnectionFactory   connectionFactory;
  private Destination         destination;
  private Context             jndiContext;
  private Connection          connection;
  private MessageConsumer     consumer;
  private Session             jmsSession;
```

```
private RegisteredQueueList rql;
private String              connFactory;
private String              destName;
private int                 sleepingTime;
private DateFormat          formatter;
private boolean             stopped       = false;
private boolean             reallyStopped = false;

public RegistrationController(String connFactory,
   String destName, int sleepingTime,
   RegisteredQueueList rql)
   throws NamingException, JMSException {

   super();

   // inizializzazione membri privati
   // cut

   // inizializzazione JMS

   jndiContext        = new InitialContext();
   connectionFactory  = (ConnectionFactory)
     jndiContext.lookup(connFactory);
   destination        = (Queue) jndiContext.lookup(destName);
   connection         = connectionFactory.createConnection();
   jmsSession         = connection.createSession(
     false, Session.AUTO_ACKNOWLEDGE);
   consumer           =
     jmsSession.createConsumer(destination);
   connection.start();
   // cut
}

// cut

public void run() {

   while (!stopped) {

     Destination destination = null;
     Message     m           = null;
     TextMessage t           = null;
     try {

       // controlla la presenza di richieste
       // sulla coda fissa

       m = consumer.receiveNoWait();
       if (m != null) {
         destination = m.getJMSReplyTo();
         if (m instanceof TextMessage)
           t = (TextMessage) m;
```

```
          }
          if ((destination != null) && (t != null)) {
            if (t.getText().equalsIgnoreCase("create"))
              rql.addQueue(destination);
            if (t.getText().equalsIgnoreCase("delete"))
              rql.removeQueue(destination);
          }
        }
        catch (JMSException e) { ... }
        try { this.sleep(sleepingTime); }
        catch (InterruptedException e) {}
      }

      // shut down

      try {
        this.jndiContext.close();
        this.consumer.close();
        this.connection.close();
        this.jmsSession.close();
      } catch (Exception e) { ... }
      reallyStopped = true;
    }
}
```

Listato 42: codice di RegistrationController

```
// cut
public class PrepareRequestDispatcher {

  private RegistrationController regc;
  private POController           soc;
  private RegisteredQueueList    rql;
  private DateFormat formatter;
  // cut

  public PrepareRequestDispatcher(String connFactory,
    String destName, String regDestName, int socSleepingTime,
    int regcSleepingTime)
    throws NamingException, JMSException {

      // cut

      this.rql  = new RegisteredQueueList();
      this.soc  = new POController(
        connFactory, destName, socSleepingTime, rql);
      this.regc = new RegistrationController(
        connFactory, regDestName, regcSleepingTime, rql);
      // cut
      soc.start();
      regc.start();
  }
```

```
// cut

public void close() {

   while (rql.getLength() != 0) {}
   regc.stopIt();
   soc.stopIt();
   while (!(regc.isStopped() && soc.isStopped())) {}
}

public static void main(String[] args) {

   // cut
   PrepareRequestDispatcher prd =
     new PrepareRequestDispatcher(
     connFactory, destName, regDestName, socSecs, regcSecs);
   // cut
}
```

Listato 43: codice di PrepareRequestDispatcher

La corretta sequenza di attivazione di IBRA prevede l'avvio del dispatcher seguito da quello dei moduli (vedi fig. 24), mentre alla chiusura l'ordine dovrebbe essere invertito.

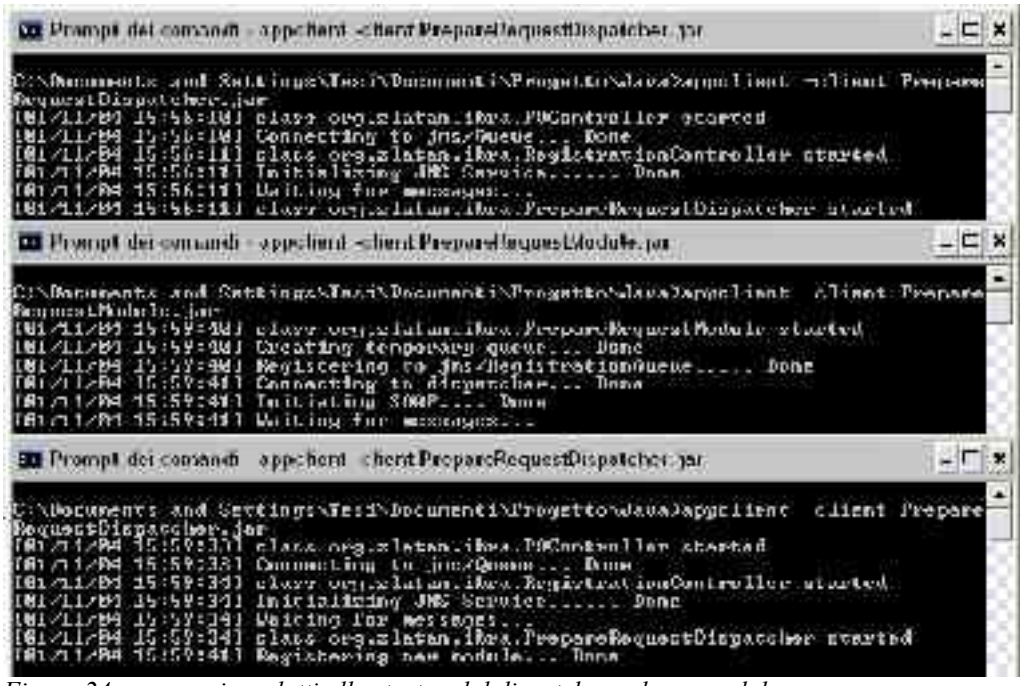

Figura 24: messaggi prodotti allo startup dal dispatcher e da un modulo

È interessante notare il meccanismo di *shut down* del dispatcher, che tenta di risolvere eventuali errori nella sequenza di disattivazione: alla pressione della combinazione `CTRL+C`, infatti, il dispatcher attende che tutti i `PrepareRequestModule` registrati vengano chiusi, quindi fa terminare in maniera pulita

i due thread interni e, solo quando questi sono effettivamente stoppati, termina anch'esso.

L'effetto di questo meccanismo viene evidenziato utilizzando l'interfaccia testuale al posto di quella grafica (`<mode>console</mode>` nei file di configurazione), come nella figura 25.

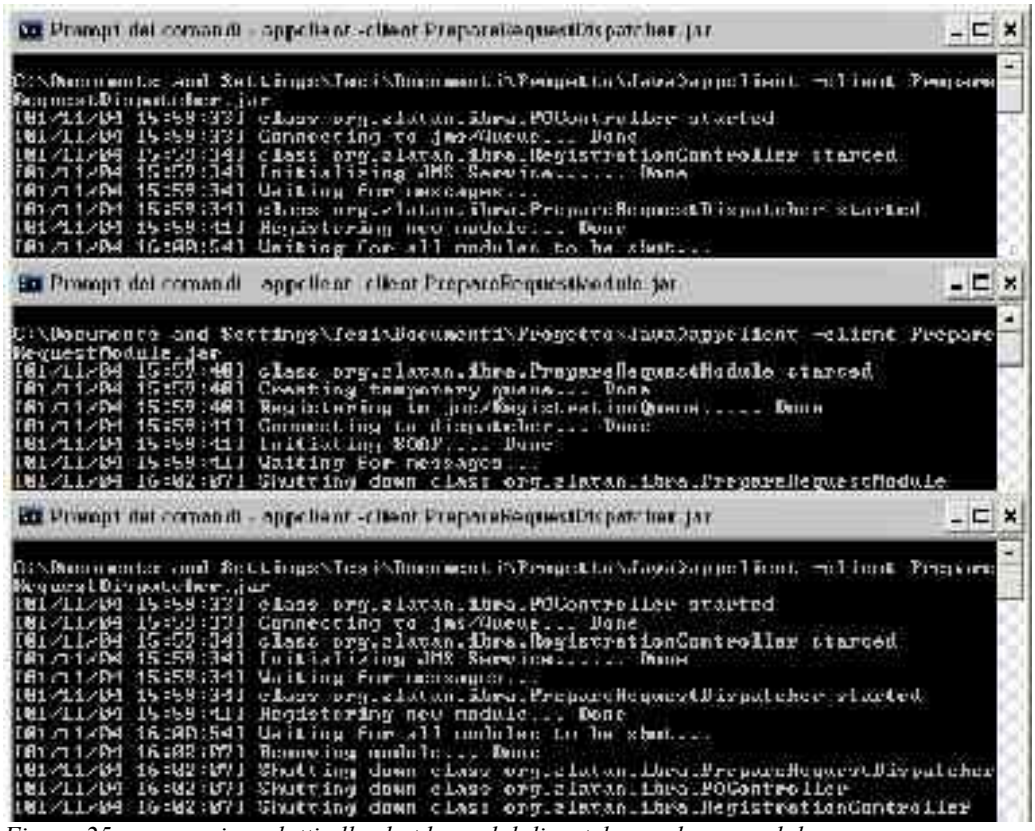

Figura 25: messaggi prodotti allo shutdown dal dispatcher e da un modulo

3.4.1.6 PrepareRequestModule

`PrepareRequestModule` è il modulo che si occupa del processamento delle richieste inoltrate dal dispatcher, implementando il ciclo di vita descritto nel § 2.1.4.1.

Il codice del modulo non è particolarmente complesso, ma contiene tre operazioni molto dispendiose e bloccanti che quindi devono essere parallelizzate nel contesto di IBRA. Le tre operazioni sono:

- deserializzazione del messaggio JMS binario contenente il `PurchaseOrderType` inoltrato da uno dei driver;
- serializzazione del messaggio in formato SOAP;
- invio del messaggio all'interprete BPEL per l'attivazione del processo ed attesa della chiusura della connessione HTTP (vedi anche § 2.2.3.2).

In ogni installazione di IBRA deve essere presente almeno una istanza di `PrepareRequestModule`. La replica delle istanze del modulo permette di aumentare le prestazioni ed il tempo di risposta del sistema.

I parametri configurabili del modulo, contenuti nel file `prepare requestmodule-config.xml`, sono:

- nomi JNDI della connection factory, della coda del dispatcher e della coda per la registrazione;
- valori dell'endpoint, dell'operazione e del namespace a cui inviare il messaggio SOAP per l'attivazione del processo BPEL;
- intervallo (in secondi) di attivazione del thread;
- icona da visualizzare nella system tray e nome dell'applicazione.

Nel listato 44 sono riportati i metodi principali dell'unica classe che implementa il server.

```
// cut
public class PrepareRequestModule extends Thread {

  // cut - membri per l'utilizzo di JMS
  // cut - membri per l'utilizzo di SOAP
  // cut - altri membri privati

  private TemporaryQueue tempQueue;
  private PurchaseOrderReceiver POReceiver;

  public PrepareRequestModule( ... )
  throws ... {

    // cut
    // inizializzazione

    // crea la coda temporanea per comunicare
    // con il dispatcher
```

```
    connectionFactory = (ConnectionFactory)
      jndiContext.lookup(connFactory);
    connection = connectionFactory.createConnection();
    jmsSession = connection.createSession(
      false, Session.AUTO_ACKNOWLEDGE);
    tempQueue = jmsSession.createTemporaryQueue();

    // invia il nome della coda
    // creata a RegistrationController

    Destination regDest =
      (Queue) jndiContext.lookup(regDestName);
    MessageProducer regProd =
      jmsSession.createProducer(regDest);
    TextMessage regMsg = jmsSession.createTextMessage();
    regMsg.setText("create");
    regMsg.setJMSReplyTo(tempQueue);
    regProd.send(regMsg);
    regProd.close();

    // crea il receiver associato alla propria coda temporanea

    POReceiver =
      new PurchaseOrderReceiver(connection, tempQueue);

    // ottiene un riferimento remoto all'engine BPEL

    this.endpoint = new URL(endpoint);
    this.service = new StartZlatanServiceLocator();
    this.port = service.getStartZlatan(this.endpoint);
}

public boolean processMessage()
throws SOAPException, javax.jms.JMSException,
java.rmi.RemoteException, javax.xml.rpc.ServiceException {

    PurchaseOrderType PO = POReceiver.receivePurchaseOrder();
    if (PO != null) {
      port.startZlatan(PO);
      return true;
    }
    return false;
}

// cut

public void close()
throws NamingException, JMSException, SOAPException,
java.rmi.RemoteException, javax.xml.rpc.ServiceException {
```

```
    synchronized (this) { closing = true; }

    // notifica la propria chiusura al dispatcher

    TextMessage m = jmsSession.createTextMessage();
    m.setText("delete");
    m.setJMSReplyTo(tempQueue);
    Destination regDest =
      (Queue) jndiContext.lookup(regDestName);
    MessageProducer regProd =
      jmsSession.createProducer(regDest);
    regProd.send(m);
    while (processMessage()) {}
    this.POReceiver.close();
    this.jmsSession.close();
    this.connection.close();
    this.jndiContext.close();
    this.soapConnection.close();
  }

  public void run() {

    while (!this.isClosing()) {
      try { this.processMessage(); }
      catch (Exception e) { ... }
      try { this.sleep(sleepingTime); }
      catch (InterruptedException e) {}
    }
  }

  public static void main(String args[]) {

    // cut
    PrepareRequestModule prm = new PrepareRequestModule(
      connFactory, destName, regDestName, endpoint,
      operationName, namespace, sleepingTime);
    prm.start();
    // cut
  }
}
```

Listato 44: codice di PrepareRequestModule

Nel codice sono state evidenziate le parti relative alla registrazione ed alla cancellazione presso il dispatcher, con l'utilizzo del metodo `setJMSReplyTo()` per trasportare il riferimento alla coda JMS temporanea creata (vedi anche § 2.1.4.1).

Il modulo utilizza uno *stub* generato con *WSDL2Java* per inviare il messaggio SOAP all'interprete BPEL (vedi § 3.4.2.1).

3.4.2 I web service

3.4.2.1 Il documento WSDL

Il documento WSDL contiene le definizioni dei tipi di dati utilizzati nell'intero sistema, le interfacce dei web service e le descrizioni necessarie per il processo BPEL (vedi § 2.2.1.1.1). Tali informazioni potrebbero anche essere strutturate logicamente su più file fisici, tramite l'utilizzo del costrutto `<import>` del WSDL e dei namespace. Ad esempio, si potrebbero definire i tipi in un file XSD, ogni web service in un proprio WSDL e le estensioni per il BPEL in un altro file. Per comodità, nel progetto le definizioni sono state riportate in un unico file.

Il documento inizia con la definizione di una serie di namespace (listato 45).

```
<definitions targetNamespace="http://wsdl.zlatan.org"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://wsdl.zlatan.org"
xmlns:impl="http://wsdl.zlatan.org"
xmlns:plt="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-
process/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
```

Listato 45: definizione dei namespace nel documento WSDL

Il namespace di riferimento è quello del WSDL. Il namespace `impl` serve per riferire gli stessi elementi definiti dentro il documento (è identico a `targetNamespace`); `plt` e `bpws` servono rispettivamente per le definizioni degli elementi `<partnerLinkType>` e `<property>`; `xs` per lo schema dei dati e `soap` per il binding *SOAP-over-HTTP*.

Nel listato 46 sono riportati i tipi di dati. I tipi complessi principali sono:

- `PurchaseOrderType`: il tipo che specifica il formato XML di un ordine di acquisto; contiene i campi descritti in tabella 2 (pag. 75);
- `CompletePurchaseOrderInputType` contiene i parametri di ingresso per il web service `CompletePurchaseOrder`, ovvero le chiavi per la ricerca dei valori di default nel database;
- `CompletePurchaseOrderOutputType` contiene, oltre ai campi recuperati nel database da `CompletePurchaseOrder`, anche la lista delle coppie `(Protocol, Address)` a cui deve essere inviata la risposta fornita da SAP R/3;
- `SAPOutputType` contiene il messaggio di risposta fornito da SAP R/3 sullo stato dell'inserimento;
- `SendResponseInputType` contiene i dati da fornire al web service `SendResponse` che invia la risposta al cliente (è semplicemente la combinazione della lista di indirizzi con il messaggio fornito da R/3).

I tipi semplici, come `DTType`, contengono alcune restrizioni secondarie (es. lunghezza della stringa) che per leggibilità non vengono riportate.

```
<types>

  <xs:schema  targetNamespace="http://wsdl.zlatan.org"
              xmlns="http://wsdl.zlatan.org">

    <xs:complexType  name="PurchaseOrderType">
      <xs:sequence>
        <xs:element name="From"    type="xs:string"
                    minOccurs="0" maxOccurs="1" />
        <xs:element name="Protocol"  type="xs:string"
                    minOccurs="1" maxOccurs="1" />
        <xs:element name="DocumentType"   type="DTType"
                    minOccurs="0" maxOccurs="1" />
        <xs:element name="SalesOrganization"
                    type="SalesOrganizationType"
                    minOccurs="0" maxOccurs="1" />
        <xs:element name="DistributionChannel"
                    type="DistributionChannelType"
                    minOccurs="0" maxOccurs="1" />
        <xs:element name="Division"  type="DivisionType"
                    minOccurs="0" maxOccurs="1" />
        <xs:element name="PartnerRole" type="PartnerRoleType"
                    minOccurs="0" maxOccurs="1" />
        <xs:element name="PartnerNumber"  type="xs:string"
                    minOccurs="0" maxOccurs="1" />
        <xs:element name="PurchaseOrderNumber"
                    type="xs:string"
                    minOccurs="0" maxOccurs="1" />
        <xs:element name="Materials" type="MaterialsType"
                    minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>

    <xs:complexType  name="MaterialsType">
      <xs:sequence>
        <xs:element name="Material" type="MaterialType"
                    minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="MaterialType">
      <xs:sequence>
        <xs:element name="Code" type="xs:string" />
        <xs:element name="Quantity"
                    type="xs:positiveInteger" />
      </xs:sequence>
    </xs:complexType>

    <xs:simpleType   name="DTType"> ... </xs:simpleType>

    <xs:simpleType   name="SalesOrganizationType">
```

```
   ...
</xs:simpleType>

<xs:simpleType  name="DistributionChannelType">
   ...
</xs:simpleType>

<xs:simpleType  name="DivisionType"> ... </xs:simpleType>

<xs:simpleType  name="PartnerRoleType">
   ...
</xs:simpleType>

<xs:complexType  name="CompletePurchaseOrderInputType">
   <xs:sequence>
      <xs:element name="From" type="xs:string"
                  minOccurs="1" maxOccurs="1" />
      <xs:element name="Protocol"  type="xs:string"
                  minOccurs="1" maxOccurs="1" />
      <xs:element name="PartnerNumber" type="xs:string"
                  minOccurs="0" maxOccurs="1" />
   </xs:sequence>
</xs:complexType>

<xs:complexType  name="CompletePurchaseOrderOutputType">
   <xs:sequence>
      <xs:element name="PartnerNumber" type="xs:string"
                  minOccurs="1" maxOccurs="1" />
      <xs:element name="DocumentType"  type="DTType"
                  minOccurs="1" maxOccurs="1" />
      <xs:element name="SalesOrganization"
                  type="SalesOrganizationType"
                  minOccurs="1" maxOccurs="1" />
      <xs:element name="DistributionChannel"
                  type="DistributionChannelType"
                  minOccurs="1" maxOccurs="1" />
      <xs:element name="Division"  type="DivisionType"
                  minOccurs="1" maxOccurs="1" />
      <xs:element name="PartnerRole" type="PartnerRoleType"
                  minOccurs="1" maxOccurs="1" />
      <xs:element name="Replies"   type="RepliesType"
                  minOccurs="1" maxOccurs="1" />
   </xs:sequence>
</xs:complexType>

<xs:complexType  name="RepliesType">
   <xs:sequence>
      <xs:element name="ReplyTo"   type="ReplyToType"
                  minOccurs="1" maxOccurs="unbounded" />
   </xs:sequence>
</xs:complexType>
```

```
    <xs:complexType  name="ReplyToType">
      <xs:sequence>
        <xs:element name="Address"   type="xs:string"
                    minOccurs="1"    maxOccurs="1" />
        <xs:element name="Protocol"  type="xs:string"
                    minOccurs="1"    maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>

    <xs:complexType  name="SAPOutputType">
      <xs:sequence>
        <xs:element name="PONumberRef" type="xs:string"
                    minOccurs="1" maxOccurs="1" />
        <xs:element name="PartnerNumberRef" type="xs:string"
                    minOccurs="1" maxOccurs="1" />
        <xs:element name="Message"   type="xs:string"
                    minOccurs="1"    maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>

    <xs:complexType  name="SendResponseInputType">
      <xs:sequence>
        <xs:element name="SAPOutput"
                    type="SAPOutputType" />
        <xs:element name="Replies"
                    type="RepliesType" />
      </xs:sequence>
    </xs:complexType>

    <xs:element name="PartnerNumberNotFoundElement"
                type="xs:string" />
    <xs:element name="PartnerNumberMismatchElement"
                type="xs:string" />

  </xs:schema>

</types>
```

Listato 46: definizione dei tipi nel documento WSDL

I due elementi definiti alla fine dello schema serviranno per la definizione dei SOAP fault: per i fault infatti, secondo le specifiche del WS-I, devono necessariamente essere forniti gli elementi e non i tipi ("*A `binding` [...] MUST refer, in each of its [...] `fault` elements, only to `part` element(s) that have been defined using the `element` attribute*", vedi [WSIBP]).

Nel listato 47 vengono riportate le definizioni astratte. Per scelta, ogni messaggio contiene una sola `<part>`: in questa maniera, la complessità della struttura dati si riversa, come visto, sulla definizione dei tipi piuttosto che su quella dei messaggi, ma si rende molto più chiara la descrizione dei servizi web.

Una scelta alternativa poteva essere quella, ad esempio, di non definire il tipo `SendResponseInputType` ma di inserire nel relativo `<message>` due `<part>` distinte, una di tipo `SAPOutputType` ed una di tipo `RepliesType`.

Da notare la riusabilità dei messaggi nella definizione dei `<portType>` (`PurchaseOrderMessage` viene usato sia in `InsertPurchaseOrderPT` che in `ZlatanServicePT`), oltre alla definizione di un messaggio (`DummyMessage`) che non viene utilizzato nel documento WSDL ma soltanto nel BPEL (vedi § 3.4.3).

Il `<portType> ZlatanServicePT` è quello che descrive l'interfaccia fornita dall'interprete BPEL per il processo: contiene infatti l'operazione per attivare il processo (`startZlatan`) e le operazioni di callback (`insertPOCallback` e le altre, usate per la gestione delle eccezioni come descritto nel § 2.2.3.2).

```
<message name="PurchaseOrderMessage">
  <part name="PurchaseOrder" type="impl:PurchaseOrderType" />
</message>
<message name="SAPOutputMessage">
  <part name="SAPOutput" type="impl:SAPOutputType" />
</message>
<message name="CompletePurchaseOrderInputMessage">
  <part name="CompletePurchaseOrderInput"
        type="impl:CompletePurchaseOrderInputType" />
</message>
<message name="CompletePurchaseOrderOutputMessage">
  <part name="CompletePurchaseOrderOutput"
        type="impl:CompletePurchaseOrderOutputType" />
</message>
<message name="SendResponseInputMessage">
  <part name="SendResponseInput"
        type="impl:SendResponseInputType" />
</message>
<message name="PartnerNumberNotFoundMessage">
  <part name="ProblemInfo"
        element="impl:PartnerNumberNotFoundElement" />
</message>
<message name="PartnerNumberMismatchMessage">
  <part name="ProblemInfo"
        element="impl:PartnerNumberMismatchElement" />
</message>
<message name="DummyMessage">
  <part name="string" type="xs:string" />
</message>

<portType     name="ZlatanServicePT">
  <operation  name="startZlatan">
    <input    message="impl:PurchaseOrderMessage" />
  </operation>
  <operation  name="insertPOCallback">
    <input    message="impl:SAPOutputMessage" />
```

```
  </operation>
  <operation  name="materialCodeError">
    <input    message="impl:SAPOutputMessage" />
  </operation>
  <operation  name="sapGeneralError">
    <input    message="impl:SAPOutputMessage" />
  </operation>
</portType>

<portType     name="CompletePurchaseOrderPT">
  <operation  name="completePO">
    <input
      message="impl:CompletePurchaseOrderInputMessage" />
    <output
      message="impl:CompletePurchaseOrderOutputMessage" />
    <fault    name="PartnerNumberNotFoundFault"
              message="impl:PartnerNumberNotFoundMessage" />
    <fault    name="PartnerNumberMismatchFault"
              message="impl:PartnerNumberMismatchMessage" />
  </operation>
</portType>

<portType     name="InsertPurchaseOrderPT">
  <operation  name="insertPO">
    <input    message="impl:PurchaseOrderMessage" />
  </operation>
</portType>

<portType     name="SendResponsePT">
  <operation  name="sendResponse">
    <input    message="impl:SendResponseInputMessage" />
  </operation>
</portType>
```

Listato 47: definizioni astratte nel documento WSDL

Nel listato 48 è riportato un unico esempio di binding a *SOAP-over-HTTP*. Tutti i `<binding>` sono standard e simili nella struttura a quello mostrato. Lo stile scelto per il `<binding>` è *rpc/literal*.

Le specifiche WSDL (vedi [WSDL]) prevedono tre possibili tipi di `<binding>`, che si differenziano fondamentalmente sull'operazione e sui tipi degli elementi (vedi anche [IBMWSDL]):

- *rpc/encoded*: nel messaggio SOAP sono specificati il nome dell'operazione invocata e tutti i tipi degli elementi passati;
- *rpc/literal*: nel messaggio SOAP è specificato soltanto il nome dell'operazione;
- *document/literal*: nel messaggio SOAP non è specificato né nome dell'operazione né quale sia il tipo di ogni elemento.

Document/literal è il tipo di binding più elastico, in quanto il corpo del messaggio SOAP è composto semplicemente da un documento XML senza strutture particolari: crea però dei problemi al momento del dispatching perché per risalire all'operazione da invocare bisogna utilizzare l'header HTTP *SOAPAction*. L'uso di tale header è tuttavia deprecato (addirittura eliminato dalle specifiche SOAP 1.2, [SOAP0]) perché introduce un legame diretto tra il protocollo HTTP e WSDL: in definitiva, *document/literal* è adatto a web service che esportano una sola operazione, per i quali l'azione di dispatching non è necessaria.

Il *Basic Profile* di *WS-I* ([WSIBP]) restringe i possibili binding soltanto a quelli *literal*, quindi la scelta di *rpc/literal* per i servizi di Zlatan diventa praticamente obbligata.

```
<binding  name="CompletePurchaseOrderBinding"
          type="impl:CompletePurchaseOrderPT">
  <soap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http" />
  <operation          name="completePO">
    <soap:operation  soapAction="" />
    <input>
      <soap:body  namespace="http://zlatan.org"
                  use="literal" />
    </input>
    <output>
      <soap:body  namespace="http://zlatan.org"
                  use="literal" />
    </output>
    <fault         name="PartnerNumberNotFoundFault">
      <soap:fault name="PartnerNumberNotFound"
                  namespace="http://zlatan.org”
                  use="literal" />
    </fault>
    <fault         name="PartnerNumberMismatchFault">
      <soap:fault name="PartnerNumberAddressMismatch"
                  namespace="http://zlatan.org"
                  use="literal" />
    </fault>
  </operation>
</binding>
```

Listato 48: esempio di definizione di un binding nel documento WSDL

Le definizioni dei servizi vengono riportate in listato 49. Notare che per il `<binding> ZlatanBinding`, implementato come detto sopra dall'interprete BPEL ed associato al processo, vengono definiti due `<service>` distinti: questo perché il PE di *GlobalBiz* richiede che ogni `<partnerLink>` sia in relazione biunivoca con una `<port>`. Dal momento che il servizio web esposto dal processo BPEL è impegnato in due `<partnerLink>` diversi (l'attivazione del servizio e la callback dell'inserimento in SAP), è necessario specificare due `<port>` distinti.

```
<service name="StartZlatanService">
  <port name="StartZlatan"
        binding="impl:ZlatanBinding">
    <soap:address location="/StartZlatan"/>
  </port>
</service>
<service name="ZlatanService">
  <port name="Zlatan"
        binding="impl:ZlatanBinding">
    <soap:address location="/Zlatan"/>
  </port>
</service>
<service name="CompletePurchaseOrderService">
  <port name="CompletePurchaseOrder"
        binding="impl:CompletePurchaseOrderBinding">
    <soap:address
        location="http://.../CompletePurchaseOrder"/>
  </port>
</service>
<service name="InsertPurchaseOrderService">
  <port name="InsertPurchaseOrder"
        binding="impl:InsertPurchaseOrderBinding">
    <soap:address location="http://.../InsertPurchaseOrder"/>
  </port>
</service>
<service name="InsertPurchaseOrderControllerService">
  <port name="InsertPurchaseOrderController"
        binding="impl:InsertPurchaseOrderControllerBinding">
    <soap:address
        location="http://.../InsertPurchaseOrderController"/>
  </port>
</service>
<service name="SendResponseService">
  <port name="SendResponse"
        binding="impl:SendResponseBinding">
    <soap:address location="http://.../SendResponse"/>
  </port>
</service>
```

Listato 49: definizione dei servizi nel documento WSDL

`InsertPurchaseOrderController` è un servizio che si interpone tra il processo BPEL ed `InsertPurchaseOrder`. Il suo ruolo è spiegato in § 3.4.2.3. Tale servizio ha la stessa interfaccia di `InsertPurchaseOrder` ma `<binding>` diverso: questo è dovuto al comportamento del tool *WSDL2Java* di Axis (vedi anche Appendice A).

Axis è stato usato per implementare in ambiente Java i web service. *WSDL2Java* (vedi anche [AXIS]) permette di creare automaticamente una serie di classi per l'implementazione o l'uso di un servizio a partire da un documento WSDL.

In particolare, WSDL2Java genera:

- un bean Java per ogni `<complexType>` o `<simpleType>` definito con XSD nella sezione `<types>`;
- una interfaccia Java per ogni `<portType>`;
- uno stub lato client, uno skeleton ed una classe per l'implementazione lato server del servizio per ogni `<binding>`;
- una interfaccia remota ed una classe per la risoluzione del servizio da lato client per ogni `<service>`.

Axis quindi distingue, con una scelta discutibile, le implementazioni del servizio in base al `<binding>` e non al `<service>`: per generare due skeleton diversi per i servizi web `InsertPurchaseOrder` e `InsertPurchase OrderController` è necessario definire due `<binding>`.

Come già detto sopra, il servizio `Zlatan` viene implementato dall'interprete BPEL e quindi le classi lato server generate dal tool non vengono utilizzate. Vengono invece comunque utilizzati gli stub lato client, per consentire ai moduli del progetto ed ai tool di test di attivare il processo BPEL senza dover costruire il messaggio SOAP a basso livello.

Nell'ultimo listato di questo paragrafo, il 50, si riportano le definizioni necessarie per il processo BPEL di cui si è spiegato il significato teorico nel § 2.2.1.1.1.

```
<plt:partnerLinkType name="StartZlatanServiceLT">
  <plt:role          name="service">
    <plt:portType    name="impl:ZlatanServicePT" />
  </plt:role>
</plt:partnerLinkType>
<plt:partnerLinkType name="CompleterLT">
  <plt:role          name="completer">
    <plt:portType    name="impl:CompletePurchaseOrderPT" />
  </plt:role>
</plt:partnerLinkType>
<plt:partnerLinkType name="InserterLT">
  <plt:role          name="inserter">
    <plt:portType    name="impl:InsertPurchaseOrderPT" />
  </plt:role>
  <plt:role          name="requester">
    <plt:portType    name="impl:ZlatanServicePT" />
  </plt:role>
</plt:partnerLinkType>
<plt:partnerLinkType name="ResponserLT">
  <plt:role          name="responser">
    <plt:portType    name="impl:SendResponsePT" />
  </plt:role>
</plt:partnerLinkType>

<bpws:property       name="partnNo" type="xs:string"/>
```

```
<bpws:property          name="orderNo" type="xs:string"/>

<bpws:propertyAlias     propertyName="impl:partnNo"
                        messageType="impl:PurchaseOrderMessage"
                        part="PurchaseOrder"
                        query="/PartnerNumber" />
<bpws:propertyAlias     propertyName="impl:partnNo"
                        messageType="impl:SAPOutputMessage"
                        part="SAPOutput"
                        query="/PartnerNumberRef" />
<bpws:propertyAlias     propertyName="impl:orderNo"
                        messageType="impl:PurchaseOrderMessage"
                        part="PurchaseOrder"
                        query="/PurchaseOrderNumber" />
<bpws:propertyAlias     propertyName="impl:orderNo"
                        messageType="impl:SAPOutputMessage"
                        part="SAPOutput"
                        query="/PONumberRef" />
```

Listato 50: definizione delle estensioni BPEL nel documento WSDL

3.4.2.2 CompletePurchaseOrder

`CompletePurchaseOrder` è il web service che si occupa di recuperare le informazioni di default per un determinato cliente dal database dell'applicazione.

Il servizio riceve in ingresso i campi `From`, `Protocol` e `PartnerNumber` (i primi due obbligatori secondo quanto dichiarato nel WSDL) ed effettua le seguenti operazioni:

- se il `PartnerNumber` e la coppia `(From, Protocol)` sono entrambi presenti, risolve dalla tabella `SENDERS` (vedi § 3.3.2) il `PartnerNumber` associato alla coppia specificata, effettua un controllo di coerenza ed eventualmente lancia una eccezione di tipo `PartnerNumber MismatchMessage`; altrimenti risolve anche i valori di default dell'ordine di acquisto dalla tabella `DEFAULT_VALUES` e gli indirizzi a cui fornire la risposta da `REPLIES`, quindi ritorna;
- se il `PartnerNumber` non è specificato ma lo è soltanto la coppia `(From, Protocol)`, risolve il `PartnerNumber`, eventualmente lanciando una eccezione di tipo `PartnerNumberNotFoundMessage`, e recupera gli altri dati come nel caso precedente.

Il codice dell'implementazione del servizio è molto semplice ed è composto essenzialmente da controlli sui valori dei campi e da interrogazioni effettuate via *JDBC*; per questi motivi non viene riportato.

In figura 26 viene mostrato invece lo schema di collaborazione delle classi che implementano il servizio.

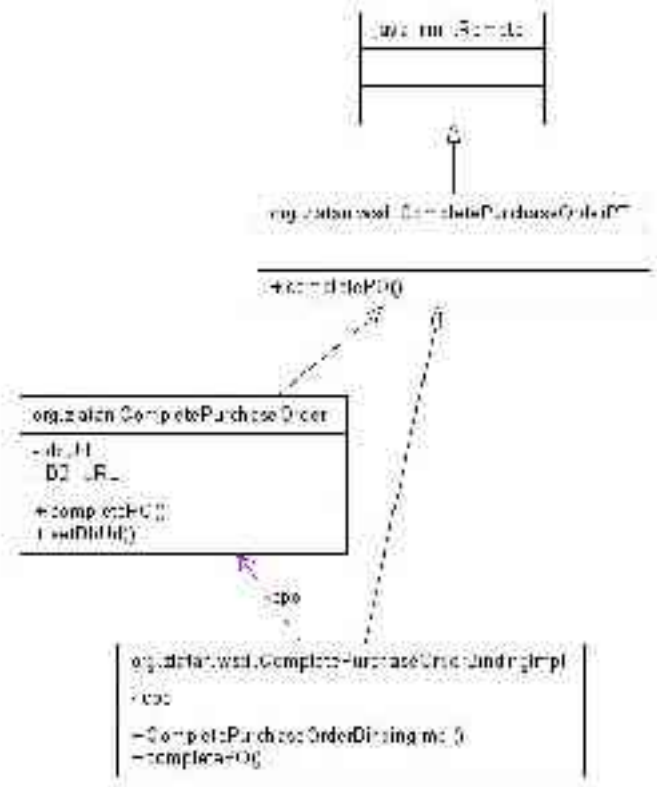

Figura 26: class diagram di CompletePurchaseOrder

Tale schema architetturale ricorre anche nelle implementazioni degli altri web service: le classi `CompletePurchaseOrderPT` e `CompletePurchaseOrderBindingImpl` sono generate automaticamente dal tool *WSDL2Java* come spiegato nel paragrafo precedente. `CompletePurchaseOrderBindingImpl` dovrebbe contenere il codice che realizza il servizio; in realtà, per questa classe viene definito un costruttore che compie le seguenti azioni:

1. legge da un file di configurazione (`completepo-ws-config.xml`) i parametri per l'inizializzazione del servizio. Per ovvi motivi di visibilità del *file system* lato server, tale file si deve trovare nella directory `WEB-INF` dell'installazione di Axis;
2. crea un oggetto di tipo `CompletePurchaseOrder` privato alla classe (`cpo` nella figura). L'oggetto viene inizializzato con i valori recuperati dal file di configurazione.

Quando viene invocato il metodo `completePO()` di `CompletePurchaseOrderBindingImpl`, che corrisponde alla `<operation>` definita nel WSDL, la chiamata viene girata all'omonimo metodo dell'oggetto `cpo`. Da notare come entrambi le classi implementino l'interfaccia del servizio web.

Nel listato 51 vengono riportati i frammenti di codice più interessanti della classe `CompletePurchaseOrderBindingImpl`.

```
public class CompletePurchaseOrderBindingImpl
implements CompletePurchaseOrderPT {

  private CompletePurchaseOrder cpo;

  public CompletePurchaseOrderBindingImpl() {

    cpo = new org.zlatan.CompletePurchaseOrder();
    org.zlatan.util.ConfigFile cf = null;
    try {
      cf = new ConfigFile("completepo-ws-config.xml", true);
      cpo.setDbUrl(cf.getValue("databaseUrl"));
    }
    catch (java.io.IOException e) { ... }
    catch (org.zlatan.InvalidDataException e) { ... }
  }

  public CompletePurchaseOrderOutputType completePO
  (CompletePurchaseOrderInputType completePurchaseOrderInput)
  throws RemoteException,
         PartnerNumberNotFoundMessage,
         PartnerNumberMismatchMessage {

    return cpo.completePO(completePurchaseOrderInput);
  }
}
```

Listato 51: codice di CompletePurchaseOrderBindingImpl

3.4.2.3 InsertPurchaseOrderController

`InsertPurchaseOrderController` è un web service che fa da *handler* per il servizio `InsertPurchaseOrder`, ovvero si occupa di intercettare le chiamate dirette a tale servizio e modificarle per renderle da esso fruibili.

Come visto nel § 2.2.3.3, le chiamate asincrone con il meccanismo della callback creano in BPEL un problema di indirizzamento. `InsertPurchaseOrderController` è stato pensato per risolvere questo problema: il processo BPEL (vedi § 3.4.3) interagisce in maniera asincrona con un `<portType>` di tipo `InsertPurchaseOrderPT`, ma il servizio contattato che lo implementa non è `InsertPurchaseOrder`, bensì l'handler stesso.

Tale handler viene portato a conoscenza, a tempo di configurazione, della URL a cui deve essere fornita la risposta asincrona. Tutte le chiamate dirette al servizio vengono deviate verso l'handler, che aggiunge al messaggio SOAP uno *header* contenente un campo `<ReplyTo>` con il valore della URL e quindi lo inoltra al servizio per l'elaborazione.

Il codice che effettua queste azioni è riportato nel listato 52. Il diagramma delle classi del servizio ha la stessa struttura evidenziata in figura 26: anche il meccanismo di inizializzazione del servizio è lo stesso.

```
private static final String NAMESPACE     =
  "http://zlatan.org/headers";
private static final String ELEMENT_NAME  = "ReplyTo";
private static final String ELEMENT_VALUE =
  "http://127.0.0.1:8082/Zlatan";
private static final String TARGET_URL    =
  "http://127.0.0.1:8081/axis/services/InsertPurchaseOrder";

private String namespace    = NAMESPACE;
private String elementName  = ELEMENT_NAME;
private String elementValue = ELEMENT_VALUE;
private String targetUrl    = TARGET_URL;

public void insertPO(PurchaseOrderType purchaseOrder)
throws RemoteException {

  try {

    InsertPurchaseOrderService inserterService =
      new InsertPurchaseOrderServiceLocator();
    InsertPurchaseOrderPT inserterPort =
      inserterService.getInsertPurchaseOrder(
      new URL(targetUrl));
    org.apache.axis.client.Stub s = (Stub) inserterPort;
    s.setHeader(namespace, elementName, elementValue);
    inserterPort.insertPO(purchaseOrder);
  }
  catch (java.net.MalformedURLException e) {
    // cut
    throw new RemoteException("...", e);
```

```
  }
  catch (javax.xml.rpc.ServiceException e) {
    // cut
    throw new RemoteException("...", e);
  }
}
```

Listato 52: codice di InsertPurchaseOrderController

3.4.2.4 InsertPurchaseOrder

`InsertPurchaseOrder` è il web service che si occupa dell'inserimento vero e proprio in SAP R/3. Le azioni che intraprende sono:

1. recupera dal *SOAP header* la URL del processo BPEL a cui inviare la risposta;
2. attiva un thread per la gestione dell'inserimento dell'ordine in SAP R/3;
3. chiude la connessione HTTP con il client.

Il thread responsabile dell'inserimento, una volta avviato, deve:

1. attivare una connessione con il server SAP;
2. convertire l'ordine di acquisto in formato BAPI;
3. eseguire la funzione di inserimento;
4. parserizzare la risposta e costruire il messaggio SOAP di ritorno;
5. inviare il messaggio alla URL trovata nel campo `<ReplyTo>` dello header.

Il thread attivato in realtà corrisponde alla stessa istanza della classe `InsertPurchaseOrder` che ha elaborato il messaggio in ingresso: tale classe infatti implementa sia `InsertPurchaseOrderPT`, secondo lo schema già visto, sia l'interfaccia `Runnable`.

Il metodo `run()` del thread utilizza le classi del package `org.zlatan.sap`, contenente una collezione di classi, tra cui:

- due bean utili per la conversione da formato XML a formato BAPI;
- una classe `ConnectionUtility` (implementata solo parzialmente) per la gestione delle connessioni al server SAP;
- due eccezioni, `MaterialCodeNotFoundException` e `SAPGeneralError Exception`, che corrispondono alle `<operation>` `materialCodeError` e `sapGeneralError` presenti nel `<portType>` esposto dal processo BPEL (vedi § 3.4.2.1);
- una classe `SalesOrder` che funge da bean per la rappresentazione SAP R/3 dell'ordine d'acquisto ed allo stesso tempo come proxy (tramite il metodo `executeCreation()`, vedi anche § 2.3.3.3) per l'apposita funzione BAPI addetta alla creazione dell'ordine nel sistema (`BAPI_SALESORDER_CREATEFROMDAT2`).

Nel listato 53 sono riportati i due metodi più importanti della classe `InsertPurchaseOrder`: `run()` ed `insertPO()`. Notare come in `run()` vengano utilizzate le eccezioni per distinguere l'operazione da chiamare sul `<portType>` di `ZlatanService`, secondo il meccanismo di chiamata asincrona con callback e `<pick>` illustrato in precedenza.

```
// cut
public class InsertPurchaseOrder
implements Runnable, InsertPurchaseOrderPT {

  // cut
  private String namespace         = NAMESPACE;
  private String elementName       = ELEMENT_NAME;
  private String jcoClientClient   = JCO_CLIENT_CLIENT;
  private String jcoClientUser     = JCO_CLIENT_USER;
  private String jcoClientPassword = JCO_CLIENT_PASSWORD;
  private String jcoClientAshost   = JCO_CLIENT_ASHOST;
  private String jcoSysnr          = JCO_SYSNR;
  private String poolName          = POOL_NAME;

  private PurchaseOrderType purchaseOrder;
  private String replyTo;
  private JCO.Pool pool;
  private JCO.Client connection;

  // cut
  public void insertPO(PurchaseOrderType purchaseOrder)
  throws RemoteException {

    this.purchaseOrder =
      PurchaseOrderReplicator.copy(purchaseOrder);

    // recupera la URL per la risposta

    org.apache.axis.MessageContext ctx =
      MessageContext.getCurrentContext();
    org.apache.axis.message.SOAPEnvelope env =
      ctx.getRequestMessage().getSOAPEnvelope();
    org.apache.axis.message.SOAPHeaderElement
      soapHeaderElement =
        env.getHeaderByName(namespace, elementName);
    this.replyTo = (String)
      soapHeaderElement.getObjectValue();

    // attiva il thread che esegue l'inserimento

    ThreadGroup taskGroup = new ThreadGroup("InsertPO Group");
    Thread inserter = new Thread(taskGroup, this);
    inserter.start();
  }

  public void run() {

    ZlatanService zlatanService = null;
    ZlatanServicePT zlatanPort = null;
    SAPOutputType sapOut = null;
```

```
try {
  try {

    // utilizza lo stub per accedere a ZlatanService

    zlatanService = new ZlatanServiceLocator();
    zlatanPort = zlatanService.getZlatan(
      new URL(replyTo));
    sapOut = new SAPOutputType();

    // prepara i campi della risposta che già conosce

    sapOut.setPONumberRef(
      purchaseOrder.getPurchaseOrderNumber());
    sapOut.setPartnerNumberRef(
      purchaseOrder.getPartnerNumber());

  } catch (Exception e)  {
    throw new SAPGeneralErrorException(e.getMessage());
  }

  // connessione a SAP

  JCO.Client connection =
    ConnectionUtility.getPooledClient(POOL_NAME,
      this.getConnectionProps());

  // creazione ed inizializzazione della classe proxy

  SalesOrder so = new SalesOrder(connection);
  so.setPartnerRole(
    purchaseOrder.getPartnerRole().getValue());
  so.setPartnerNumber(purchaseOrder.getPartnerNumber());
  // cut
  so.setPurchaseOrderNumber(
    purchaseOrder.getPurchaseOrderNumber());

  Vector materials = new Vector();
  int l =  purchaseOrder.getMaterials().
    getMaterial().length;
  for (int i = 0; i < l; i++) {
    Material mat = new Material( ... );
    materials.add(mat);
  }
  so.setMaterials(materials);

  // esecuzione della funzione BAPI

  Vector returned  = so.executeCreation();
  String sapOutMsg = new String("");
  Enumeration e    = null;
```

```
      for (e = returned.elements(); e.hasMoreElements();) {

        ReturnMessage rm    = (ReturnMessage) e.nextElement();
        int type            = rm.getType();
        String text         = new String(rm.getText());
        String sType  = ReturnMessage.typeIntToString(type);
        sapOutMsg          += sType + ": " + text + " -";

        if ((type == ReturnMessage.ERROR) &&
           (text.indexOf("Material") > 0) &&
           (text.indexOf("is not defined") > 0))
             throw new MaterialCodeNotFoundException(text);

        if (type == ReturnMessage.ERROR)
           throw new SAPGeneralErrorException(text);
      }

      // se non ci sono errori, il messaggio va alla callback

      sapOut.setMessage(sapOutMsg);
      zlatanPort.insertPOCallback(sapOut);
    }

    // se ci sono stati errori, invocare le relative operation

    catch (MaterialCodeNotFoundException e) {
      sapOut.setMessage(e.getMessage());
      try {
        zlatanPort.materialCodeError(sapOut);
      } catch (java.rmi.RemoteException r) {}
    }

    catch (SAPGeneralErrorException e) {
      sapOut.setMessage(e.getMessage());
      try {
        zlatanPort.sapGeneralError(sapOut);
      } catch (java.rmi.RemoteException r) {}
    }

    catch (JCO.Exception e) { ... }
    catch (RemoteException e) { ... }
    finally {
      try {
        ConnectionUtility.closePooledClient(connection);
      } catch (Exception e) {}
    }
  }
// cut
}
```

Listato 53: codice di InsertPurchaseOrder

3.4.2.5 SendResponse

L'ultimo web service implementato è anche l'ultimo tra quelli che compongono il processo. Si tratta di `SendResponse`, il servizio che si occupa semplicemente di inviare il messaggio fornito da SAP R/3 agli indirizzi ai quali il cliente che ha inviato l'ordine ha richiesto la notifica. Come si nota dalla descrizione contenuta nel WSDL (vedi § 3.4.2.1), si tratta di un servizio di tipo *one-way*.

I protocolli implementati in uscita sono HTTP, SMTP e SOAP. Nel caso HTTP, viene effettuata una `POST` contenente tre parametri: `action` (settato ad `insert`), `number` (contenente il numero d'ordine d'acquisto di riferimento) e `message` (la stringa con il messaggio). Il messaggio HTTP ha quindi un formato simile a quello di listato 54.

```
POST /zlatan/control2.jsp HTTP/1.1
User-Agent: Java/1.4.2_04
Host: 127.0.0.1
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 256

action=insert&number=PO-1234567890-
688&message=S+SALES_HEADER_IN+has+been+processed+successfully+-
+E+Material+R-1060+is+not+defined+for+sales+org.1000%
2C++distr.chan.10%2C++language+DE+-
+E+Error+in+SALES_ITEM_IN+000010+-
+E+Sales+document++was+not+changed
```

Listato 54: esempio di post HTTP effettuato da SendResponse

Nel caso SMTP, lo stesso messaggio viene formattato in maniera leggibile dagli umani come in listato 55 (sono stati cancellati i campi contenenti URL, indirizzi IP ed email).

```
From - Sat Oct 30 16:07:12 2004
[cut]
Message-ID: <5663550.1099145211202.JavaMail.Tesi@pavel>
From: zlatan@tele2.it
To: XXXXXXXXXXXXXXXX@tele2.it
To: XXXXXXXXXXXXXXXX@tele2.it
Subject: Conferma ricezione ordine da Zlatan
Mime-Version: 1.0
Content-Type: text/plain; charset=Cp1252
Content-Transfer-Encoding: quoted-printable
Date: Sat, 30 Oct 2004 16:06:53 +0200

Il Suo ordine numero PO-1234567890-881 =E8 stato processato dal
sistema ed ha prodotto il seguente messaggio:

S SALES_HEADER_IN has been processed successfully - E Material
R-1060 is not defined for sales org.1000,  distr.chan.10,
language DE - E Error in SALES_ITEM_IN 000010 - E Sales
document  was not changed
```

Listato 55: esempio di messaggio email inviato da SendResponse

Il formato SOAP viene infine riportato in listato 56.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <ns1:SAPOutput xmlns:ns1="http://wsdl.zlatan.org">
      <PONumberRef>PO-1234567890-688</PONumberRef>
      <PartnerNumberRef>PARTN001</PartnerNumberRef>
      <Message>

S SALES_HEADER_IN has been processed successfully - E Material
R-1060 is not defined for sales org.1000,   distr.chan.10,
language DE - E Error in SALES_ITEM_IN 000010 - E Sales
document  was not changed

      </Message>
    </ns1:SAPOutput>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listato 56: esempio di messaggio SOAP inviato da SendResponse

Come già per i web service precedenti, sia il sistema di inizializzazione che lo schema delle classi ricalcano quelli di `CompletePurchaseOrder`. Il web service utilizza inoltre le librerie *JavaMail* per l'invio delle email, la classe `java.net.HttpURLConnection` per il post HTTP e *SAAJ* per la costruzione a basso livello del messaggio SOAP.

3.4.3 Il documento BPEL

La descrizione del processo BPEL è la parte più significativa dell'intero sistema. Come già detto più volte, in particolare nel § 2.2.3, la reale portabilità del codice BPEL è limitata fortemente da problemi ancora aperti che riguardano le specifiche del linguaggio e le implementazioni degli interpreti. La versione del documento riportata è ottimizzata per il funzionamento con l'interprete *GlobalBiz Process Engine*, le cui caratteristiche sono già state elencate nel § 2.2.4.1.

Nell'analisi si procederà riportando e commentando gruppi di elementi omogenei, come per il documento WSDL nel § 3.4.2.1.

Il primo listato riporta l'elemento *root* del documento, `<process>`, con la definizione dei vari namespace utilizzati.

```
<process name="zlatan"
  targetNamespace="http://bpel.zlatan.org"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:tns="http://bpel.zlatan.org"
  xmlns:pos="http://wsdl.zlatan.org"
  xmlns:bpelext="urn:DDABPE">
```

Listato 57: definizione dei namespace nel documento BPEL

Il namespace `pos`, coerentemente, fa riferimento alla URL definita come `targetNamespace` del documento WSDL, mentre il namespace `bpelext` è quello utilizzato dal PE per le estensioni al linguaggio.

Come già visto nel § 2.2.1.1, i primi elementi ad essere definiti nel corpo di un processo sono i `<partnerLink>`, le variabili ed i `<correlationSet>` con visibilità globale.

```
<partnerLinks>
  <partnerLink  name="client"
                partnerLinkType="pos:ZlatanServiceLT"
                myRole="service" bpelext:myPort="pos:StartZlatan" />
  <partnerLink  name="completer"
                partnerLinkType="pos:CompleterLT"
                partnerRole="completer"
                bpelext:partnerPort="pos:CompletePurchaseOrder" />
  <partnerLink  name="inserter"
                partnerLinkType="pos:InserterLT"
                partnerRole="inserter"
                bpelext:partnerPort=
                  "pos:InsertPurchaseOrderController"
                myRole="requester" bpelext:myPort="pos:Zlatan" />
  <partnerLink  name="responser"
                partnerLinkType="pos:ResponserLT"
                partnerRole="responser"
                bpelext:partnerPort="pos:SendResponse" />
</partnerLinks>

<variables>
  <variable   name="PO"
```

```
                messageType="pos:PurchaseOrderMessage" />
    <variable   name="PO2"
                messageType="pos:PurchaseOrderMessage" />
    <variable   name="insertOutput"
                messageType="pos:SAPOutputMessage" />
    <variable   name="completeInput"
                messageType="pos:CompletePurchaseOrderInputMessage" />
    <variable   name="completeOutput"
                messageType="pos:CompletePurchaseOrderOutputMessage" />
    <variable   name="responseInput"
                messageType="pos:SendResponseInputMessage" />
    <variable   name="pnoNotFound"
                messageType="pos:PartnerNumberNotFoundMessage" />
    <variable   name="pnoMismatch"
                messageType="pos:PartnerNumberMismatchMessage" />
    <variable   name="tempString"
                messageType="pos:DummyMessage" />
</variables>

<correlationSets>
    <correlationSet  name="POCorrelationSet"
                     properties="pos:partnNo pos:orderNo" />
</correlationSets>
```

Listato 58: definizione dei partnerLink, delle variabili e dei correlationSet nel documento BPEL

La presenza degli attributi `bpelext:myPort` e `bpelext:partnerPort` risolve il problema dell'indirizzamento dei messaggi SOAP ed è già stata analizzata nel § 2.2.3.1. L'unico `<partnerLink>` in cui sono coinvolti due `<portType>` è quello per l'inserimento dell'ordine d'acquisto in modalità asincrona, come si evince anche dalla figura 18 di pagina 72. Altri due `<partnerLink>` riguardano l'invocazione (rispettivamente, sincrona e *one-way*) dei servizi `CompletePurchaseOrder` e `SendResponse`, mentre il primo `<partnerLink>` modella l'interazione tra client che attiva il processo e interprete BPEL.

Viene definita una variabile per ogni tipo di messaggio presente nel WSDL, con l'eccezione di `PO` e `PO2`, entrambe di tipo `PurchaseOrderMessage`. Questo è necessario perché, all'attivazione del processo, l'ordine d'acquisto ricevuto non è completo in tutti i suoi campi, e quindi la variabile `PO` viene istanziata soltanto in parte. Il completamento avviene in seguito, ma poiché nella sua versione attuale PE ha bisogno di conoscere la forma completa di una variabile quando viene inizializzata, l'aggiunta di nuovi elementi può creare conflitti. Per questo, verrà istanziata una nuova variabile vuota, `PO2`, nella quale verranno copiati i valori presenti in `PO` integrati con quelli ritornati da `CompletePurchaseOrder`. In questo meccanismo rientra anche la presenza della variabile `tempString`, il cui tipo è il `DummyMessage` a cui si era accennato nel § 3.4.2.1.

L'unico `<correlationSet>` definito è quello che permette di correlare i messaggi di chiamata e risposta al servizio asincrono `InsertPurchaseOrder`. Le proprietà coinvolte nella correlazione sono l'ID del cliente che ha inviato il messaggio ed il numero d'ordine interno al sistema del cliente. Le proprietà sono mappate sugli elementi `PartnerNumber` e `PurchaseOrderNumber` nel messaggio SOAP di chiamata

e sugli elementi `PartnerNumberRef` e `PONumberRef` nel messaggio SOAP di ritorno (vedi anche la definizione delle proprietà nel § 3.4.2.1).

Il processo prosegue dopo queste definizioni con la descrizione della sequenza delle attività da intraprendere (listato 59).

```
<sequence> <!-- begin sequence 0 -->

  <receive  name="begin"
            partnerLink="client"
            operation="startZlatan"
            variable="PO"
            createInstance="yes" />
```

Listato 59: inizio del processo BPEL

La prima azione del processo è ovviamente la ricezione del messaggio contenente l'ordine d'acquisto da un `client`. Notare la presenza dell'attributo `createInstance` che indica al PE di generare una nuova istanza del processo. Dopo l'attivazione, il processo entra in uno `<scope>`.

```
<scope><!-- begin scope 0 -->
  <faultHandlers>
    <catch faultName="pos:PartnerNumberNotFoundFault"
           faultVariable="pnoNotFound">
      <terminate />
    </catch>
    <catch faultName="pos:PartnerNumberMismatchFault"
           faultVariable="pnoMismatch">
      <terminate />
    </catch>
  </faultHandlers>
```

Listato 60: gestori di fault del primo scope del processo BPEL

Le eccezioni gestite in questo `<scope>` sono quelle lanciate dall'unico web service invocato al suo interno, ovvero `CompletePurchaseOrder`. Lo `<scope>` infatti contiene una sequenza di `<assign>` dedicati alla preparazione della variabile di ingresso del servizio e l'invocazione sincrona del servizio stesso.

```
<sequence>  <!-- begin sequence 1 -->
  <assign     name="completeDummy1">
    <copy>
      <from  expression=
               "'&lt;From/&gt;&lt;Protocol/&gt;&lt;PartnerNumber/&gt;'" />
      <to    part="string" variable="tempString" />
    </copy>
  </assign>
  <assign     name="completeDummy2">
    <copy>
      <from  expression="partToDOM( 'tempString', 'string' )" />
      <to    part="CompletePurchaseOrderInput"
             variable="completeInput" />
    </copy>
  </assign>
```

```
  <assign     name="complete1">
    <copy>
      <from  variable="PO" part="PurchaseOrder" query="/From" />
      <to    variable="completeInput"
             part="CompletePurchaseOrderInput" query="/From" />
    </copy>
  </assign>
  <assign     name="complete2">
    <copy>
      <from  variable="PO" part="PurchaseOrder" query="/Protocol" />
      <to    variable="completeInput"
             part="CompletePurchaseOrderInput" query="/Protocol" />
    </copy>
  </assign>
  <assign     name="complete3">
    <copy>
      <from  variable="PO"
             part="PurchaseOrder"
             query="/PartnerNumber" />
      <to    variable="completeInput"
             part="CompletePurchaseOrderInput"
             query="/PartnerNumber" />
    </copy>
  </assign>

  <invoke     name="invokeCompleter"
              partnerLink="completer"
              operation="completePO"
              inputVariable="completeInput"
              outputVariable="completeOutput" />

</sequence> <!-- end sequence  1 -->
</scope>    <!-- end scope     0 -->
```

Listato 61: invocazione del primo servizio nel processo BPEL

Le `<assign>` `completeDummy1` e `completeDummy2` servono per la segnalare a PE l'inizializzazione della variabile, mentre le tre `<assign>` successive sono quelle che effettivamente copiano i valori degli elementi da `PO` a `completeInput`.

Lo `<scope>` seguente inizia con la creazione della variabile `PO2` e con l'inizializzazione in base ai valori presenti in `PO` e ritornati da `CompletePurchaseOrder` in `completeOutput`. Per questo blocco non sono presenti gestori di fault.

```
<scope><!-- begin scope 1 -->
<sequence>  <!-- begin sequence 2 -->

  <assign     name="insertDummy1">
    <copy>
      <from  expression="'&lt;From/>&lt;Protocol/>&lt;DocumentType/>
             &lt;SalesOrganization/>&lt;DistributionChannel/>
             &lt;Division/>&lt;PartnerRole/>&lt;PartnerNumber/>
```

```
            &lt;PurchaseOrderNumber/>&lt;Materials/>'" />
      <to    part="string" variable="tempString" />
    </copy>
  </assign>
  <assign     name="insertDummy2">
    <copy>
      <from  expression="partToDOM('tempString', 'string')" />
      <to    part="PurchaseOrder" variable="PO2" />
    </copy>
  </assign>
  <assign     name="insert0">
    <copy>
      <from  variable="PO" part="PurchaseOrder" query="/From" />
      <to    variable="PO2" part="PurchaseOrder"    query="/From" />
    </copy>
  </assign>
  <assign     name="insert1">
    <copy>
      <from  variable="PO" part="PurchaseOrder" query="/Protocol" />
      <to    variable="PO2"
             part="PurchaseOrder" query="/Protocol" />
    </copy>
  </assign>
  <assign     name="insert2">
    <copy>
      <from  variable="PO" part="PurchaseOrder"
             query="/PurchaseOrderNumber" />
      <to    variable="PO2" part="PurchaseOrder"
             query="/PurchaseOrderNumber" />
    </copy>
  </assign>
  <assign     name="insert3">
    <copy>
      <from  variable="completeOutput"
             part="CompletePurchaseOrderOutput"
             query="/PartnerNumber" />
      <to    variable="PO2" part="PurchaseOrder"
             query="/PartnerNumber" />
    </copy>
  </assign>
  <assign     name="insert4">
    <copy>
      <from  variable="completeOutput"
             part="CompletePurchaseOrderOutput"
             query="/DocumentType" />
      <to    variable="PO2"
             part="PurchaseOrder"
             query="/DocumentType" />
    </copy>
  </assign>
```

```
<assign     name="insert5">
  <copy>
    <from  variable="completeOutput"
           part="CompletePurchaseOrderOutput"
           query="/SalesOrganization" />
    <to    variable="PO2" part="PurchaseOrder"
           query="/SalesOrganization" />
  </copy>
</assign>
<assign     name="insert6">
  <copy>
    <from  variable="completeOutput"
           part="CompletePurchaseOrderOutput"
           query="/DistributionChannel" />
    <to    variable="PO2" part="PurchaseOrder"
           query="/DistributionChannel" />
  </copy>
</assign>
<assign     name="insert7">
  <copy>
    <from  variable="completeOutput"
           part="CompletePurchaseOrderOutput"
           query="/Division" />
    <to    variable="PO2"
           part="PurchaseOrder"
           query="/Division" />
  </copy>
</assign>
<assign     name="insert8">
  <copy>
    <from  variable="completeOutput"
           part="CompletePurchaseOrderOutput"
           query="/PartnerRole" />
    <to    variable="PO2"
           part="PurchaseOrder"
           query="/PartnerRole" />
  </copy>
</assign>
<assign     name="insert9">
  <copy>
    <from  variable="PO"
           part="PurchaseOrder"
           query="/Materials" />
    <to    variable="PO2"
           part="PurchaseOrder"
           query="/Materials" />
  </copy>
</assign>
```

Listato 62: preparazione della variabile di ingresso del secondo servizio nel processo BPEL

La chiamata al web service che inserisce l'ordine di acquisto, riportata in listato 63, è asincrona e seguita da una <pick> strutturata secondo il modello di chiamata asincrona con gestione degli errori (vedi § 2.2.3.2 e § 3.4.2.1). La chiamata asincrona istanzia il <correlationSet>, che viene usato in ricezione da tutti i blocchi <onMessage>. Tali blocchi hanno tutti il corpo <empty>: la gestione dell'errore non è differenziata ma usata solo a scopo dimostrativo.

```
<invoke   name="invokeAsyncInserter"
          partnerLink="inserter"
          operation="insertPO"
          inputVariable="PO2">
  <correlations>
    <correlation  set="POCorrelationSet"
                  initiate="yes"
                  pattern="out" />
  </correlations>
</invoke>

<pick name="pickAsyncInserter">

  <onMessage  operation="insertPOCallback"
              variable="insertOutput"
              partnerLink="inserter">
    <correlations>
      <correlation set="POCorrelationSet" initiate="no" />
    </correlations>
    <empty />
  </onMessage>

  <onMessage  operation="materialCodeError"
              variable="insertOutput"
              partnerLink="inserter">
    <correlations>
      <correlation set="POCorrelationSet" initiate="no" />
    </correlations>
    <empty />
  </onMessage>

  <onMessage  operation="sapGeneralError"
              variable="insertOutput"
              partnerLink="inserter">
    <correlations>
      <correlation set="POCorrelationSet" initiate="no" />
    </correlations>
    <empty />
  </onMessage>

</pick>
```

Listato 63: invocazione del secondo servizio nel processo BPEL

Il listato 64 presenta infine la chiamata *one-way* al servizio che invia la risposta al cliente, preceduta naturalmente dall'inizializzazione della variabile come già visto nei listati precedenti, e seguita dalla chiusura della `<sequence>`, dello `<scope>` e del processo.

Notare in particolar modo la potenzialità del costrutto `<assign>` usato insieme a *XPath* in `response2`, dove una intera `<part>` di un messaggio viene copiata in un sotto-elemento di una `<part>` di un `<message>` diverso.

```
<assign     name="responseDummy1">
  <copy>
    <from  expression="'&lt;Replies/&gt;&lt;SAPOutput/&gt;'" />
    <to    part="string" variable="tempString" />
  </copy>
</assign>
<assign     name="responseDummy2">
  <copy>
    <from  expression="partToDOM('tempString', 'string')" />
    <to    part="SendResponseInput" variable="responseInput" />
  </copy>
</assign>
<assign     name="response1">
  <copy>
    <from  variable="completeOutput"
           part="CompletePurchaseOrderOutput"
           query="/Replies" />
    <to    variable="responseInput"
           part="SendResponseInput"
           query="/Replies" />
  </copy>
</assign>
<assign     name="response2">
  <copy>
    <from  variable="insertOutput" part="SAPOutput" />
    <to    variable="responseInput" part="SendResponseInput"
           query="/SAPOutput" />
  </copy>
</assign>

<invoke     name="invokeResponser"
            partnerLink="responser"
            operation="sendResponse"
            inputVariable="responseInput" />

</sequence>   <!-- end sequence  2 -->
</scope>      <!-- end scope     1 -->
</sequence>   <!-- end sequence  0 -->
</process>
```

Listato 64: invocazione del terzo servizio nel processo BPEL

3.4.4 L'interfaccia web

L'interfaccia utente di *Zlatan* è implementata sotto forma di web application. Le azioni che possono essere effettuate tramite l'interfaccia sono:

- invio di un ordine di acquisto in formato SOAP direttamente al PE di *GlobalBiz*, per le installazioni di Zlatan dove non è disponibile IBRA;
- invio di un ordine di acquisto via HTTP al `HttpController` di IBRA;
- consultazione via web dei messaggi prodotti da SAP dopo l'inserimento, nel caso in cui `SendResponse` abbia prodotto una risposta HTTP;
- gestione delle tabelle accedute da `CompletePurchaseOrder`.

In figura 27 e successive sono riportate alcune schermate di esempio dell'interfaccia.

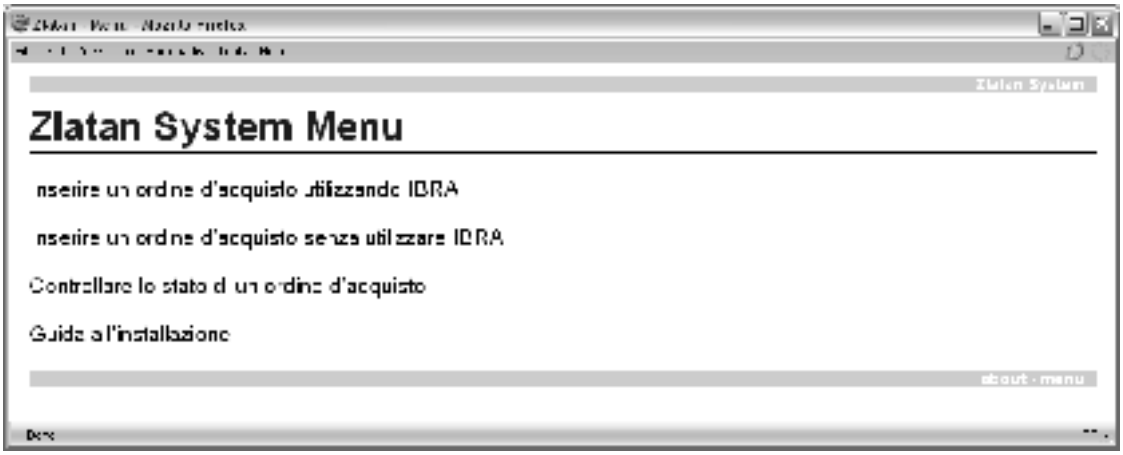

Figura 27: menu dell'interfaccia grafica di Zlatan

Figura 28: pagina per l'inserimento di un ordine d'acquisto via IBRA

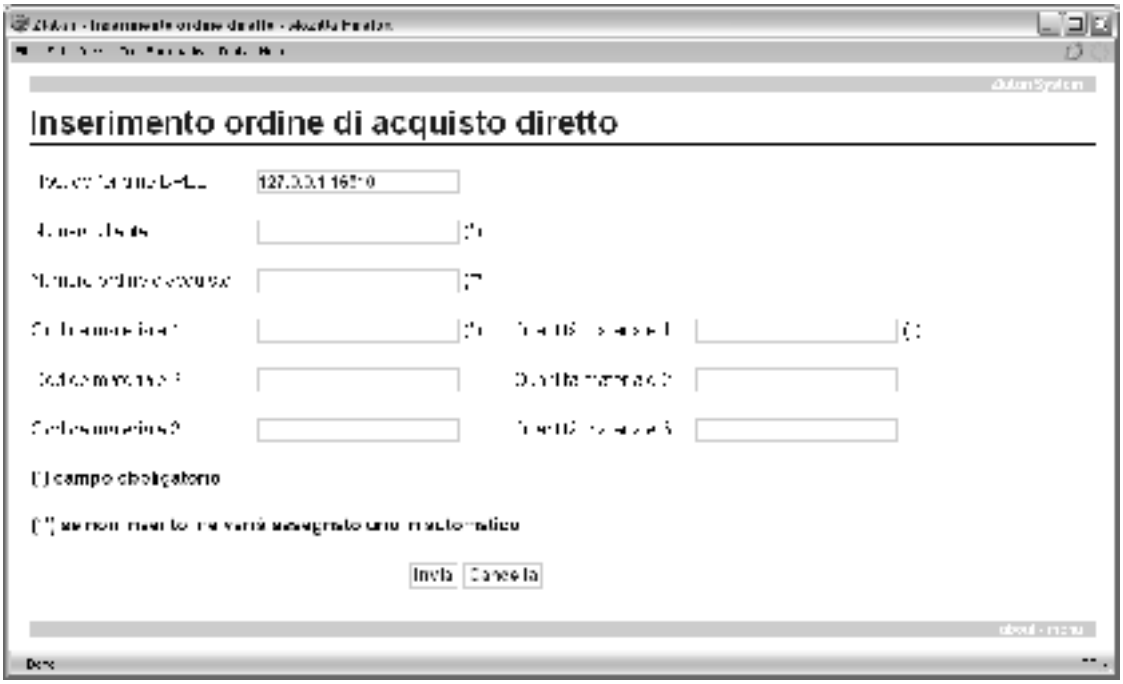

Figura 29: pagina per l'invio di un messaggio SOAP diretto all'engine BPEL

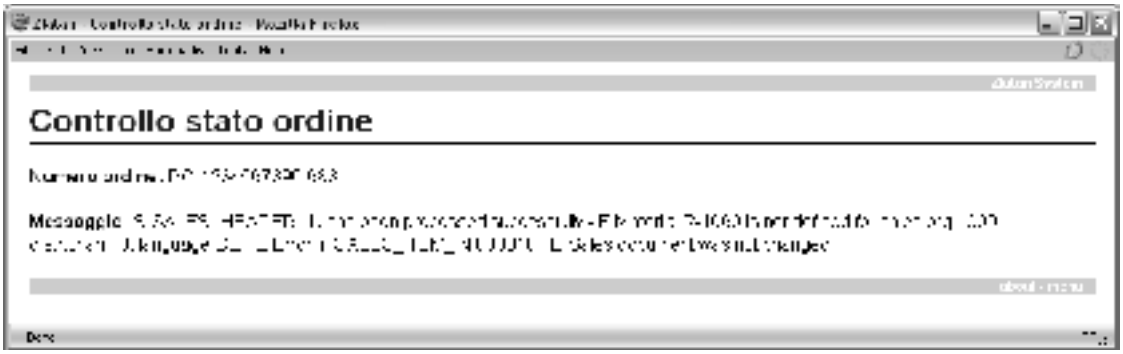

Figura 30: risultato dell'interrogazione sullo stato di un ordine

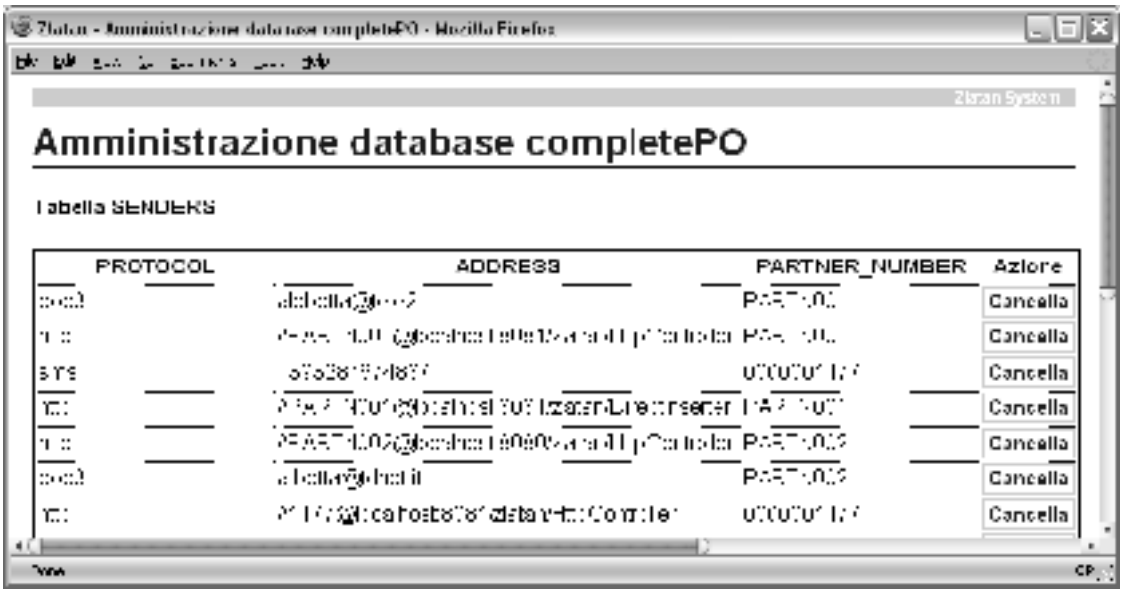

Figura 31: pagina per la gestione del database di CompletePurchaseOrder

3.4.5 Esempi d'uso

In questo paragrafo vengono proposti due esempi d'uso completi del sistema, corredati dalle schermate generate dai vari moduli e da alcuni messaggi scambiati.

Nel primo esempio, il cliente numero 1177 invia un ordine d'acquisto via web e riceve il messaggio di risposta via posta elettronica; nel secondo, un cliente tenta di inviare un ordine via SMS da un numero di cellulare non autorizzato e quindi non riconosciuto come valido dal sistema, che blocca il processo di inserimento dell'ordine.

3.4.5.1 Successo con invio via web e risposta via posta elettronica

Il primo esempio mostra l'utilizzo della web application illustrata nel § 3.4.4 per l'invio dell'ordine di acquisto. Il driver di IBRA coinvolto in questo esempio è `HttpController`.

Nelle figure 32 e 33 sono riportate le schermate della web application compilate dal cliente.

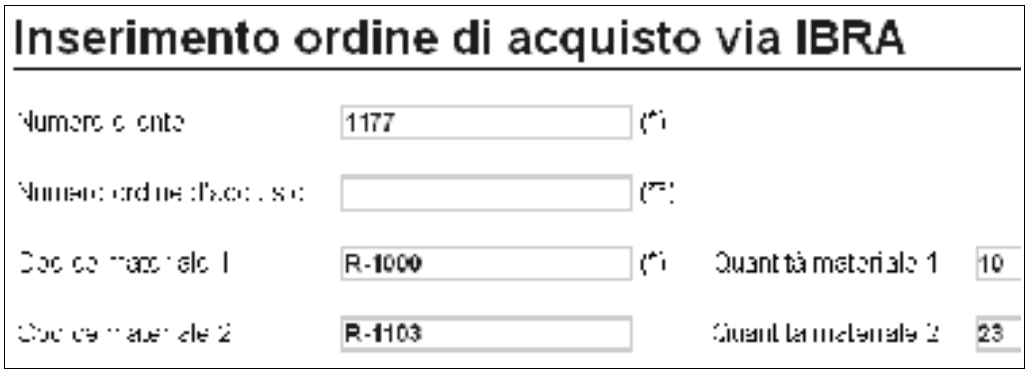

Figura 32: dati inseriti dal partner 1177 all'invio dell'ordine

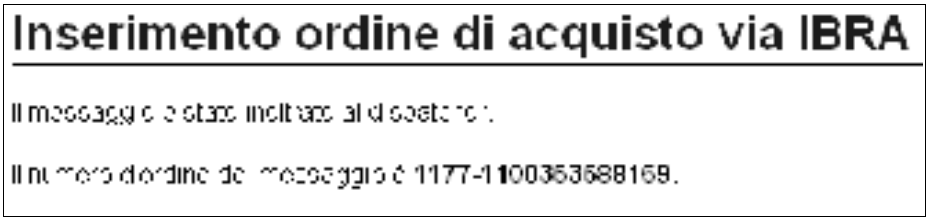

Figura 33: risposta fornita dalla web application al partner 1177

Il messaggio HTTP contenente i dati dell'ordine viene postato presso `HttpController` (listato 65), il quale provvede a trasferire il messaggio via JMS al `PrepareRequestDispatcher` e quindi ad uno dei `PrepareRequest Module` attivi (vedi figure 34 e 35).

```
POST /zlatan/HttpController HTTP/1.1
Host: 127.0.0.1
[cut]
Content-Length: 91

partnerNumber=1177&PONumber=&mat1Code=R-
1000&mat1Qty=10&mat2Code=R-1103&mat2Qty=23&matNum=2
```

Listato 65: messaggio postato presso HttpController dal partner 1177

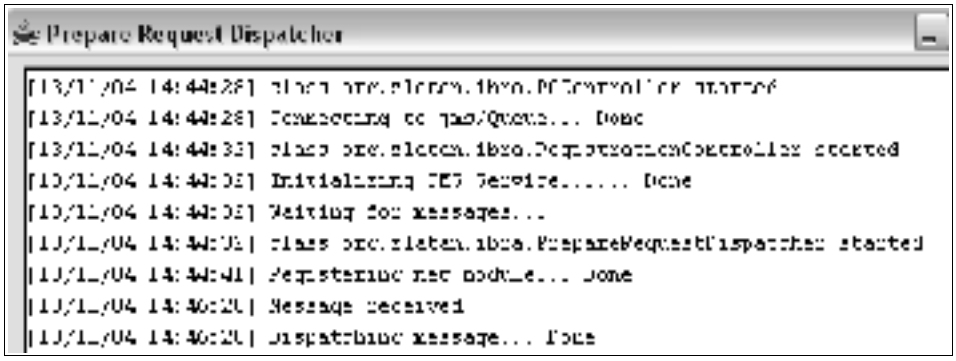

Figura 34: messaggi generati dal dispatcher alla ricezione dell'ordine dal partner 1177

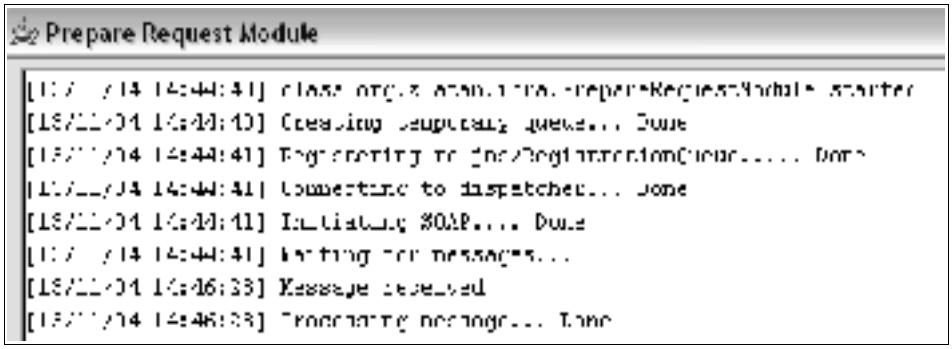

Figura 35: messaggi generati da un modulo alla ricezione dell'ordine dal dispatcher

Il modulo prepara quindi un messaggio SOAP come in listato 66 con il quale attiva il processo BPEL. Il nuovo processo viene mostrato in cima alla lista dei processi recenti nel pannello di amministrazione del *Process Engine*, da cui è accessibile anche la rappresentazione grafica in linea (vedi figure 36 e 39). Il colore del processo è azzurro poiché si tratta di un processo attivo (i processi terminati a buon fine sono verdi, i processi falliti arancio).

```
POST /StartZlatan HTTP/1.0
Content-Type: text/xml; charset=utf-8
[cut]
SOAPAction: ""
Content-Length: 777

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope ...>
     <soapenv:Body>
       <startZlatan xmlns="http://zlatan.org">
         <PurchaseOrder xmlns="">
           <From>//1177@127.0.0.1/zlatan/HttpController</From>
           <Protocol>http</Protocol>
           <PartnerNumber>1177</PartnerNumber>
           <PurchaseOrderNumber>
             1177-1100353588169
           </PurchaseOrderNumber>
           <Materials>
             <Material>
               <Code>R-1000</Code>
               <Quantity>10</Quantity>
```

```
            </Material>
            <Material>
              <Code>R-1103</Code>
              <Quantity>23</Quantity>
            </Material>
          </Materials>
        </PurchaseOrder>
      </startZlatan>
    </soapenv:Body>
  </soapenv:Envelope>
```

Listato 66: messaggio SOAP generato dal modulo per l'attivazione del processo BPEL

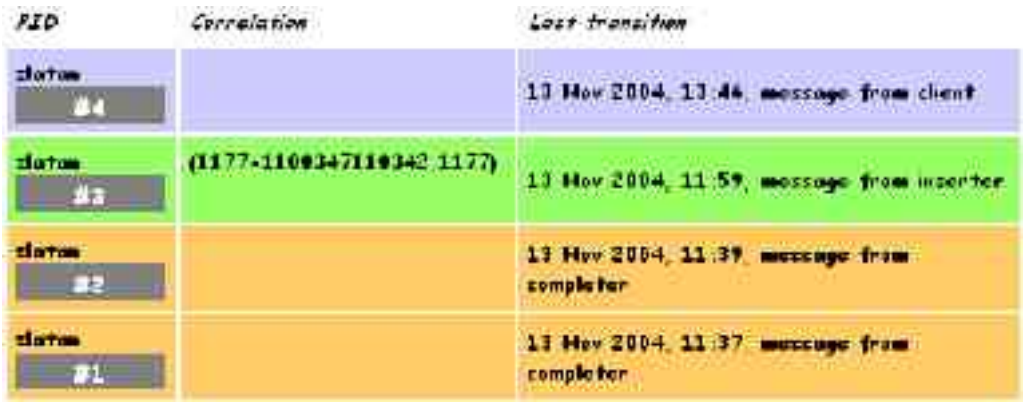

Figura 36: lista dei processi recenti mostrata dal PE all'avvio del nuovo processo

L'interprete BPEL invoca i vari servizi che compongono il processo: i messaggi SOAP relativi al primo servizio invocato sono riportati come esempio nei listati 67 e 68. Notare come, con l'avanzamento del processo, nuove informazioni vengano rese disponibili nel pannello di amministrazione e nella rappresentazione grafica generata. In particolare, in figura 37, catturata nel mezzo della chiamata asincrona al servizio `InsertPurchaseOrder`, si può notare la presenza del correlation set appena istanziato dalla `<invoke>`.

```
<env:Envelope ...>
    <env:Body ...>
      <ddabpe1:completePO xmlns:ddabpe1="http://zlatan.org">
        <CompletePurchaseOrderInput>
          <From>//1177@127.0.0.1/zlatan/HttpController</From>
          <Protocol>http</Protocol>
          <PartnerNumber>1177</PartnerNumber>
        </CompletePurchaseOrderInput>
      </ddabpe1:completePO>
    </env:Body>
</env:Envelope>
```

Listato 67: messaggio SOAP inviato dal PE al servizio CompletePurchaseOrder

```
<soapenv:Envelope ...>
  <soapenv:Body>
    <completePOResponse xmlns="http://zlatan.org">
      <CompletePurchaseOrderOutput xmlns="">
```

```
          <PartnerNumber>1177</PartnerNumber>
          <DocumentType>TA</DocumentType>
          <SalesOrganization>1000</SalesOrganization>
          <DistributionChannel>10</DistributionChannel>
          <Division>00</Division>
          <PartnerRole>AG</PartnerRole>
          <Replies>
            <ReplyTo>
              <Address>alebotta@tele2.it</Address>
              <Protocol>smtp</Protocol>
            </ReplyTo>
          </Replies>
        </CompletePurchaseOrderOutput>
      </completePOResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Listato 68: risposta fornita al PE da CompletePurchaseOrder per il partner 1177

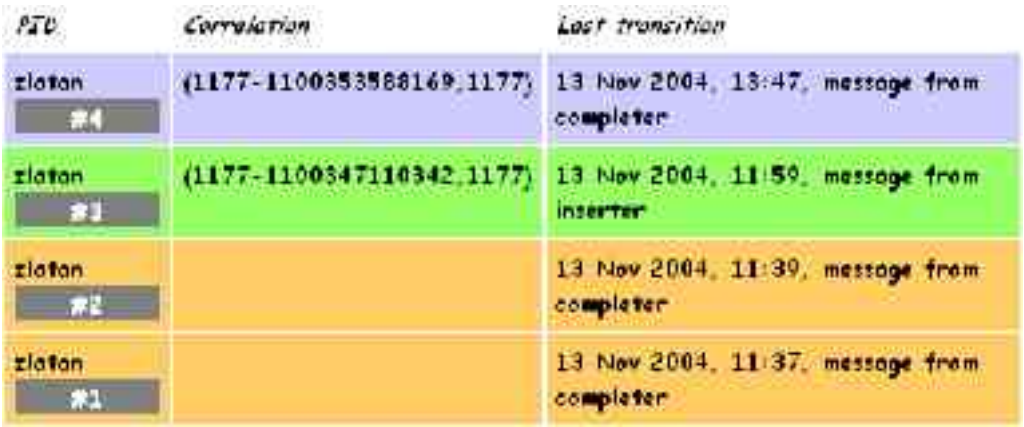

Figura 37: lista dei processi recenti mostrata dal PE dopo la chiamata a InsertPurchaseOrder

L'ultimo servizio invia indietro al partner 1177 una email contenente i dettagli sull'ordine: una volta terminato il processo, nel pannello di amministrazione la riga relativa al processo viene evidenziata in verde come in figura 38.

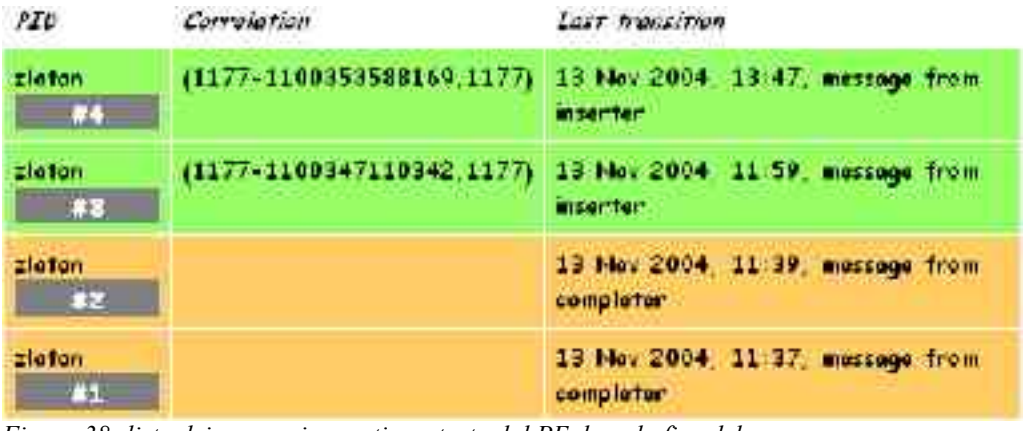

Figura 38: lista dei processi recenti mostrata dal PE dopo la fine del processo

È interessante confrontare le rappresentazioni grafiche SVG dello stato del processo generate in linea dal *Process Engine* durante lo svolgimento: i tre step riportati in figura 39 corrispondono rispettivamente all'avvio del processo, alla chiamata asincrona ed allo stato finale.

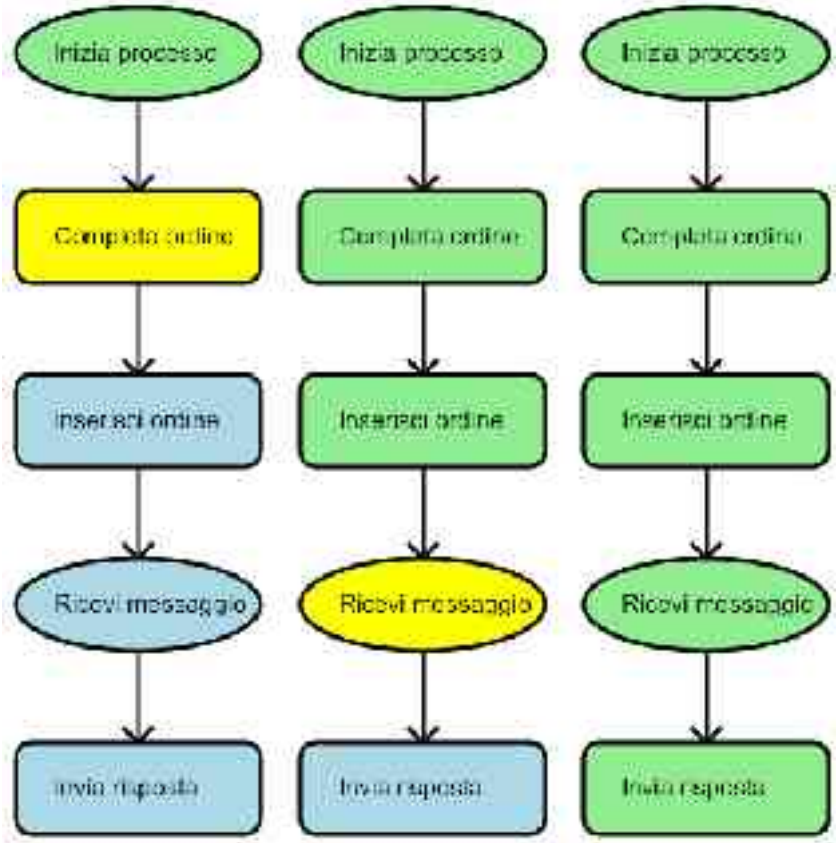

Figura 39: rappresentazione del processo catturata in tre momenti diversi

3.4.5.2 Fallimento con invio via SMS

Il secondo esempio mostra l'utilizzo di *Zlatan* con un input via SMS e la gestione di un caso di errore: un utente non registrato come cliente tenta di inviare un messaggio SMS al sistema (vedi listato 69) ed il processo viene terminato a causa dell'eccezione sollevata dal servizio `CompletePurchase Order`.

```
PO Material R-1000 10
```

Listato 69: testo del messaggio SMS inviato dall'utente non registrato

Il driver che raccoglie il messaggio è `SmsController`, che presenta un output sulla console come in figura 40.

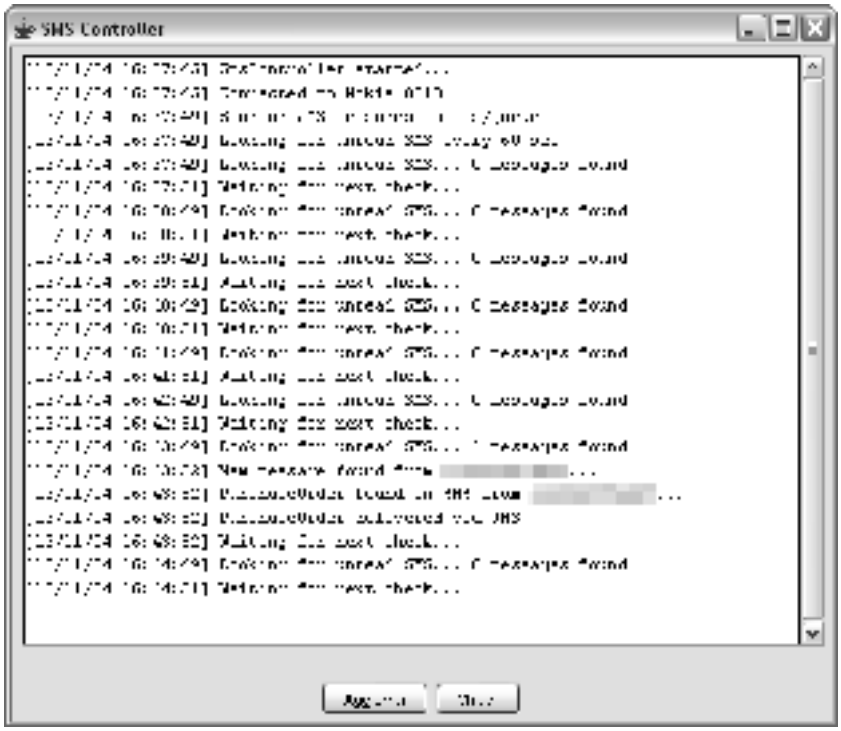

Figura 40: output prodotto dalla console di SmsController

L'ordine d'acquisto viene trasformato in formato XML in maniera simile a quanto visto nell'esempio precedente e viene quindi attivato un nuovo processo per l'elaborazione.

Il *Process Engine* invoca `CompletePurchaseOrder` con il messaggio del listato 70.

```
<env:Envelope ...>
  <env:Body  ...>
    <ddabpe1:completePO xmlns:ddabpe1="http://zlatan.org">
      <CompletePurchaseOrderInput>
        <From>+39328XXXXXXX</From>
        <Protocol>sms</Protocol>
        <PartnerNumber/>
      </CompletePurchaseOrderInput>
    </ddabpe1:completePO>
```

```
  </env:Body>
</env:Envelope>
```

Listato 70: messaggio SOAP inviato dal PE al servizio CompletePurchaseOrder

Il servizio non riesce a trovare una riga nella tabella `SENDERS` per la coppia `(From, Protocol)` specificata nel messaggio e quindi ritorna al processo il SOAP fault del listato 71. L'interprete BPEL cattura l'eccezione, termina il processo e genera la rappresentazione grafica di figura 41.

```
<soapenv:Envelope ...>
  <soapenv:Body>
    <soapenv:Fault>
      <faultcode>soapenv:Server.generalException</faultcode>
      <faultstring></faultstring>
      <detail>
        <ns1:PartnerNumberNotFoundElement
          xmlns:ns1="http://wsdl.zlatan.org">
          No partnerNumber for sms:+39328XXXXXXX
        </ns1:PartnerNumberNotFoundElement>
        <ns2:exceptionName
          xmlns:ns2="http://xml.apache.org/axis/">
          org.zlatan.wsdl.PartnerNumberNotFoundMessage
        </ns2:exceptionName>
      </detail>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

Listato 71: SOAP fault generato dal servizio CompletePurchaseOrder

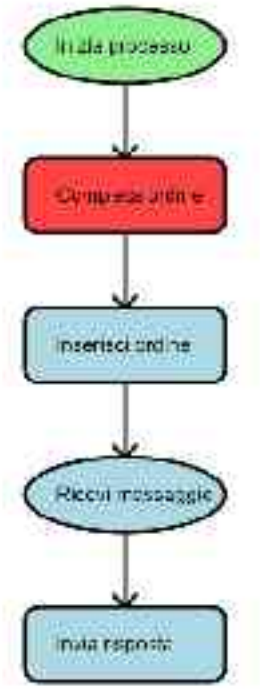

Figura 41: rappresentazione SVG di un processo fallito

4 Conclusioni

L'obiettivo della realizzazione di Zlatan era principalmente quello di testare alcuni aspetti del *Process Engine* e realizzare un prototipo dimostrativo funzionante. In particolare, è stato interessante approfondire l'interoperabilità del PE con la piattaforma Java, l'utilizzo di JMS come protocollo di trasporto e la distribuzione del sistema su più macchine. Altri aspetti interessanti hanno riguardato l'analisi dei problemi del linguaggio BPEL e lo studio delle soluzioni di connettività offerte da SAP.

Il progetto ha anche evidenziato tutte le potenzialità di Java: il linguaggio è stato usato come supporto per la soluzione di una vastissima gamma di tipologie di problemi, dalla programmazione distribuita all'interazione con periferiche seriali, dallo sviluppo di interfacce grafiche alla programmazione web lato server ed all'interrogazione di basi di dati.

4.1 Sviluppi futuri di Zlatan

Il proseguimento nell'implementazione di Zlatan non può non prescindere da una completa reingegnerizzazione del sistema, da effettuarsi sulla base delle conoscenze maturate durante la scrittura del primo prototipo.

Dovrebbe essere innanzi tutto definito in maniera precisa un documento di specifica dei requisiti, attività che non è stata fatta in precedenza a causa della natura estremamente sperimentale del progetto e della scarsa conoscenza iniziale degli strumenti da utilizzare.

Il prototipo suggerisce almeno tre caratteristiche da implementare in una ipotetica versione futura di Zlatan:

1. un ambiente SDK configurabile e parametrico che permetta la scrittura di nuovi driver per i protocolli in ingresso in maniera semplice e automatica, utilizzando la struttura già implementata (ma non standardizzata) per i driver del prototipo (architettura delle classi e dei package, interfaccia utente, pacchettizzazione, file di configurazione);
2. un sistema di autorizzazioni e riconoscimento basato su firma digitale e supportato dalle tecnologie XML già disponibili sul mercato;
3. un modello per la generalizzazione dei dati trattati: dal singolo ordine d'acquisto a qualsiasi tipo di dato aziendale, utilizzando un formato XML compatibile con tecnologie come SAP BC o SAP WAS (vedi § 2.3).

4.2 Considerazioni finali

L'integrazione delle applicazioni aziendali rappresenta sicuramente una buona parte del futuro del mercato dell'informatica. I nomi delle aziende citate più volte nel corso di questo documento ne sono una testimonianza più che valida: i principali produttori di software di tutto il mondo sono impegnati, in maniera diversa, nella specifica e nell'implementazione di protocolli condivisi che garantiscano l'interoperabilità tra le piattaforme.

L'architettura orientata ai servizi potrebbe rimanere per molto tempo il punto di riferimento di questo processo di integrazione, data la sua estrema generalità e le potenzialità non ancora completamente esplorate. Possono invece sorgere dubbi sulle tecnologie che al momento attuale la implementano: l'uso di XML comporta un certo grado di overhead dovuto alla rappresentazione testuale ed alla ridondanza delle informazioni; inoltre, WSDL, SOAP e gli altri standard citati sono afflitti da problemi ampiamente analizzati nei paragrafi precedenti. Non è quindi da escludersi che il mercato possa imporre anche in un futuro prossimo modelli, standard e protocolli (condivisi o imposti *de facto*) migliori di quelli attuali.

Per concludere, si può riprendere l'esempio già citato nell'introduzione. Allo stato attuale, l'aspetto tecnologico nell'ambito dell'integrazione di applicazioni business è ancora troppo preponderante nei confronti degli aspetti applicativi: la potenza e la flessibilità dell'architettura a servizi non potranno essere sfruttate completamente, finchè gli sviluppatori di sistemi *service oriented* ed EAI dovranno dedicare del tempo alla soluzione di problemi di basso livello come il tracciamento dei messaggi SOAP scambiati tra i servizi.

Appendice A – Apache Axis

Apache Axis [AXIS] è un tool di sviluppo per web service in ambiente Java realizzato dall'*Axis Development Team* della *Apache Software Foundation*. È disponibile una versione di Axis anche per C++.

Axis 1.1, la versione utilizzata nel progetto *Zlatan*, è composto da:

- una web application che si occupa di gestire l'ambiente di esecuzione dei servizi (*routing*, *istance pooling*, serializzazione e deserializzazione dei messaggi SOAP, ecc...);
- una API composta da classi di utilità per la scrittura di servizi web e da classi necessarie al funzionamento della web application e dei tool;
- una serie di tool, tra cui:
  - *WSDL2Java* per generare scheletri lato server e *stub* lato client dei servizi web a partire dalla descrizione WSDL;
  - *Java2WSDL* per generare la descrizione come servizio web di una classe Java;
  - diversi tool per l'amministrazione e la gestione dei servizi installati;
  - un *TCP monitor stand-alone* ed un *SOAP monitor* integrato nella web application per controllare la forma dei messaggi scambiati tra i servizi nelle fasi di debug e test.

Tra le caratteristiche più interessanti di Axis c'è la possibilità di creare web service in maniera immediata a partire da classi Java molto semplici con estensione `.jws` (*Java Web Service*).

Axis può essere quindi utilizzato in una serie di scenari anche molto diversi tra loro, ad esempio può servire per:

- *la creazione di applicazioni client di servizi web già esistenti per i quali è disponibile il WSDL*: utilizzando WSDL2Java si possono creare in maniera automatica gli stub per l'accesso a servizi esistenti implementati con qualsiasi piattaforma. Le applicazioni client che utilizzano gli stub non necessitano di ambienti di esecuzione particolari ma soltanto della presenza della libreria `axis.jar` nel proprio *classpath*;
- *la comunicazione via JAX-RPC tra processi Java*: le API di Axis implementano una versione di JAX-RPC, rendendo possibile lo sviluppo di applicazioni distribuite Java con protocollo di trasporto SOAP;
- *la creazione di servizi web a partire da classi Java*: Axis offre diversi meccanismi per l'implementazione dei servizi. Quello più semplice e completamente trasparente per il programmatore è JWS, ma sono disponibili anche modelli di servizi più complicati dove è possibile personalizzare, ad esempio, le modalità di serializzazione dei messaggi, la struttura dei package ed il formato dei parametri, senza mai occuparsi della descrizione WSDL o del formato SOAP dei messaggi, grazie all'integrazione tra l'ambiente di esecuzione e Java2WSDL;
- *l'implementazione di servizi web a partire da descrizioni WSDL*: il tool WSDL2Java è particolarmente utile quando, come nel caso di Zlatan, si parte dalla descrizione dei servizi per la realizzazione di un sistema piuttosto che dalla loro implementazione.

I problemi principali di Axis riguardano lo stretto legame con il web application container Tomcat (anche se è possibile con qualche sforzo installare l'ambiente in altri server, come fatto con SJSAS8 in Zlatan) e la conformità soltanto parziale alle specifiche del WS-I, come già evidenziato nel § 2.2.3.2.

Appendice B – Installazione e configurazione di Zlatan

Zlatan è un sistema dimostrativo non commerciale, basato su tecnologie eterogenee, alcune delle quali instabili o non abbastanza mature e consolidate: è naturale che l'installazione e la configurazione del sistema siano operazioni complesse.

Di seguito viene riportata una *checklist* di azioni da compiere nel caso si voglia installare il sistema su una macchina *stand-alone*. Tale guida può servire per comprendere meglio la struttura del sistema e le relazioni tra i vari moduli e file. Dalla guida è esclusa l'installazione di *GlobalBiz Process Engine*.

Operazioni preliminari:

- Installare SJSAS8 (vedi [J2EE]);
- Installare SAP JCo (vedi [ARAJCO]);
- Installare Apache Axis (vedi [AXIS]);
- Installare JDIC TrayIcon (vedi [JDIC]);
- Installare JavaCOMM (vedi [JCOMM]);
- Abilitare Axis in SJSAS8:
  - Copiare il file `wsdl4j.jar` in una directory di libreria di SJSAS8;
  - Garantire le autorizzazioni per l'esecuzione del codice di Axis nel file `java.policy` dell'installazione locale di *JRE* (*Java Runtime Environment*, vedi listato 72);
  - Deployare la web application presente nella cartella `AXIS_INST\webapps\axis` in SJSAS8;
- Abilitare SAP JCo in SJSAS8:
  - Copiare la libreria `sapjco.jar` in una directory di libreria di SJSAS8.

```
grant codeBase "file:c:/Programmi/Zlatan/-" {
  permission java.security.AllPermission;
};

grant codeBase "file:c:/Programmi/axis-1_1/webapps/axis/WEB-INF/lib/-" {
  permission java.security.AllPermission;
};
```

Listato 72: modifiche da introdurre al file java.policy per l'esecuzione di Zlatan

Copia dei file necessari per Zlatan:

- Creare una directory `Zlatan` senza spazi nel path (es. `C:\Programmi\Zlatan`);
- Copiare nella directory i file `completePO.mdb` e `POS.mdb`;
- Creare una sotto-directory `IBRA` e copiare i file necessari come in figura 42;
- Creare una sotto-directory `webapps\zlatan` e copiare i file necessari come in figura 42.

Installazione dei web service:

- Copiare i due file `zlatan-ws.jar` e `zlatan-deploy.wsdd` nella directory `lib` della web application di Axis;
- Copiare i file di configurazione dei web service (`XXX-ws-config.xml`) nella directory `WEB-INF` della stessa web application;
- Deployare i web service con il comando `adminclient zlatan-deploy.wsdd`;
- Creare una fonte ODBC per il database contenuto in `completePO.mdb` utilizzato da `CompletePurchaseOrder`.

Figura 42: struttura del filesystem da creare per Zlatan

Installazione di IBRA e dell'interfaccia grafica :

- Garantire le autorizzazioni per l'esecuzione del codice di Zlatan in maniera simile a quanto già fatto con Axis (vedi listato 72);
- Creare gli oggetti amministrati necessari per il funzionamento di JMS in SJSAS8: una `QueueConnectionFactory`, una coda per le richieste di registrazione dei moduli di elaborazione di IBRA ed una coda per l'invio dei messaggi dai driver al dispatcher (vedi [J2EETUT] per la procedura);
- Creare una fonte ODBC per il database contenuto in `POS.mdb` utilizzato dalla pagina per il controllo dello stato dell'ordine via HTTP;
- Deployare la web application presente nella cartella `ZLATAN_INST\webapps\axis` in SJSAS8.

Configurazione di Zlatan:

- Sincronizzare i valori dei nomi JNDI degli oggetti JMS nei file di configurazione dei moduli di IBRA e nel file `ZLATAN_INST\webapps\zlatan\WEB-INF\web.xml`;
- Configurare i valori per l'accesso all'host POP3 in `pop3controller-config.xml`, il nome della porta seriale collegata alla scheda GSM in `smscontroller-config.xml`, l'URL del `rmiregistry` in `rmicontroller-config.xml` e l'endpoint dell'interprete BPEL in `preparerequestmodule-config.xml`;
- Modificare le URL delle fonti ODBC nel file `web.xml` della web application di Zlatan e nel file di configurazione del web service `CompletePurchaseOrder`;
- Settare i parametri per la connessione a SAP R/3 e l'endpoint dell'interprete BPEL nel file di configurazione `insertpo-ws-config.xml`.

Avvio del sistema:

- Avviare nell'ordine:
  - l'interprete BPEL;
  - SJSAS8;
  - eventuali monitor TCP, HTTP o SOAP;
  - `PrepareRequestDispatcher`;
  - uno o più `PrepareRequestModule`;
  - i driver desiderati, ad eccezione di `HttpController` (prima di avviare `RmiController`, attivare `rmiregistry`).

Bibliografia

[ACTBPEL] *Active BPEL Engine Site*. http://www.activebpel.org

[ADVJCO] Schuessler; *Advanced SAP Java Connector (JCo) Programming*; SAP 2003.
http://www.sap.com/community/int/ShowDoc.aspx?docid=518&Type=Transcript
(necessita di account SAP Community)

[ARA] *ARASoft Site.* http://www.arasoft.de

[ARAJCO] Schuessler; *Developing Applications with the "SAP Java Connector" (JCo)*; ARASoft 2002.

[AXIS] Axis Developers Group; *Axis User's Guide*; Apache Software Found. 2004.
http://ws.apache.org/axis/java/user-guide.html

[BPEL] Andrews, Curbera, Dholakia e.a.;
Business Process Execution Language for Web Services Version 1.1; IBM 2003.
http://www-106.ibm.com/developerworks/library/ws-bpel

[DOXY] *Doxygen Site*. http://www.doxygen.org

[ECL] *eclipse.org Site.* http://www.eclipse.org

[FMQ] *FioranoMQ Site.* http://fiorano.best.vwh.net/products/fmq

[HTTPR] Todd, Parr, Conner; *A Primer for HTTPR*; IBM 2002.
http://www-106.ibm.com/developerworks/webservices/library/ws-phtt

[IBMBPEL3] Duftler, Khalaf; *Business Process with BPEL4WS: Learning BPEL4WS, Part 3;* IBM 2002.
http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol3

[IBMBPEL6] Khalaf, Nagy; *Business Process with BPEL4WS: Learning BPEL4WS, Part 6*; IBM 2003.
http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol6

[IBMBPEL7] Khalaf, Nagy; *Business Process with BPEL4WS: Learning BPEL4WS*, Part 7; IBM 2003.
http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol7

[IBMBPEL8] Khalaf, Mukhi; *Business Process with BPEL4WS: Learning BPEL4WS, Part 8*; IBM 2003.
http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol8

[IBMJMS] Farrell; *Writing Java Message Service programs using WebSphere MQ and WebSphere Studio Application Developer, Part 1*; IBM 2002.
http://www-106.ibm.com/developerworks/ibm/library/i-jmsmq

[IBMLEG] Kuebler, Eibach; *Adapting legacy applications as Web services*; IBM 2002.
http://www-106.ibm.com/developerworks/webservices/library/ws-legacy

[IBMMQSJMS] Farrell; *Writing Java Message Service programs using MQSeries and VisualAge for Java, Enterprise Edition*; IBM 2001.
http://www-128.ibm.com/developerworks/ibm/library/it-farrell1

[IBMSOA] Channabasavaiah, Tuggle; *Migrating to a service-oriented architecture*; IBM 2003.
http://www-106.ibm.com/developerworks/webservices/library/ws-migratesoa

[IBMWMQ] *WebSphere MQ (formerly MQ Series) Site.*
http://www-306.ibm.com/software/integration/wmq

[IBMWSDL] Butek; *Which style of WSDL should I use?*; IBM 2003.
http://www-106.ibm.com/developerworks/webservices/library/ws-whichwsdl

[IFR] *SAP Interface Repository*. http://ifr.sap.com

[J2EE] *Java 2 Platform, Enterprise Edition (J2EE)*. http://java.sun.com/j2ee

[J2EETUT] Armstrong, Ball, Bodoff, Carson e.a.; *The J2EE 1.4 Tutorial*; Sun 2004.
http://java.sun.com/j2ee/1.4/docs/tutorial/doc

[JCA] *J2EE Connector Architecture*. http://java.sun.com/j2ee/connector

[JCOMM] *Java Communications API*. http://java.sun.com/products/javacomm

[JDIC] *JDesktop Integration Component*. https://jdic.dev.java.net

[JSMS] *jSMSEngine Site*. http://www.jsmsengine.org

[MSMQ] *Microsoft Message Queuing (MSMQ) Center.*
http://www.microsoft.com/windows2000/technologies/communications/msmq

[MSSOA] *Service Oriented Architecture Microsoft Blueprints*.
http://msdn.microsoft.com/architecture/soa

[ORABPEL] *Oracle BPEL Process Manager*.
http://www.oracle.com/technology/products/ias/bpel

[PBBLOG] Paul Brown; *pbblog*; 2004. http://blogs.fivesight.com/pbblog/index.php?cat=6

[PKI] PKI TC; *PKI Action Plan*; OASIS Open Group 2004.
http://www.oasis-open.org/committees/pki/pkiactionplan.pdf

[SAPBPEL] *BPEL4WS Abstract*. http://ifr.sap.com/bpel4ws

[SOAP] Gudgin, Hadley, Mendelsohn, Moreau, Nielsen; *SOAP Version 1.2*; W3 Consortium 2003.
http://www.w3.org/TR/soap

[SOAP0] Mitra; *SOAP Version 1.2 Part 0: Primer*; W3 Consortium 2003.
http://www.w3.org/TR/soap12-part0

[UDDI] Bellwood, Clément, von Riegen; *UDDI Version 3.0.1*; OASIS Open Group 2003.
http://uddi.org/pubs/uddi_v3.htm

[W3CSOA] Haas; *Designing the architecture for Web services*; W3 Consortium 2003.
http://www.w3.org/2003/Talks/0521-hh-wsa

[WSA] Box, Christensen, Curbera e.a.; *Web Services Addressing*; W3 Consortium 2004.
http://www.w3.org/Submission/ws-addressing

[WSDL] Christensen, Curbera, Meredith, Weerawarana;
Web Services Description Language (WSDL) 1.1; W3 Consortium.
http://www.w3.org/TR/wsdl

[WSFL] Leymann; *Web Service Flow Language (WSFL 1.0)*; IBM 2001.
http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[WSIBP] Ballinger, Ehnebuske, Ferris e.a.; *Web Service Interoperability Basic Profile Version 1.1*;
WS-I Consortium 2004.
http://www.ws-i.org/Profiles/BasicProfile-1.1.html

[WSS] Atkinson, Della-Libera, Hada e.a.; *Web Services Security (WS-Security)*; IBM 2002.
http://www-106.ibm.com/developerworks/webservices/library/ws-secure

[XLANG] Thatte; *XLANG - Web Services for Business Process Design*; Microsoft 2001.
http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[XPATH] Clark, DeRose; *XML Path Language (XPath) Version 1.0*; W3 Consortium 1999.
http://www.w3c.org/TR/xpath

Note sul copyright

Questo documento, compresi i listati di esempio, le figure ed il codice originale allegato, hanno scopo didattico e non possono essere commercializzati in alcuna forma.

In particolare, il codice originale è posto sotto GPL e può essere liberamente distribuito, fino a indicazione contraria, a patto che ne venga citata la fonte e l'autore ed in accordo con gli usi previsti dalla licenza.

Tutti i marchi citati sono proprietà dei rispettivi detentori.

La figura 1 è tratta da [W3CSOA]. I class diagram del § 2.1 e di tutto il § 3 sono stati generati usando Doxygen e dot (vedi [DOXY]). La classe `MobilePhoneConnection` contiene codice ripreso da *jSMSEngine* (vedi [JSMS]).

Risorse sul web

All'indirizzo http://etd.adm.unipi.it è disponibile la versione elettronica di questo documento.

A tale indirizzo è possibile scaricare una copia dei sorgenti e dei binari del progetto Zlatan.