Ingegneria del Software Orientata ai Servizi: relazione di progetto

Bartolomeo Lombardi, Amerigo Mancino, Andrea Segalini

Università degli Studi di Bologna

bartolomeo.lombardi@studio.unibo.it
amerigo.mancino@studio.unibo.it
andrea.segalini@studio.unibo.it

Anno Accademico 2015-16

Sommario

Costruire sistemi software è un lavoro complesso. Costruire grandi sistemi software facendo interagire diverse componenti che appartengono ad organizzazioni differenti è ancora più complesso. I servizi e i processi sono le astrazioni che l'Ingegneria del Software insegna ad usare per affrontare questa complessità. In questa relazione descriveremo la progettazione e realizzazione delle attività di ACME, un'azienda che si occupa della fornitura e dell'assemblaggio di biciclette, compresi i relativi accessori. Verranno analizzate nel dettaglio tutte le fasi di modellazione affrontate, a partire dalla descrizione dei processi di ACME mediante un BPMS, fino ad arrivare all'esposizione esaustiva di alcune capability esterne implementate mediante API REST. Verranno inoltre discusse le difficoltà incontrate e saranno mostrati gli espedienti che hanno portato alla loro risoluzione.

I. Introduzione: tecnologie utilizzate

In questo paragrafo verranno brevemente descritte le principali tecnologie utilizzate durante la realizzazione del progetto:

- Camunda
  Piattaforma open-source per la gestione di processi business e di diagrammi di flusso, oltre che per l'orchestrazione dei servizi, usata principalmente, nel nostro caso, per la simulazione di human workflow.
- Jolie
  Linguaggio di programmazione "orientato ai servizi" in grado di supportare la nozione di *operation* e che facilita la comunicazione fra processi ed entità differenti.
- Signavio Process Editor
  Lanciato nel Maggio del 2009, è un tool completamente gratuito e web-based per la modellazione di processi business.
- HSQLDB
  Scritto principalmente in Java e sviluppato da HSQLDB Developer Group, è un RDBMS rilasciato con licenza libera e completamente compatibile con Jolie che può essere usato sia come server che come istanza interna ad un'applicazione.

- JAX-WS
  Java API for XML Web Services è un insieme di procedure (API) del linguaggio di programmazione Java dedicate allo sviluppo di servizi web, che ci sono servite, ad esempio, nell'implementazione dei web services client e server in Java.

II. Interpretazione della specifica

L'azienda ACME si occupa di fornire biciclette assemblate su richiesta ed accessori a rivendite che operano direttamente con il pubblico. Gli accessori e i componenti sono stati intesi genericamente come "pezzi", i quali possono essere "assemblabili", ossia montati nel magazzino principale, oppure "non assemblabili", ossia spediti direttamente alle rivendite e quindi venduti separatamente o montati in loco. Per esempio, abbiamo considerato il pezzo "ruota" come assemblabile mentre il pezzo "campanello" come non assemblabile. In prima approssimazione, ad ACME giungono degli ordini costituiti da liste di pezzi, ognuno dei quali è rappresentato con il codice prodotto e con la quantità richiesta. Quando viene ricevuto un ordine, l'azienda controlla che i pezzi siano compatibili tra loro e che siano presenti tutti quelli indispensabili per la costruzione della bicicletta. In caso negativo, un impiegato notifica al cliente il problema e lo invita a ripetere dall'inizio l'ordine, il quale viene abortito, terminando il processo. In caso positivo, invece, il processo procede verificando la disponibilità dei pezzi come segue:

1. Si controlla se i pezzi sono presenti già nel magazzino principale.
2. Per tutti quelli non presenti, vengono contattati tutti i magazzini secondari e recuperate le informazioni.
3. Qualora ci fossero ancora pezzi mancanti, viene contattato un fornitore esterno ad ACME.

Al termine di questa fase di controllo, otteniamo una tabella contenente, per ogni pezzo, le informazioni sul magazzino che lo possiede e in quale quantità. Nel caso del fornitore esterno, recuperiamo anche il prezzo di vendita. Queste informazioni saranno anche utilizzate per determinare, per ogni pezzo non assemblabile, il magazzino geograficamente più vicino al cliente che si preoccuperà della spedizione tramite un corriere. Se, al termine di questa procedura, dovessero esserci dei pezzi mancanti (non disponibili), si avviserà il cliente di tale problema e il processo verrà abortito senza possibilità di soluzione.
Riguardo i pezzi non assemblabili, per ogni pezzo viene eseguito un ordinamento basato sulla distanza dei magazzini che lo posseggono dal cliente e vengono scelti i magazzini più vicini, fino ad esaurire la quantità richiesta. Ad esempio, se vengono richiesti due campanelli e il magazzino più vicino al cliente ne possiede uno solo, quest'ultimo si preoccupa di spedire l'oggetto nella quantità che possiede e i restanti sono gestiti dal successivo magazzino nella graduatoria.
Nel caso in cui tutti i pezzi siano disponibili in qualche modo, viene preparato un preventivo e valutata l'applicazione di uno sconto. Indi, viene inviata la proposta di pagamento al cliente, il quale ha cinque giorni di tempo per accettare o rifiutare l'offerta. In caso di diniego, il processo termina. In caso di accettazione, invece, il cliente invia un riferimento alla ricevuta del versamento di un anticipo verso ACME. Abbiamo ipotizzato che l'azienda verifichi periodicamente se il pagamento è stato effettuato, chiedendo conto alla banca. Se, dopo tre giorni, l'accredito non dovesse essere ancora pervenuto, l'intero ordine viene annullato.
Nella fase successiva, è necessario riservare tutti i pezzi in maniera tale da evitare possibili conflitti fra ordini concorrenti. Questa operazione avviene come se fosse una transazione, ossia tutti i pezzi vengono giustamente riservati, oppure vengono disfatte tutte le prenotazioni di quell'ordine già effettuate e l'ordine stesso viene cancellato. In parallelo si contattano tutti i magazzini secondari, il magazzino principale e il fornitore esterno. Se, alla fine, tutti i pezzi sono stati correttamente riservati, si notifica ad ognuno dei magazzini e al supplier di procedere alla spedizione, mediante un corriere. In base al pezzo, la spedizione è divisa in due gruppi: una con destinazione il cliente e/o una con destinazione il magazzino principale. Da questo momento in poi l'ordine non potrà più essere annullato.
In seguito, il magazzino principale riceve parallelamente da tutte le sedi secondarie i vari componenti e procede con l'assemblaggio e la spedizione del prodotto finale. Quindi, ACME rimane in attesa della ricevuta da parte del cliente dell'avvenuto pagamento restante. La verifica avviene contattando la banca nella stessa modalità esposta in precedenza.

Se, dopo tre giorni, l'accredito non dovesse essere ancora pervenuto, l'amministrazione dell'azienda si attiva per prendere i dovuti provvedimenti. In ogni caso, a questo punto, il processo termina.

III. SOA MODELING

In questo paragrafo descriveremo brevemente le capabilities offerte dal sistema, con particolare enfasi a quelle che sono state maggior oggetto di studio durante l'elaborazione del progetto. Il nostro business espone all'utente una sola capability, che avvia il processo di costruzione e consente di effettuare un ordine. Il magazzino principale si occupa della coordinazione di tutti i magazzini secondari (compresa la parte operativa di sé stesso) e, pertanto, deve essere capace di:

- Mandare richieste di "verifica disponibilità" verso tutti i magazzini secondari e presso un eventuale fornitore esterno.
- Mandare richieste di "riserva pezzi" a tutti i magazzini secondari e presso un eventuale fornitore esterno.
- Mandare richieste di "elimina pezzi" verso tutti i magazzini secondari e presso un eventuale fornitore esterno.
- Eseguire l'ordine, informando i magazzini secondari di procedere alla spedizione dei diversi pezzi richiesti (se necessario), mettendosi poi in attesa delle varie consegne.
- Assemblare il ciclo e preparare il tutto per la spedizione.

Come si vede, il magazzino principale è quindi composto da una parte, appunto, "principale", che funge da coordinatore, e da un magazzino secondario. In aggiunta, un generico magazzino secondario deve poter:

- Verificare la disponibilità dei pezzi e recuperare le informazioni a riguardo per il suo magazzino.
- Riservare i pezzi presso il suo magazzino.
- Eliminare la prenotazione di determinati pezzi presso il suo magazzino.
- Eseguire l'ordine, che comporta l'invio dei pezzi verso il magazzino principale oppure direttamente verso l'utente affidandosi ad un corriere esterno.

L'amministrazione, a sua volta, deve essere in grado di:

- Creare una nuova istanza dell'ordine.
- Verificare la compatibilità di pezzi.
- Calcolare le distanze fra due punti geografici.
- Chiedere alla banca se un certo ammontare è stato pagato da un cliente.
- Verificare i pagamenti, contattando la banca.
- Trovare il miglior magazzino per ogni pezzo dell'ordine.

Sappiamo anche che il fornitore esterno, nonostante non faccia parte del nostro business, è capace di:

- Verificare la disponibilità dei pezzi e recuperare le informazioni disponibili a riguardo.

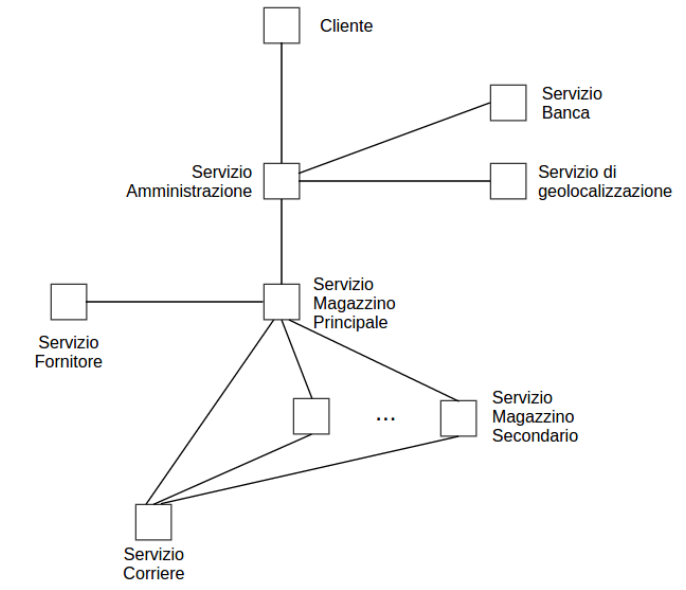


Figura 1: *Schema delle interazioni fra componenti del sistema.*

- Riservare i pezzi richiesti da ACME.
- Eliminare la prenotazione di determinati pezzi.
- Eseguire l'ordine, spedendo i pezzi richiesti al magazzino principale di ACME.

Chiaramente alcune delle capabilities qui esposte sono state accorpate per essere implementate come un unico servizio offerto all'utente.

La nostra SOA, quindi, è formata da tre servizi fondamentali: l'amministrazione, il magazzino principale e un servizio di magazzino secondario. La prima espone come operation al cliente una sola capability che consente di effettuare l'ordine e di ricevere il prodotto finale. Le altre capabilities amministrative non vengono esposte sotto forma di operations ma sono integrate nell'applicazione e non rese accessibili all'esterno. Sia il servizio magazzino principale che il secondario implementano tutte le capabilities sopra indicate.

Illustriamo brevemente in figura 1 le interazioni fra le varie componenti del sistema.

IV. Implementazione dei servizi

In questo paragrafo tratteremo delle scelte implementative che abbiamo preso nella realizzazione della nostra SOA, descrivendo anche i web service server implementati con Jolie. Abbiamo definito due applicativi principali che espongono le capabilities del magazzino principale (amministrazione e coordinazione) e secondario (parte operativa).

I. Magazzino principale

Realizzato in Jolie, l'interfaccia del servizio è descritta di seguito:

```
include "dataTypes.iol"
```

```
interface InterfacciaMagazzinoPrincipale {
  RequestResponse:
    verificaDisponibilitaMagazzini(RichiestaOrdine)(RispostaVerificaDisponibilita),
    verificaDisponibilitaMagazzino(RichiestaMagazzino)(RispostaMagazzino),
    verificaDisponibilitaFornitore(RichiestaFornitore)(RispostaFornitore),

    prenotaPezziMagazzini(RichiestaPrenotazioneMagazzini)(PezziMancanti),
    prenotaPezziMagazzino(RichiestaPrenotazioneMagazzino)(PezziMancanti),
    prenotaPezziFornitore(RichiestaPrenotazioneFornitore)(PezziMancanti),

    creaIstanzaOrdine(RichiestaCreaOrdine)(void),

    eliminaPrenotazioneMagazzini(RichiestaEliminazioneMagazzini)(void),
    eliminaPrenotazioneMagazzino(RichiestaEliminazioneMagazzino)(void),
    eliminaPrenotazioneFornitore(RichiestaEliminazioneFornitore)(void),

    eseguiOrdineMagazzini(RichiestaEsecuzioneMagazzini)(void),
    eseguiOrdineMagazzino(RichiestaEsecuzioneMagazzino)(void),
    eseguiOrdineFornitore(RichiestaEsecuzioneFornitore)(void),

    assemblaCiclo(RichiestaAssembla)(void)
}
```

dove *dataTypes.iol* è il file contenente le definizioni di tutti i tipi di dato scambiati durante l'invocazione delle operazioni. Il magazzino principale funge, dunque, da coordinatore di tutti i magazzini secondari, verificando la disponibilità dei pezzi, prenotandoli o eliminandoli da un ordine esistente. I magazzini secondari, che descriveremo nel dettaglio in seguito, rappresentano invece la parte operativa di un magazzino, ossia possiedono unicamente la capacità di memorizzare le quantità di pezzi da loro posseduti, gli ordini (lotti di pezzi da recapitare al cliente), etc. Le operations di questa interfaccia sono esposte attraverso la porta Jolie *MagazzinoPrincPort*:

```
inputPort MagazzinoPrincPort {
  Location: "socket://localhost:8000"
  Protocol: soap {
    .wsdl = "file:magazzinoPrincipale.wsdl";
    .wsdl.port = "MagazzinoPrincServicePort";
    .dropRootValue = true
  }
  Interfaces: InterfacciaMagazzinoPrincipale
}
```

La porta utilizza il protocollo SOAP e il suo corrispondente WSDL è stato generato mediante il comando:

```
$ jolie2wsdl -i ~/Scrivania/interfacce/:/usr/lib/jolie/include/ \
  -namespace org.acme.magazzino-principale -portName MagazzinoPrincPort \
  -portAddr http://localhost:8000 \
  -o magazzinoPrincipale.wsdl magazzinoPrincipale.ol
```

Il servizio comunica con le altre entità attraverso le seguenti output port:

```
outputPort ServizioMagazzinoPrincipale {
  Location: "socket://localhost:8000"
  Protocol: soap
```

```
  Interfaces: InterfacciaMagazzinoPrincipale
}
```

ServizioMagazzinoPrincipale consente al magazzino di comunicare con sé stesso, in quanto operations come la `verificaDisponibilitaMagazzini`, ad esempio, richiama, per tutti gi *n* magazzini secondari, l'operazione di `verificaDisponibilitaMagazzino`. La prima, infatti, si occupa di invocare la seconda per ogni magazzino gestito da ACME, mettendo poi insieme le informazioni ricavate mentre la `verificaDisponibilitaMagazzino` permette di contattare immediatamente il magazzino stabilito. Le altre operations che si occupano di eseguire l'ordine, prenotare i pezzi o eliminare una prenotazione funzionano pressappoco nella stessa maniera. Nel database registriamo tutte le informazioni necessarie per gestire la molteplicità di magazzini secondari, ossia, in breve:

- Un identificatore opportuno.
- L'URL del servizio magazzino secondario (nota: questo approccio molto naive è stato adottato per semplicità, evitando di addentrarsi in strumenti per la gestione del naming).
- Un booleano che identifica se si tratta di un fornitore esterno oppure di un magazzino interno (nota: anche i fornitori vengono memorizzati insieme ai magazzini secondari per semplicità).

Mediante questo stratagemma, riusciamo a ridurre la complessità del codice. La altre porte:

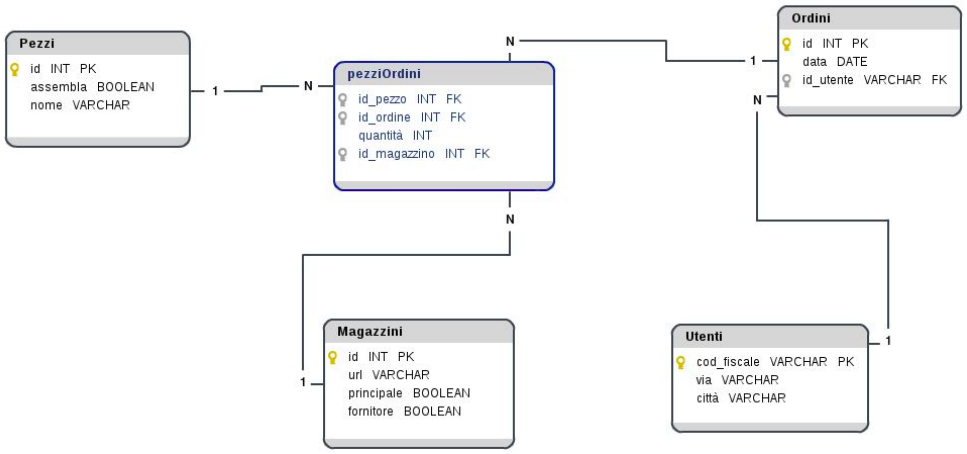
```
outputPort ServizioMagazzinoSecondario {
  Protocol: soap
  Interfaces: InterfacciaMagazzinoSecondario
}

outputPort ServizioFornitore {
  Protocol: soap
  Interfaces: InterfacciaFornitore
}
```

non possiedono la direttiva di *location*, dal momento che viene impostata dinamicamente prima di una chiamata in base ai magazzini secondari presenti nel database. Nel codice mostrato di seguito notiamo come in `tabellaMagazzini` sono presenti le informazioni dei magazzini e, in questo caso, il campo `url` viene assegnato al campo `location` della porta `ServizioMagazzinoSecondario`:

```
ServizioMagazzinoSecondario.location = tabellaMagazzini.(idMagazzino).url;
```

Sottolineiamo la presenza di un file di configurazione che, nel nostro intento originario, doveva servire per impostare i parametri di deployment (location e port). Tuttavia in Jolie è possibile settare tali parametri solo staticamente. Questo file è stato utilizzato allora principalmente per caricare i dati relativi alla connessione con il DB (utente e password, driver, porta, url, etc). Il database è gestito da HSQLDB [1], scelto poiché consigliato dalla documentazione ufficiale di Jolie. La sua struttura è descritta dal diagramma mostrato di seguito:

[1] Sito web `www.hsqldb.org`



Ribadiamo che, la tabella `pezzi`, nel caso del magazzino principale, funge da "catalogo" dell'intera collezione di oggetti che ACME dispone e non è a conoscenza della quantità posseduta dai magazzini secondari. In aggiunta, è necessario memorizzare anche le informazioni relative all'utente, dal momento che il fornitore esterno necessita di sapere l'indirizzo di eventuali spedizioni di pezzi non assemblabili.

II. Magazzino secondario

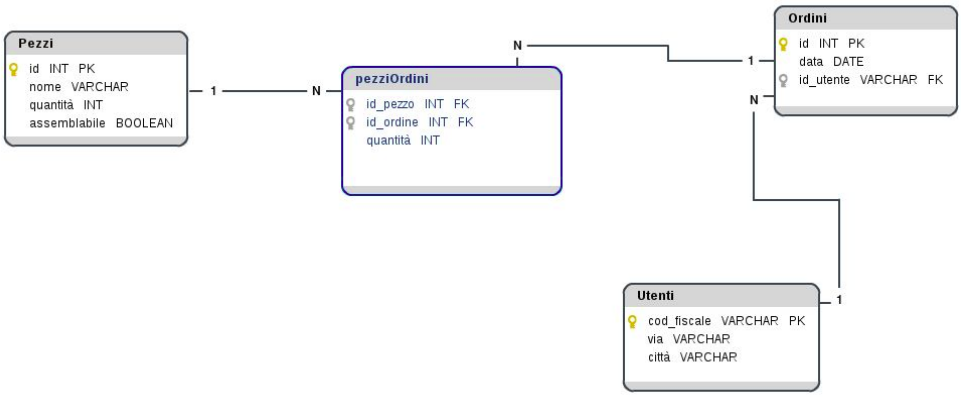
Il servizio magazzino secondario, realizzato anch'esso in Jolie, implementa tutte le capabilities esposte nel paragrafo III. La sua interfaccia è mostrata di seguito:

```
include "dataTypes.iol"

interface InterfacciaMagazzinoSecondario {
  RequestResponse:
    verificaDisponibilitaPezzi(RichiestaDisponibilitaPezzi)(RispostaMagazzinoSecondario),
    prenotaPezzi(RichiestaPrenotazionePezzi)(PezziMancanti)

  OneWay:
    eliminaPrenotazionePezzi(int),
    eseguiOrdinePezzi(int)
}
```

Il magazzino secondario si occupa della verifica della presenza di pezzi al suo interno, oltre che della spedizione dei suddetti pezzi verso il magazzino principale oppure direttamente verso il cliente. Il servizio è esposto attraverso la porta Jolie:

```
inputPort MagazzinoSecPort {
  Location: "socket://localhost:8002"
  Protocol: soap
  Interfaces: InterfacciaMagazzinoSecondario
}
```

Per il magazzino secondario valgono all'incirca le stesse considerazioni fatte nel paragrafo precedente. Di questo servizio non è stato generato un WSDL poiché, allo stato attuale, serve unicamente il magazzino principale (scritto anch'egli in Jolie, come già detto). Inoltre, poggia anch'esso su un datatabase, la cui struttura è mostrata di seguito:

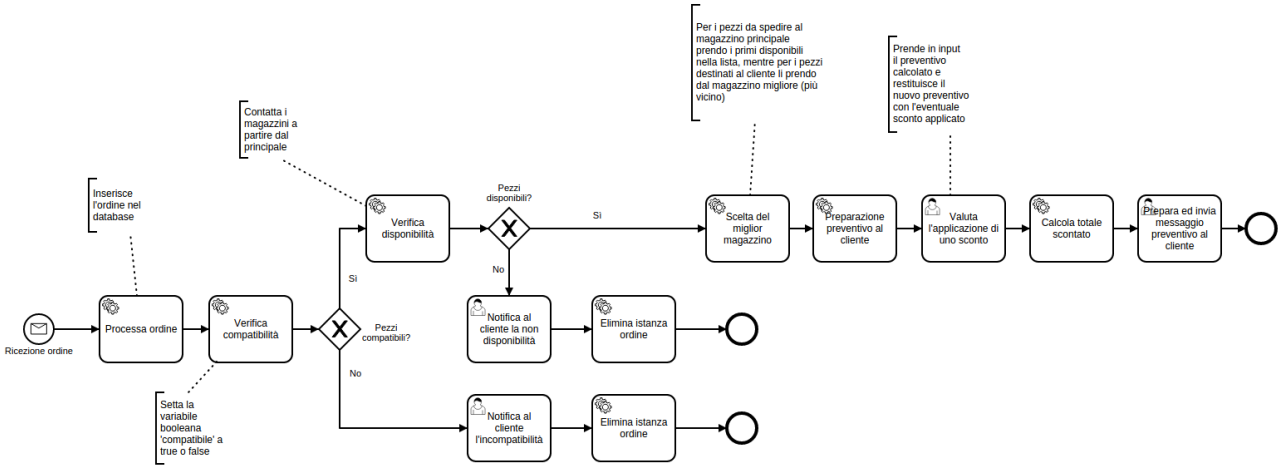


In questo caso, la tabella `pezzi` contiene le informazione relative ai diversi componenti posseduti dal magazzino fisico, compresa la quantità. I dati sugli utenti sono necessari per il recapito di un'eventuale spedizione. Notiamo che i magazzini secondari non memorizzano informazioni sugli altri.

III. Amministrazione

Il servizio di amministrazione espone come operations l'intero processo business di ordine e assemblaggio del ciclo e funge, quindi, da orchestratore dell'applicazione, richiamando le operations delle altre componenti quando opportuno. È stato implementato utilizzando Camunda [2], un BPMS che funge da execution engine per un diagramma BPMN, offrendo di default un'interfaccia grafica per la parte di human workflow.

In questo progetto, è stata realizzata una parte del diagramma dell'amministrazione, riportata di seguito:



Le service task sono state realizzate attraverso una classe Java che implementa l'interfaccia `JavaDelegate`. Il codice presente all'interno del metodo `execute` sarà quello eseguito dall'execution engine durante l'avanzamento del flusso.

Nel processo vengono invocate operations di web services attraverso Java e l'utilizzo di JAX-WS. In particolare, è stata utilizzata l'implementazione presente in Apache CXF [3], con il quale sono state generate, partendo dal WSDL del

[2] Sito web `www.camunda.org`
[3] Sito web `www.cxf.apache.org`

magazzino principale, delle classi Java. Tra queste è presente la rappresentazione dei tipi di dati scambiati durante le invocazioni e il proxy per effettuare le chiamate.

Per la generazione, è stato necessario aggiungere al file `pom.xml` il plugin di CXF, modificando WSDL e relativa location:

```
<plugin>
     <groupId>org.apache.cxf</groupId>
     <artifactId>cxf-codegen-plugin</artifactId>
     <version>3.1.4</version>
     <executions>
        <execution>
            <id>generate-sources</id>
            <phase>generate-sources</phase>
            <configuration>
                <sourceRoot>${project.build.directory}/generated/cxf</sourceRoot>
                <wsdlOptions>
                    <wsdlOption>
                        <wsdl>${basedir}/src/main/resources/magazzinoPrincipale.wsdl</wsdl>
                        <wsdlLocation>classpath:magazzinoPrincipale.wsdl</wsdlLocation>
                    </wsdlOption>
                </wsdlOptions>
            </configuration>
            <goals>
                <goal>wsdl2java</goal>
            </goals>
        </execution>
     </executions>
</plugin>
```

Il processo viene attivato dalla ricezione di un messaggio SOAP che invoca l'operation di "faiOrdine". I parametri del messaggio sono i dati dell'utente e l'ordine (lista dei pezzi con relativa quantità). Questi ultimi sono rappresentati dalle classi autogenerate `Utente` e `List<Pezzo>`. Per consentire ciò, è stato realizzato un web service server Java, mostrato di seguito:

```
@WebService
public class StartProcessService {
  @Resource(mappedName = "java:global/camunda-bpm-platform/process-engine/default")
  private ProcessEngine processEngine;

  @WebMethod
  public void faiOrdine(Utente utente, List<Pezzo> pezzi) {
    Map<String, Object> vars = new HashMap<String, Object>();
    vars.put("utente", utente);
    vars.put("ordine", pezzi);

    processEngine.getRuntimeService().startProcessInstanceByMessage("inizia", vars);
  }
}
```

Nel corpo del metodo, utilizziamo delle primitive che consentono di avviare il processo Camunda attraverso un messaggio identificato da `inizia`. Notiamo che i parametri `utente` e `pezzi` vengono inseriti nella sessione del processo appena

avviato. Questa soluzione è attuabile soltanto in application server JBoss e richiede che entrambi gli applicativi di web server services e Camunda risiedano all'interno della stessa piattaforma.

È degna di nota la classe `VerificaDisponibilita`, di cui riportiamo un frammento, delegate del task "Verifica disponibilità Magazzini", in cui vengono eseguite le invocazioni dei servizi Jolie dei magazzini tramite SOAP:

```
List<Pezzo> ordine = (List<Pezzo>) execution.getVariable("ordine");
Utente utente = (Utente) execution.getVariable("utente");
int idOrdine = (Integer) execution.getVariable("idOrdine");

Holder<List<DisponibilitaPezzi>> holderDisponibilitaPezzi =
   new Holder<List<DisponibilitaPezzi>>();
Holder<List<Pezzo>> holderPezziMancanti =
   new Holder<List<Pezzo>>();

MySelfService service = new MySelfService();
MySelf port = service.getMySelfServicePort();

port.verificaDisponibilitaMagazzini(ordine,
    utente,
    holderDisponibilitaPezzi,
    holderPezziMancanti);
```

Qui viene mostrata l'esecuzione della chiamata al web service: nella prima parte vengono prelevati dalla sessione i dati da utilizzare come parametri per l'invocazione. Dopodiché sono inizializzati gli `Holder` che fungono da contenitori per i valori di ritorno, mentre le due istruzioni successive servono per l'inizializzazione del proxy attraverso i cui metodi viene effettivamente eseguita l'operazione di chiamata propriamente detta.

Un esempio significativo di Service Task è la classe `SceltaMagazzino`, la quale, ricevendo in input una tabella in cui, per ogni pezzo, corrisponde l'insieme dei magazzini che lo posseggono, restituisce in output un insieme di ordini da effettuare per ogni magazzino. In particolare, viene prima calcolata la distanza dei diversi magazzini con l'utente, ottenuta tramite l'utilizzo di API Rest di Google Maps, e in seguito vengono ordinati i diversi magazzini tenendo conto di tale distanza. Per ogni pezzo vengono quindi considerati in ordine i diversi magazzini fino al completo soddisfacimento della richiesta.

Per quanto riguarda le human task, Camunda offre una web GUI al cui interno è possibile modificare l'interfaccia visualizzata dall'utente umano, adattandola al compito da assolvere. Queste interfacce vengono realizzate tramite HTML e Javascript per la gestione dei dati e della loro presentazione nella pagina. Di seguito inseriamo un esempio di form relativo alla valutazione di uno sconto (`valuta-sconto-form.html`) da parte dell'operatore. Nel seguente frammento illustriamo in che modo vengono prelevate dalla sessione le variabili di interesse:

```
<form name="ListaPezziIncompatibili" role="form">

  <script cam-script type="text/form-script">
    camForm.on('form-loaded', function() {
    camForm.variableManager.fetchVariable('jsonOrdine');
    camForm.variableManager.fetchVariable('utente');
    camForm.variableManager.fetchVariable('idOrdine');
     camForm.variableManager.fetchVariable('prezzoTotale');
    });

    camForm.on('variables-fetched', function() {
    $scope.jsonOrdine = camForm.variableManager.variable('jsonOrdine').value;
```

```
  $scope.utente = camForm.variableManager.variable('utente').value;
  $scope.idOrdine = camForm.variableManager.variable('idOrdine').value;
  $scope.prezzoTotale = camForm.variableManager.variable('prezzoTotale').value;
  });
</script>
```

Si noti la classe `PezziWrapper` dell'attributo `jsonOrdine`, espediente utilizzato in modo tale che Camunda serializzasse in formato json la lista di pezzi, requisito necessario per poter essere acceduta all'interno delle form. Nella seguente porzione invece viene stampata una tabella contenente ordine e quantità:

```
<p class="lead">Ordine:</p>
<table border="1" style="width:100%">
    <thead>
      <tr>
        <td class="control-label">Id Pezzo</td>
        <td class="control-label">Quantita</td>
      </tr>
    </thead>
    <tbody>
   <tr ng-repeat="pezzo in jsonOrdine.pezzi">
    <td>{{pezzo.idPezzo}}</td>
    <td>{{pezzo.qta}}</td>
      </tr>
  </tbody>
</table>
<br>
```

Infine, in quest'ultimo pezzo di codice si è deciso di inserire una form che permette ad un utente umano di applicare uno sconto al costo totale:

```
<p class="lead">Sconto (in percentuale):</p>
<address>
  <strong>{{prezzoTotale}} Euro</strong>
</address>
<form>
  <input required
       type="number"
       cam-variable-name="sconto"
       cam-variable-type="Double"
       min="0"
       max="100"/> %
</form>
<br>
</form>
```

Gli attributi `cam-variable-name` e `cam-variable-type`, propri di Camunda, permettono di introdurre una nuova variabile nella sessione.

Di seguito illustriamo una delle form che abbiamo generato per la simulazione del processo:

IV. Servizi aggiuntivi

Descriviamo brevemente i servizi accessori che sono stati integrati nel nostro sistema.

Calcolo distanze geografiche. Per ogni prodotto richiesto, il magazzino che viene selezionato dal sistema per la spedizione è quello che ha il prodotto disponibile e che è geograficamente più vicino alla sede del cliente. Quindi, per rendere automatica tale procedura, si è sviluppata una funzione che prende in input due città e ritorna la distanza tra queste in chilometri. Per l'implementazione sono state usate API REST, facendo affidamento su Google, che mette a disposizione, attraverso *Google Maps API*, la possibilità di effettuare richieste REST, ottenendo cosi informazioni dettagliate sul percorso preso in considerazione. Un esempio concreto di richiesta, è il seguente:

```
https://maps.googleapis.com/maps/api/distancematrix/json?origins=parma&destinations=pisa
```

Le risposte alle query di *Google Maps Distance Matrix API* vengono restituite nel formato indicato dal flag di uscita all'interno del percorso di richiesta dell'URL. Infatti tale richiesta risponde in output con una pagina in formato JSON, contenente le informazioni di percorso tra Bologna (settata come parametro d'esempio di origin) e Napoli (che in questo caso è settato come destination):

```
{
   "destination\_addresses" : [ "Napoli, Italia" ],
   "origin\_addresses" : [ "Bologna, Italia" ],
   "rows" : [
```

```
     {
        "elements" : [
           {
              "distance" : {
                 "text" : "575 km",
                 "value" : 575400
              },
              "duration" : {
                 "text" : "5 ore 34 min",
                 "value" : 20046
              },
              "status" : "OK"
           }
        ]
     }
  ],
  "status" : "OK"
}
```

Si noti che se si desidera estrarre dei valori dai risultati, in genere si necessita di analisi e parsing JSON. Il parametro di interesse è `text`, contenuto nell'oggetto `distance`, ossia il valore testuale che contiene la distanza del percorso.

Siccome inizialmente la chiamata "calcola distanza" doveva essere effettuata all'interno di un servizio Jolie, è stato introdotto un intermediario, sempre realizzato in Jolie, capace di effettuare le chiamate REST al servizio di Google, esponendole poi attraverso web service SOAP. La porta per effettuare richieste HTTP in Jolie è la seguente:

```
outputPort DistanceMatrixService {
       Location: "socket://maps.googleapis.com:80/"
       Protocol: http {
              .method = "get";
              .osc.default.alias = "maps/api/distancematrix/xml";
              .format = "xml"
       }
       Interfaces: DistanceMatrixInterface
}
```

All'interno della service task di Camunda, invece, la chiamata è stata effettuata direttamente in Java senza passare per attraverso l'intermediario Jolie:

```
URL url = new URL("https://maps.googleapis.com/maps/api/distancematrix/json?origins=" +
            origine + "&destinations=" + destinazione);
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setRequestProperty("Accept", "application/json");
BufferedReader br = new BufferedReader(new InputStreamReader((conn.getInputStream())));
```

Sistema bancario. ACME deve verificare, attraverso l'interazione con un sistema bancario, il pagamento degli importi che gli vengono accreditati. Quindi è stato implementato un sistema elementare attraverso richieste REST. Passando come parametro un identificativo del bonifico, risponde (in maniera casuale), tramite un booleano, l'avvenuta o la mancata esecuzione del pagamento. L'output contiene, poi, anche la somma pagata dal cliente che deve essere poi verificata corrispondere all'importo richiesto.

```
@Path("/bonifico")
public class Bonifico {

	public static boolean getRandomBoolean() { return Math.random() < 0.5; }

	@GET
	@Produces("application/json")
	public Response paytobank(@QueryParam("i") int i) throws JSONException {
		JSONObject jsonObject = new JSONObject();

		int price = 50;
		if(getRandomBoolean() == true){
			jsonObject.put("paycheck", true);
			jsonObject.put("price", price);
		} else{
			jsonObject.put("paycheck", false);
			jsonObject.put("price", price);
		}
		return Response.status(200).entity(jsonObject.toString()).build();
	}
}
```

Si noti che questa metodologia di pagamento è stata ipotizzata arbitrariamente, non conoscendo le esatte funzionalità realmente offerte da una banca moderna.

Fornitore esterno. Il servizio di fornitore esterno è stato implementato con un'interfaccia di fatto identica ai magazzini secondari. Tuttavia ciò non deve indurre a pensare che abbiano una qualche relazione strutturale. Nel magazzino principale, infatti, c'è una netta distinzione fra la gestione delle operazioni verso i magazzini secondari e di quelle verso il fornitore esterno. La similarità fra essi è una pura semplificazione.

V. Guida al deployment

In questa sezione verranno illustrate le operazioni per la messa in opera di tutto il sistema. È richiesto:

- Java Development Kit 7.
  Nel progetto si è scelto di usare OpenJDK.
- Un application server JBoss AS 7.4.0.
  In particolare, poi, nel progetto si è adoperato quello distribuito da Camunda [4].
- Jolie Language.
  Nel nostro caso, si è adottata la versione 1.4.1 [5].
- Apache Maven.
  Per i nostri esperimenti, si è installata la versione 3.3.3.
- Un accesso ad internet per accedere ai web services di Google.

[4] Pagina di download `www.camunda.org/download`
[5] Sito web `www.jolie-lang.org`

I. Deployment del magazzino principale

La struttura generale dell'applicativo può essere sintetizzata nel seguente albero:

```
magazzino_principale
├── magazzinoPrincipale.ol Applicazione Jolie service server
├── creaDBMagazzinoPrincipale.ol Crea il database "db/DB_magazzinoPrinc"
├── popolaDBMagazzinoPrincipale.ol Popola il database con dati di prova
├── lib
│   └── hsql.jar DBMS hsql
└── db
    └── ...  File del DB hsql "DB_magazzinoPrinc"
```

È fondamentale che le directory `interfacce` e `magazzino_principale` (che, ricordiamo, possiede anche un magazzino secondario) siano contenute nella stessa cartella:

```
root
├── interfacce
│   ├── dataTypes.iol Definizione struttura messaggi XML
│   ├── interfacciaMagazzinoPrincipale.iol Definizione operation magazzino principale
│   ├── interfacciaMagazzinoSecondario.iol Definizione operation magazzino secondario
│   └── interfacciaFornitore.iol Definizione operation fornitore esterno
├── magazzino_principale
└── magazzino_secondario1
```

Le operazioni per il deployment del magazzino principale possono essere riassunte nei seguenti punti:

1. Configurazione del file `magazzinoPrincipale.conf`, in cui vengono impostati i parametri del database (si consiglia di lasciare inalterati i parametri i default).
2. Creazione del database tramite l'esecuzione di un programma Jolie:

```
$ jolie creaDBmagazzinoPrincipale.ol
```

3. (Opzionale) Popolamento database con dati di prova:

```
$ jolie popolaDBmagazzinoPrincipale.ol
```

4. Avvio del web service server del magazzino principale:

```
$ jolie magazzinoPrincipale.ol
```

Con le impostazioni di default, il magazzino principale ascolta all'indirizzo `localhost:8000`. Per modificare la porta d'ascolto è necessario cambiare l'attributo nel file di configurazione e nel sorgente del magazzino principale. Questo avviene dal momento che Jolie non permette di settare dinamicamente la location d'ascolto. In standard output viene stampato il log del servizio con alcune informazioni utili per eventuale debugging.

II. Deployment del magazzino secondario

La struttura generale dell'applicativo può essere sintetizzata nel seguente albero:

```
magazzino_secondario
├── magazzinoSecondario.ol Applicazione Jolie service server
├── creaDBMagazzinoSecondario.ol Crea il database "db/DB_magazzinoSec"
├── popolaDBMagazzinoSecondario.ol Popola il database con dati di prova
├── lib
│   └── hsql.jar DBMS hsql
└── db
    └── ...  File del DB hsql "DB_magazzinoSec"
```

Anche in questo caso è necessario che le directory `magazzino_secondario` e `interfacce` siano contenute nella stessa cartella.

Le operazioni per il deployment del magazzino secondario si possono riassumere nei seguenti punti:

1. Configurazione del file `magazzinoSecondario.conf`, in cui vengono impostati i parametri del database (anche in questo caso, si consiglia di lasciare inalterati i parametri i default).
2. Creazione del database tramite l'esecuzione di un programma Jolie:

```
$ jolie creaDBmagazzinoSecondario.ol
```

3. (Opzionale) Popolamento del database con dati di prova (appositi per il magazzino *n*-esimo, nell'esempio sotto si fa riferimento al secondo):

```
$ jolie popolaDBmagazzinoSecondario2.ol
```

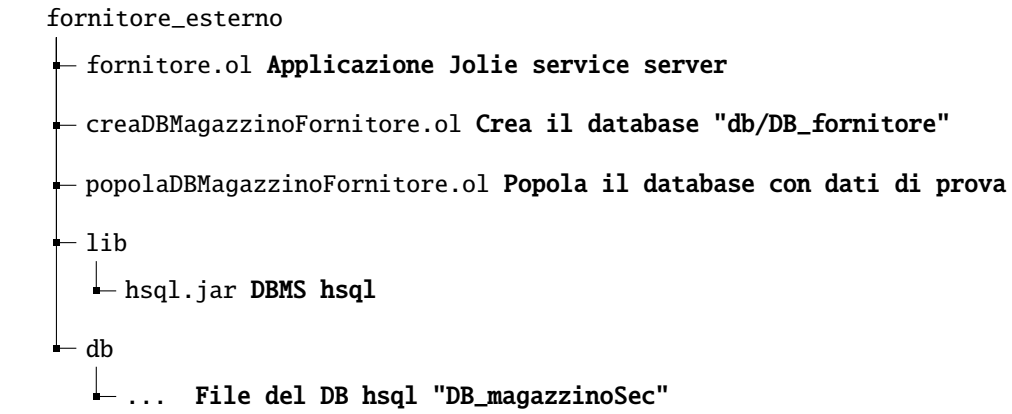
4. Avvio del web service server del magazzino secondario:

```
$ jolie magazzinoSecondario.ol
```

Per aggiungere un nuovo magazzino a tutto il sistema, è necessario creare una copia dell'applicativo di un magazzino secondario e modificare il file di configurazione, specificando l'id corrispondente alla nuova entità che si viene a creare. Ricordiamo che è necessario modificare anche l'input port nel sorgente `magazzinoSecondario.ol` con la location desiderata. In questa implementazione del sistema è necessario registrare nel database del magazzino principale un nuovo record nella tabella `Magazzini`, in modo che possa essere automaticamente considerato nell'esecuzione delle operazioni di ACME.

III. Deployment del fornitore esterno

La struttura generale del fornitore esterno può essere sintetizzata nel seguente albero:



Ancora una volta, è necessario che le directory `fornitore_esterno` e `interfacce` siano contenute nella stessa cartella, come spiegato sopra.

Le operazioni per il deployment del fornitore esterno si possono riassumere nei seguenti punti:

1. Configurazione del file `fornitore.conf`, in cui vengono impostati i parametri del database (anche in questo caso, si consiglia di lasciare inalterati i valori di default).
2. Creazione del database tramite l'esecuzione di un programma Jolie:

```
$ jolie creaDBmagazzinoFornitore.ol
```

3. (Opzionale) Popolamento del database con dati di prova:

```
$ jolie popolaDBmagazzinoFornitore.ol
```

4. Avvio del web service server del fornitore esterno:

```
$ jolie magazzinoFornitore.ol
```

Come per i magazzini secondari, il fornitore esterno deve essere aggiunto all'interno della tabella `Magazzini` nel database del magazzino principale.

IV. Deployment dell'amministrazione (processo Camunda)

Il deployment avviene mediante l'esportazione dell'applicazione in un archivio `war`, collocato poi nell'apposito application server. Per ottenerlo, aperto un terminale e collocatisi nella directory del progetto, è sufficiente eseguire l'istruzione:

```
$ mvn clean package
```

Il `war` generato è presente nella directory `target`. Notiamo come nella consegna siano presenti due diverse applicazioni per l'implementazione del business process, a causa di alcuni problemi che saranno discussi dettagliatamente in

sezione VI. `selling-cycles-no-ws-start-working` è l'applicativo che non consente l'avvio, tramite messaggio, del processo Camunda, ma permette l'esecuzione di tutti i task necessari alla corretta conclusione del processo. L'altro, al contrario, consente al processo Camunda di avviarsi con successo, ma fallisce durante l'esecuzione delle chiamata ai web service Jolie.

Per quanto riguarda `selling-cycles-no-ws-start-working`, l'avvio avviene manualmente attraverso la web gui di Camunda. I dati dell'ordine sono, a scopo di testing, semplicemente costruiti all'interno della service task "Processa ordine".

VI. Conclusioni e problematiche

La progettazione e realizzazione (tramite paradigma SOA) del sistema di ACME non è stata esente da problemi. In primo luogo, il linguaggio Jolie si presta bene all'esposizione di servizi, ma risulta molto macchinoso nella programmazione general purpose, a causa dell'assenza del concetto di "funzione", sostituito dalle "operations", e dello scope dinamico che ci ha costretto ad usare soluzioni non sempre eleganti. Segnaliamo inoltre alcuni problemi riscontrati durante lo sviluppo, nel momento in cui è stato necessario far comunicare fra loro entità realizzate con tecnologie differenti:

- Jolie ha dei problemi con gli "a capo" `<CR>` che client come SoapUI inseriscono nei messaggi XML. L'opzione `dropRootValue` risolve il problema **solo** per la radice del messaggio; in presenza quindi di tipi strutturati non primitivi la problematica persiste. Per questo, in programmi con messaggi semplici (come la calcolatrice vista a lezione) ciò non accade. Nel nostro caso, invece, a quanto pare è presente un bug non ancora fixato che ne impedisce il corretto funzionamento. Riportiamo il link alla discussione originale [6].
- È stato riscontrato che, realizzando un client Java per un servizio Jolie, le operation "oneway" bloccano l'esecuzione del codice. Questo problema non è stato approfondito e non è noto se sia dovuto alla generazione automatica a partire dal WSDL oppure se si tratta di una vera e propria incompatibilità tra Java e Jolie.

In teoria, sarebbe stato necessario usare JBoss poiché supporta le primitive che consentono, ricevendo un'invocazione all'interno di un web service, di avviare un processo Camunda. Tuttavia in questo application server non si è riusciti ad effettuare le chiamate ai servizi Jolie necessarie per il completamento di tutte le operazioni. Il problema sembra dipendere dalle mancate dipendenze richieste dal web service client auto-generato a partire dal WSDL con Apache CXF. Alcuni test ci hanno permesso di identificare il problema proprio nell'auto-generazione a partire da quel determinato WSDL ottenuto dal codice Jolie. Le chiamate verso altri server, infatti, funzionano correttamente. Nel codice presentato, questo problema è stato aggirato utilizzando il web server Tomcat, il quale, pur senza realizzare l'auto-avvio del processo Camunda a seguito di un messaggio, permette di simulare l'esecuzione del processo business.

Concludendo, tramite la realizzazione e lo studio delle attività di ACME, è stato possibile apprezzare a pieno i vantaggi del paradigma service-oriented. Contemporaneamente, siamo rimasti alquanto delusi da come gli standard risentano delle diverse implementazioni, portando talvolta a fastidiose incompatibilità che rappresentano un'incongruenza rispetto alla suddetta filosofia.

[6] Discussione bug `www.github.com/jolie/jolie/issues/44`