Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

PROGETTAZIONE E REALIZZAZIONE DI UN EDITOR WEB-BASED PER IL LINGUAGGIO JOLIE

Tesi di Laurea in Linguaggi di Programmazione

Relatore:
Prof. Maurizio Gabbrielli

Presentata da:
FEDERICO ROFFI

Correlatore:
Dott. Claudio Guidi

Terza Sessione
Anno Accademico 2009/10

Alla mia famiglia

Introduzione

Il *service-orientation* è un paradigma emergente per lo sviluppo di sistemi distribuiti basati su *servizi*. I servizi possono essere composti al fine di progettare servizi più complessi, sfruttando *orchestratori*. Gli orchestratori sono entità in grado di richiamare e coordinare gli altri servizi, sfruttando modelli di composizione di *workflow*.

In questo ambito troviamo il progetto JOLIE, un linguaggio di programmazione che permette l'*orchestration* di servizi. Il progetto è open-source ed è nato presso l'Università di Bologna. I punti di forza di questo linguaggio sono il linguaggio formale su cui è basato e una sintassi simile al C, già nota alla maggior parte degli sviluppatori.

Questa tesi è stata svolta nel contesto del progetto *jEye*, un editor visuale web-based per il linguaggio JOLIE, di cui ho continuato lo sviluppo cominciato durante la mia attività di tirocinio presso Italiana Software S.r.l.. L'azienda è stata fondata dai due principali sviluppatori del linguaggio JOLIE, che lo utilizzano per scopi commerciali e ne continuano lo sviluppo.

La fase iniziale di questo documento è incentrata sulla *Service-Oriented Architecture* (SOA), in cui sono approfonditi gli aspetti chiave della service-orientation. Saranno definite le caratteristiche dei servizi e dei sistemi service-oriented, a cui seguirà un approfondimento la tecnologia Web Service, che rappresenta attualmente la scelta più diffusa per l'implementazione di sistemi distribuiti. In conclusione a questo capitolo sono illustrati i vantaggi di un'architettura service-oriented.

Il secondo capitolo illustra le caratteristiche del linguaggio JOLIE, ana-

lizzandone la sintassi e l'architettura.

Il terzo capitolo è interamente dedicato al progetto jEye, il cui principale obiettivo è fornire uno strumento innovativo di progettazione di workflow JOLIE. Saranno illustrate le fasi di progettazione e d'implementazione di questo editor visuale, ponendo particolare attenzione sulla struttura dell'applicazione e sulle caratteristiche delle tecnologie utilizzate.

Indice

Introduzione **i**

1 Service-Oriented Architecture **1**

1.1 Servizi 1

1.1.1 Come comunicano i servizi 2

1.1.2 Come progettare i servizi 2

1.2 Web service 3

1.2.1 Gli standards fondamentali dei Web service 3

1.3 I vantaggi della SOA 4

1.3.1 Migliore integrazione 4

1.3.2 Riutilizzo 5

1.3.3 Architetture e soluzioni essenziali 5

1.3.4 Sfruttamento degli investimenti legacy 5

1.3.5 Investimenti mirati sulle infrastrutture di comunicazione 6

1.3.6 Agilità organizzativa 6

2 JOLIE **7**

2.1 Il linguaggio 8

2.1.1 Identifier 8

2.1.2 La struttura di un programma 8

2.1.3 Statement 10

2.1.4 Statements composers 11

2.2 L'architettura dell'interprete JOLIE 12

2.2.1 Il Parser e l'Object Oriented Interpretation Tree 13

2.2.2 Comminication Core 13

3 jEye **15**

3.1 Obiettivi e specifiche 16

3.2 La struttura 16

3.3 Parte server 17

3.3.1 Leonardo 17

3.3.2 WorkflowManager 17

3.4 Parte client 18

3.4.1 Google Web Toolkit 19

3.4.2 Jolie Visual Model 22

3.4.3 Visitor pattern 33

Conclusioni **37**

Bibliografia **39**

Elenco delle figure

2.1 JOLIE architecture [Gui07] 12

3.1 Sequence 24
3.2 Parallel 25
3.3 LogActivity 26
3.4 IfThenElse 27
3.5 Loop 27
3.6 For 28
3.7 Notification 29
3.8 OneWay 30
3.9 RequestResponse 31
3.10 SolicitResponse 31
3.11 Condition 32
3.12 DataCreation 33
3.13 Spostamento di una Activity 34
3.14 Visualizzazione del codice JOLIE 36

Capitolo 1

Service-Oriented Architecture

La *Service-Oriented Architecture* (SOA) [GeoSch09] è un paradigma per la progettazione di applicazioni distribuite in ambienti eterogenei che si basa sull'ultilizzo di **servizi**.

L'obiettivo della SOA è la creazione di applicazioni composte da servizi cooperativi e velocemente adattabili a cambiamenti di requisiti.

Il modello *service-oriented* [Tho05] incoraggia la crescita autonoma, ma non isolata delle singole componenti. A tale scopo ognuna di esse deve essere conforme a un insieme di principi che permettono di mantenere un livello sufficiente di omogeneità e standardizzazione.

1.1 Servizi

All'interno delle SOA i servizi [Jos07] possono essere utilizzati da altri servizi o da altri programmi. Per interagire, i servizi devono essere a conoscenza degli altri servizi attraverso l'uso delle **service descriptions**.

Una service description nel suo formato più semplice stabilisce il **nome** del servizio e i **dati** attesi e restituiti dal servizio. Il modo in cui i servizi utilizzano le service descriptions si traduce in un rapporto classificato come *loosely coupled*.

1.1.1 Come comunicano i servizi

Quando un servizio manda un messaggio, perde subito il controllo su cosa succederà al messaggio stesso. Per questo motivo è importante che i messaggi siano unità indipendenti di comunicazione, quindi autonomi e dotati di sufficiente intelligenza per gestire le proprie parti del *processing logic*.

I servizi che forniscono service descriptions e comunicano tramite messaggi formano una architettura base. Quello che distingue un'architettura distribuita da una architettura service-oriented è come le tre componenti fondamentali (servizi, service descriptions, e messaggi) vengono progettate.

1.1.2 Come progettare i servizi

In modo del tutto simile all'object-orientation, il service-orientation è diventato un approccio di progettazione che introduce principi che stabiliscono il posizionamento e la progettazione delle componenti architetturali.

L'applicazione dei principi del service-orientation nel processing logic permette di ottenere processing logic service-oriented standardizzati.

Gli aspetti chiave di questi principi sono:

Loose coupling : i servizi mantengono relazioni che minimizzano le dipendenze e che richiedono solo la conoscenza l'uno dell'altro;

Service contract : i servizi aderiscono ad un accordo di comunicazione, come definito da uno o più service descriptions e dai documenti relativi;

Autonomia : i servizi hanno il controllo sulla logica che contengono;

Astrazione : i servizi nascondono la logica alle componenti esterne;

Riusabilità : la logica è suddivisa in servizi con l'intento di incoraggiare il riutilizzo;

Composability : insiemi di servizi possono essere coordinati e assemblati per formare servizi composti (composite services);

Statelessness : i servizi mantengono la quantità minima di informazioni specifiche ad una attività;

Discoverability : i servizi vengono progettati per essere raggiungibili attraverso meccanismi di ricerca.

1.2 Web service

Web service (WS) [Tho05] è una architettura che permette ad applicazioni di parlare l'una con l'altra e definisce un insieme di standard che favorisce l'interoperabilità.

I WS offrono la possibilità di soddisfare i requisiti necessari per implementare una SOA, a patto che vengano appositamente progettati per farlo. Questo framework è molto flessibile e lo si può utilizzare anche per implementare le componenti di un sistema distribuito proprietario.

1.2.1 Gli standards fondamentali dei Web service

Esistono cinque standard riguardanti i WS. Due sono standard generali:

XML : utilizzato come formato generale per descrivere modelli, formati e tipi. Molti altri standard sono standard XML, come ad esempio XML 1.0, XML Schema Definition e XML namespace;

HTTP(S) : è il protocollo di basso livello usato per Internet, ed è uno dei più comuni protocolli usati per l'invio di Web services in rete.

Gli altri tre standard fondamentali sono specifici ai WS:

WSDL (*Web Service Description Language*): permette di definire interfacce di servizi. Può descrivere tre aspetti distinti di un servizio: la sua *signature* (nome e parametri), il suo *binding* (protocollo) e i dettagli di *deployment* (location).

SOAP (*Simple Object Access Protocol*): definisce il protocollo utilizzato. Mentre HTTP è il protocollo di basso livello, utilizzato anche per Internet, SOAP è uno specifico formato per lo scambio dei dati dei WS sopra questo protocollo.

UDDI (*Universal Description, Discovery, and Integration*): è uno standard per la gestione di WS, come ad esempio la registrazione e la ricerca di servizi.

Utilizzare lo standard WSDL è generalmente la caratteristica chiave dei WS. SOAP e HTTP non sono, infatti, gli unici standard utilizzati per inviare richieste a servizi, mentre UDDI non è indispensabile e ricopre un ruolo secondario.

1.3 I vantaggi della SOA

La SOA porta benefici organizzativi in diversi modi, secondo gli obiettivi e la maniera in cui la SOA e le sue tecnologie vengono applicate. Possiamo analizzare alcuni vantaggi comuni che questa piattaforma architetturale ha da offrire.

1.3.1 Migliore integrazione

Una SOA può portare alla creazione di soluzioni composte da servizi intrinsecamente interoperabili. Utilizzare soluzioni di questo tipo è parte della *Service-Oriented Integration* e porta alla creazione di una *Service-Oriented Integration Architecture*. Grazie a framework di comunicazioni neutrali, le imprese hanno la possibilità di implementare service description e message structure altamente standardizzate. L'interoperabilità intrinseca ottenuta trasforma quindi un progetto d'integrazione di cross-application in un esercizio di modellazione e riduce lo sforzo di sviluppo personalizzato.

1.3.2 Riutilizzo

Il service-orientation promuove la progettazione di servizi che sono intrinsecamente riutilizzabili, permettendo così di sfruttare meglio l'*automation logic* esistente. Sviluppare soluzioni service-oriented in modo che i servizi soddisfano immediatamente i requisiti del livello applicazione, pur supportando un livello di riusabilità da potenziali futuri richiedenti, stabilisce un ambiente in cui gli investimenti in sistemi esistenti possono venire sfruttati per la costruzione di nuove soluzioni. Anche se sviluppare servizi intrinsecamente riutilizzabili richiede uno sforzo maggiore e l'utilizzo di standard di progettazione, la possibilità di riutilizzare servizi riduce costi e sforzi di costruzione di nuove soluzioni service-oriented.

1.3.3 Architetture e soluzioni essenziali

Il concetto di composizione è una parte fondamentale della SOA e non si limita all'aggregazione di insiemi di servizi. Questo aspetto delle SOA può infatti portare alla costruzione di ambienti di automazione ottimizzati, in cui solo le tecnologie richieste vanno a far parte dell'architettura. I benefici di architetture e soluzioni essenziali sono la riduzione di elaborazioni e di requisiti tecnici necessari per gestire applicazioni, servizi ed estensioni di servizi.

1.3.4 Sfruttamento degli investimenti legacy

L'accettazione del settore delle tecnologie dei WS ha generato un grande mercato, consentendo ad ambienti *legacy* di partecipare a *Service-Oriented Integration Architecture*. Ambienti che prima rimanevano isolati ora possono interoperare senza la necessità di sviluppare costosi e a volte fragili canali di integrazione specifici. Sebbene ci siano ancora rischi legati principalmente a come sistemi legacy devono fare fronte ad un volume di utilizzo maggiore, l'abilità di utilizzare ciò che si ha già a disposizione con delle soluzioni service-oriented attuali e future è molto interessante. In questo modo la necessità

di sostituire sistemi legacy diminuisce ed il costo della loro integrazione in soluzioni attuali viene ridotto.

1.3.5 Investimenti mirati sulle infrastrutture di comunicazione

Poiché i WS stabiliscono standard di comunicazione, una SOA può centralizzare applicazioni di comunicazione interne ed esterne. Questo consente alle organizzazioni di espandere l'infrastruttura investendo in una sola tecnologia responsabile per la comunicazione, riducendo così i costi.

1.3.6 Agilità organizzativa

L'agilità è una qualità inerente a ogni aspetto di un'azienda. Esiste una relazione diretta tra l'agilità di un'intera organizzazione e come ogni sua componente viene costruita, posizionata ed utilizzata. Gran parte della service-orientation è basata sull'assunzione che quello che viene costruito oggi subirà cambiamenti in futuro, pertanto una SOA ben progettata deve proteggere le organizzazioni dall'impatto con l'evoluzione. I cambiamenti possono essere distruttivi, costosi e potenzialmente dannosi per organizzazioni rigide. Servizi interoperabili, standardizzati, loosely coupled, interoperabili e potenzialmente riutilizzabili compongono un ambiente che è più facilmente adattabile ai cambiamenti. Inoltre, astraendo la business logic e la tecnologia in livelli specializzati di servizi, SOA può stabilire una relazione loosely coupled tra questi due domini.

Capitolo 2

JOLIE

In un sistema basato sulla SOA i servizi possono essere composti tra loro al fine di progettare servizi più complessi [MonGui06]. Sono stati ideati due approcci con lo scopo di ridurre la complessità di connessione fra servizi: l'*orchestration* e la *choreography*. Gli orchestrator sono in grado di richiamare e coordinare altri servizi sfruttando tipici modelli di workflow come la composizione parallela, sequenziale e di scelta. Diverso invece è l'approccio della choreography, che permette di progettare un sistema distribuito dall'alto.

Attraverso l'analisi di entrambi gli approcci è emerso che l'orchestration rappresenta un passo in avanti verso il perfezionamento della progettazione di applicazioni service-oriented. Anche se la choreography non produce sistemi eseguibili, l'orchestration permette di gestire ogni servizio coinvolto nell'applicazione.

JOLIE (*Java Orchestration Language and Interpreter Engine*) [Gui07] è un linguaggio open source di orchestrazione basato sul paradigma di programmazione service-oriented e offre la possibilità di creare nuovi servizi da zero o di comporre quelli esistenti per ottenere nuove funzionalità.

2.1 Il linguaggio

JOLIE fornisce una sintassi simile al C, che rende il linguaggio intuitivo e semplice da imparare a programmatori abituati a questa sintassi. Introduciamo di seguito le nozioni base del linguaggio.

2.1.1 Identifier

Gli Identifiers (a volte abbreviato come *id*) sono nomi non ambigui conservati nella memoria condivisa dell'orchestrator che identificano una location, un operatore, una variabile o un link.

2.1.2 La struttura di un programma

La struttura di un programma JOLIE è rappresentato dalla seguente grammatica:

```
program::=
locations { Locations-definition* }
operations { Operations-declaration* }
variables { Variables-declaration }
links { Links-declaration }
definition*
main { Process }
definition*
definition:=
define id { Process }
```

2.1.2.1 Location

Le comunicazioni sono basate sulle socket. Un orchestrator, infatti, attende messaggi su una porta di rete. Per comunicare con un altro orchestrator è necessario conoscere il suo hostname (o indirizzo ip) e la porta su cui è in ascolto: queste informazioni vengono conservate in una location.

2.1.2.2 Operation

Le operation rappresentano il modo in cui gli orchestrator interagiscono con altri orchestrator. Distinguiamo due tipi di operation: **Input operation** e **Output operation**.

Le *Input operation* rappresentano i punti di accesso che un orchestrator offre per comunicare con esso. Possiamo distinguerle in:

- **One-Way**: rimane in attesa della ricezione di un messaggio;
- **Request-Response**: rimane in attesa di un messaggio, esegue un blocco di codice e poi invia il messaggio di risposta a chi l'ha invocato.

Le *Output operation* sono utilizzate per invocare le operazioni di input di un altro orchestrator. Possiamo distinguerle in:

- **Notification**: permette di invocare una One-Way di un altro orchestror;
- **Solicit-Response**: permette di invocare una Request-Response. Dopo aver inviato il messaggio di richiesta, rimane bloccato finché non riceve la risposta del servizio invocato.

2.1.2.3 Variabili

Le variabili in JOLIE non sono tipate. Stringhe ed interi sono implicitamente supportati. La dichiarazione di variabili non-terminal richiede solo una lista di identificatori che rappresentano le variabili della memoria condivisa.

2.1.2.4 Link

I link vengono utilizzati per la sincronizzazione di processi paralleli. Come per le variabili, la dichiarazione di link di non terminazione richiedono solo una lista di identificatori.

2.1.2.5 Main

Il blocco main permette di definire il processo che verrà eseguito all'inizio dell'esecuzione del programma. Lo si può paragonare alla funzione main di un programma scritto in C.

2.1.3 Statement

Statement di controllo:

- *if (condition)* {...} *else if (condition)* {...} *else* {...} : statement di condizione;
- *while (condition)* {...} : statement di ciclo.

Statement delle Operation:

- *id<id list>* : attende un messaggio dall'operazione One-Way *id*, e registra il suo valore nelle variabili *id list*;
- *id<id list><id list>(Process)* : attende un messaggio dall'operazione RequestResponse *id*, registra il suo valore nelle variabili *id list*, esegue il codice del blocco *Process* e invia un messaggio di risposta contenente il valore delle variabili della seconda *id list*;
- *id@id<id list>* : utilizza l'operation Notification identificata dal primo *id* per spedire il messaggio che contiene i valori delle variabili *id list* all'orchestrator corrispondente al secondo *id*;
- *id@id <id list><id list>* : utilizza l'operation Solicit-Response identificata dal primo *id* per spedire il messaggio che contiene i valori delle variabili *id list* all'orchestrator corrispondente al secondo *id*. Una volta inviato il messaggio, rimane attesa la risposta dalla Request-Response invocata e i suoi valori sono conservati nelle variabili identificate dal secondo *id list*;

Statement di sincronizzazione:

- *linkIn(id)* : attende il segnale di un linkOut associato allo stesso identificatore interno *id*;
- *linkOut(id)* : manda un segnale di sincronizzazione ad un linkIn associato allo stesso identificatore interno *id*. Se ci sono più processi in attesa sullo stesso link interno, si sincronizza con uno di loro seguendo una politica non deterministica.

Altri statement:

- *sleep(n)*: blocca l'esecuzione del processo corrente per n (numero naturale) millisecondi;
- *nullProcess*: operazione nulla.

2.1.4 Statements composers

JOLIE fornisce tre modi per comporre gli statement: in sequenza, in parallelo e nei rami di una scelta non deterministica.

2.1.4.1 Sequence

Una sequenza é composta utilizzando l'operatore *;* .

2.1.4.2 Parallel

I processi sono composti in parallelo utilizzando l'operatore —— , che combina un insieme di Sequence. L'esecuzione della Parallel termina quando tutte le Sequence terminano.

2.1.4.3 Non-deterministic choice

Si può esprimere in diversi modi utilizzando l'operatore *++* attraverso questa sintassi: *[g1]p1 ++ [g2]p2 ++ ... ++[gn-1]pn-1 ++ [gn]pn*. Il controllo del ramo può essere solo una operazione di input, mentre il ramo può

essere qualsiasi processo. Raggiunta una Non-deterministic choice, l'interprete ferma l'esecuzione in attesa di input in uno dei suoi controlli. Quando arriva un input, il processo associato a quel controllo viene eseguito e gli altri rami disattivati.

2.2 L'architettura dell'interprete JOLIE

L'algoritmo dell'interpretazione di JOLIE e la parti che compongono l'interprete sono illustrate nella seguente figura.

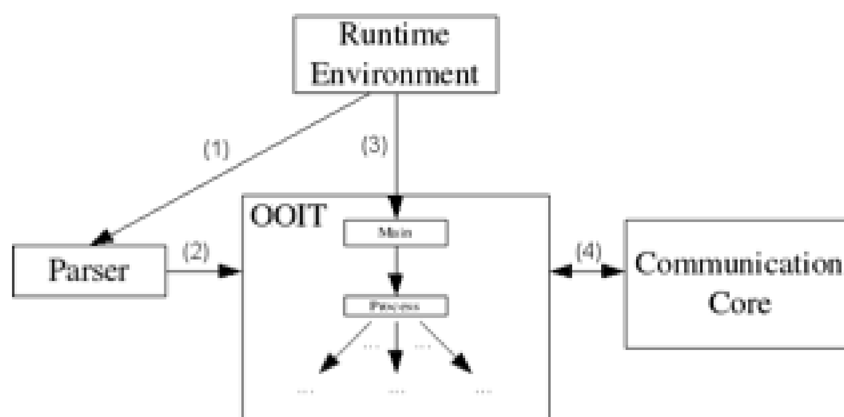

Figura 2.1: JOLIE architecture [Gui07]

Per spiegare come funziona JOLIE descriviamo gli step principali dell'ambiente a tempo di esecuzione e le sue componenti principali: il *Parser*, l'*Object Oriented Interpretation Tree* (OOIT) e il *Communication Core*.

2.2.0.4 Il comportamento dell'interprete

1. inizializzazione del Communication Core;

2. creazione di una istanza del Parser;

3. crezione dell'OOIT;

4. invocazione del metodo run() del nodo radice dell'OOIT (che corrisponde al main).

2.2.1 Il Parser e l'Object Oriented Interpretation Tree

JOLIE è basato su un'infrastruttura Object-Oriented creata durante il parsing dell'orchestration da eseguire e realizzata da un parser a discesa ricorsiva. Per capire com'è realizzato, introduciamo gli elementi principali presenti nell'OOIT.

2.2.1.1 Process

Process è una classe che rappresenta un pezzo generico di codice JOLIE e fornisce un metodo run() che esegue le attività che l'oggetto rappresenta.

2.2.1.2 Basic Process

Il Basic Process è un Process composto da un solo statement (ad esempio una operazione di assegnamento). In questo caso il metodo run() esegue lo statement.

2.2.1.3 Composite Process

Un Composite Process è un Process composto di altri oggetti Process (attraverso gli statements Parallel, Sequence o Non-deterministic choice). Il metodo run() di un Composite Process richiama i metodi run() dei Process racchiusi. Grazie a questo incapsulamento per cominciare l'esecuzione è sufficiente richiamare il metodo run() del nodo radice.

2.2.2 Comminication Core

Il Communication Core fornisce una interfaccia di comunicazione fra servizi e permette di astrarre dal *protocollo* e dal *mezzo di comunicazione*. Queste astrazioni sono supportate attraverso un canale di comunicazione *Comm-*

Channel. L'ambiente a tempo di esecuzione permette ai canali di comunicazione di inviare e ricevere dati. Una volta istanziato, un oggetto CommChannel può inviare e ricevere oggetti *CommMessage* che sono composti dal nome dell'Operation e da una array di valori.

Capitolo 3

jEye

jEye è un editor web-based per la progettazione di *workflow* basati su servizi. Il progetto è nato con l'intento di creare uno strumento innovativo e multipiattaforma rivolto a diversi tipi di utenti. Accedendo tramite browser all'interfaccia web di jEye gli utenti possono progettare in maniera visuale workflow che verranno convertiti su richiesta in codice JOLIE.

Gli utenti inesperti saranno guidati dall'interfaccia stessa nella composizione di nuovi workflow, poiché le uniche operazioni possibili sono quelle che permettono la progettazione di un workflow che genera un codice sintatticamente corretto. Per fare questo ovviamente è stato necessario ridurre la flessibilità e la libertà di composizione concessa all'utente, fornendo un numero finito di operazioni e ponendo alcuni limiti strutturali che non si avrebbero scrivendo a mano in linguaggio JOLIE.

jEye inoltre permette di velocizzare molte operazioni di progettazione, rivelandosi uno strumento utile anche per utenti esperti nel campo della programmazione.

In questo capitolo si analizzano le fasi di progettazione e realizzazione del progetto, descrivendone prima la struttura generale per poi entrare nei dettagli di ogni componente.

3.1 Obiettivi e specifiche

Durante la fase di analisi dei requisiti di jEye sono stati fissati i seguenti obiettivi:

- l'interfaccia di progettazione di workflow deve essere utilizzabile su computer che non dispongono di strumenti specifici di programmazione;
- la fase di progettazione deve essere semplice, intuitiva e il numero di digitazioni da tastiera ridotto al minimo;
- i workflow progettati devono poter essere convertiti in linguaggio Jolie in maniera semplice;
- la struttura dell'applicazione deve prevedere l'introduzione futura di nuove funzionalità.

3.2 La struttura

Dalla prima fase di analisi è emerso che per garantire la massima espandibilità sarebbe stato necessario implementare l'architettura separando la componente di interfaccia dell'editor (**parte client**) da quella di conversione ed elaborazione (**parte server**).

Come approfondito nel secondo capitolo, JOLIE è stato ritenuto il linguaggio adatto a implementare tutte le funzionalità che andranno a comporre la **parte server**. Le caratteristiche d'interoperabilità e aggregazione del linguaggio hanno permesso la realizzazione di un'architettura composta da un insieme di servizi dal ruolo specifico.

Vista la natura service-oriented del linguaggio JOLIE si è ricercato un *framework Ajax* che consentisse di implementare una **parte client** accessibile da browser. L'utilizzo di un framework Ajax, e quindi l'implementazione di un'interfaccia web-based, ha permesso di soddisfare due importanti obiettivi: richiedere agli utenti un browser (fornito da qualsiasi sistema operativo

attuale) come unico strumento per l'accesso all'interfaccia di progettazione e rendere l'applicazione multipiattaforma.

Nelle seguenti sezioni sono approfonditi i dettagli più importanti delle due componenti di jEye: la parte server e la parte client.

3.3 Parte server

Grazie alle funzioni di aggregazione di JOLIE è possibile accedere a servizi appartenenti ad altri servizi JOLIE e renderli disponibili come propri scrivendo poche righe di codice. Questa caratteristica ha permesso un'implementazione flessibile della parte server, creando un insieme aggregato di servizi specifici.

Il servizio principale si chiama **Leonardo** e svolge la funzione di web server. Attraverso Leonardo è possibile accedere al **WorkflowManager**, che richiama altri sotto-servizi necessari alla progettazione e alla traduzione.

Nei paragrafi seguenti vengono descritti i servizi Leonardo e WorkflowManager.

3.3.1 Leonardo

Leonardo è il web server di jEye. Visitando con un browser l'indirizzo descritto dalla *Location* associata all'interfaccia *HTTPInterface* si può accedere alle risorse disponibili. Leonardo gestisce ogni richiesta HTTP, permettendo il caricamento di pagine web e delle sue risorse, consentendone la corretta visualizzazione.

3.3.2 WorkflowManager

Il *WorkflowManager* consente l'accesso a metodi di supporto alla progettazione e al servizio di traduzione.

Il metodi di supporto alla progettazione sono **getRoles** e **getRole** e permettono di richiedere informazioni relative alla composizione di statement

di operation. getRoles restituisce la lista dei ruoli (Role) disponibili. A ogni ruolo è associata una lista di operazioni (Operation). Richiamando il metodo getRole e passando come parametro uno dei nomi dei ruoli disponibili verrà restituita un messaggio contenente la descrizione delle operazioni, dei dati di input e di output e dei loro tipi.

Il servizio di traduzione fornisce il metodo **parseWorkflow**. Questo metodo richiede come parametro di input una struttura chiamata **WorkflowTree** e restituisce in output la stringa contenente la traduzione in linguaggio JOLIE.

Nel seguente paragrafo analizziamo in dettaglio in cosa consiste un WorkflowTree.

3.3.2.1 WorkflowTree

Un *WorkflowTree* è una struttura ad albero che rappresenta un workflow JOLIE. Il nodo radice è di tipo Workflow, a cui è possibile innestare altri nodi che rappresentano statement del linguaggio JOLIE. I sottonodi possono essere composti a loro volta da altri nodi. I tipi del WorkflowTree specificano quali dati sono necessari per la corretta traduzione in uno statement JOLIE e quali sottonodi possono contenere.

Ecco un esempio di un semplice WorkflowTree:

```
1 Value
2   .activity[0]
3     .library[0]
4       .logactivity[0]
5         .message[0] = hello world! : java.lang.String
6   .name[0] = Sequence1298845497210 : java.lang.String
```

3.4 Parte client

Le versioni attuali dei framework Ajax (*Asynchronous JavaScript and XML*) permettono la realizzazione di applicazioni web caratterizzate da un

livello di interazione che in passato era ottenibile solo realizzando un'applicazione tradizionale. Questo è reso possibile dallo scambio di dati in background fra browser e server, che consente l'aggiornamento dinamico di una pagina web. Un noto problema delle applicazioni Ajax è la compatibilità. L'uso di un valido framework Ajax permette però di risolvere quasi totalmente questo problema, poiché sono forniti un insieme di funzioni sviluppate e testate per funzionare sui browser più diffusi.

Uno dei problemi principali emersi durante la fase di progettazione della parte client è la necessità di creare una complessa interfaccia utente che permettesse di formare un WorkflowTree privo di inconsistenza dei dati destinato al server. Per questo motivo si è pensato che la soluzione più efficace è il salvataggio delle informazioni necessarie alla creazione del WorkflowTree direttamente all'interno degli elementi grafici di cui è composta l'interfaccia web dell'editor. In questo modo le informazioni sono reperibili visitando l'intera struttura che descrive il workflow. E' risultata evidente la necessità di adottare un **Visitor pattern**, che oltre a separare l'algoritmo dalla struttura a cui è applicato permette di aggiungere nuove operazioni senza dover modificare la struttura stessa.

Nel seguente paragrafo sono illustrate le caratteristiche del framework Ajax *Google Web Toolkit* (GWT) e dei motivi per cui è stato scelto per l'implementazione di jEye, mentre nei paragrafi successivi verranno descritti alcuni dettagli implementativi importanti riguardanti l'interfaccia utente e la comunicazione con la parte server.

3.4.1 Google Web Toolkit

Sviluppare un'applicazione web compatibile con browser diversi è un processo complicato e incline agli errori. Inoltre scrivere, riutilizzare e mantenere grandi porzioni di codice JavaScript e componenti Ajax può essere molto difficile.

Google Web Toolkit (GWT) [Gwt10] facilita questo compito, consentendo agli sviluppatori di creare e gestire applicazioni JavaScript complesse ma

comunque altamente performanti.

L'SDK di GWT permette di scrivere applicazioni in linguaggio *Java* che verranno cross-compilate in linguaggio JavaScript ottimizzato e compatibile con la maggior parte dei browser. Durante la generazione del codice JavaScript il compilatore di GWT esegue analisi e ottimizzazioni (ad esempio l'eliminazione di porzioni di codice e parametri inutilizzati) che permettono di ottenere spesso un codice più veloce di uno equivalente scritto a mano.

Siccome GWT utilizza Java, è possibile verificare e correggere errori sui tipi e di battitura mentre si scrive il codice. Inoltre è possibile aumentare la produttività utilizzando funzioni di *refactoring* e di *suggerimento/completamento* del codice forniti da diversi IDE. In particolare esiste un completissimo plugin per Eclipse che fornisce un insieme di strumenti per facilitare lo sviluppo, la correzione di errori e il deploy di applicazioni GWT.

Una caratteristica fondamentale di GWT è che il compilatore genera file JavaScript indipendenti che potranno essere utilizzati su diversi tipi di server, senza nessun bisogno di Java, in quanto vengono forniti diversi protocolli di comunicazione.

3.4.1.1 Libreria jolie-gwt

jolie-gwt è una libreria GWT che permette lo scambio di dati fra una applicazione GWT ed un servizio Jolie.

I dati scambiati sono composti attraverso l'utilizzo delle classi *Value* e *ValueVector* che consentono la creazione di complesse strutture ad albero contenenti valori di tipo *String*, *Integer* e *Double*.

La comunicazione avviene attraverso l'implementazione di una *JolieCallback* la sua chiamata:

```
1 // Preparazione del messaggio
2 Value v = new Value();
3 // Invocazione dell'operation "prova"
4 JolieService.Util.getInstance().call("prova", v, new JolieCallback() {
5
```

```
 6    public void onSuccess(Value ret) {
 7      // ret è il Value che contiene il messaggio di risposta.
 8      // Si può accedere a resMessage nel seguente modo:
 9      v.getFirstChild("reqMessage").strValue();
10    }
11    public void onError(Throwable t) {
12      // Codice da eseguire in caso di errore di connessione.
13    }
14    public void onFault(FaultException fault) {
15      // Codice da eseguire in caso di fault proveniente da Jolie.
16    }
17  });
```

3.4.1.2 GWT e jEye

Già nella prima fase di analisi GWT è risultato il framework ideale per l'implementazione della parte client dell'editor jEye.

Le applicazioni GWT sono compatibili non solo con i browser più diffusi, ma anche con i browser dei dispositivi mobili Android e iOS.

La comunicazione fra server Jolie e client GWT é agevolata dall'utilizzo della libreria jolie-gwt. Analogamente ad altri framework Ajax, la comunicazione client-server sarebbe potuta avvenire anche attraverso richieste HTTP, ma sarebbe stata necessaria la definizione di un protocollo di comunicazione più complesso in JSON o XML.

A livello di codice, lo sviluppo di applicazioni web dalla struttura complessa come jEye viene notevolmente agevolata dall'utilizzo di GWT. In Java, infatti, l'adozione di un Visitor pattern è molto semplice e grazie al supporto di un valido IDE risultano immediate anche operazioni complesse come il refactoring e l'introduzione di nuovi elementi.

3.4.2 Jolie Visual Model

Per rappresentare la struttura di un workflow Jolie in jEye è stato necessario elaborare un modello di rappresentazione visuale del linguaggio che prende il nome di **Jolie Visual Model**.

E' stato attribuito il nome di **Activity** all'insieme di elementi grafici che rappresentano gli statement composers (**Sequence** e **Parallel**), gli statement di controllo (**IfThenElse**, **For** e **Loop**), gli statement delle operation (**Notification**, **OneWay**, **SolicitResponse** e **RequestResponse**), un insieme di istruzioni che hanno lo scopo di dichiarare variabili (**DataCreation**) e una istruzione di logging con stampa su terminale (**LogActivity**).

Gli elementi **DatumInput** e **DatumOutput** rappresentano rispettivamente l'input e l'output dei dati comuni alle Activity che rappresentano le operation, e vengono inseriti all'interno di tali Activity.

Analogamente l'elemento **Condition** rappresenta un'espressione logica che viene inserita negli Activity di controllo IfThenElse e Loop, permettendo così l'elaborazione della condizione necessaria alla composizione di queste due Activity.

Per evitare che la composizione risultasse disordinata e poco chiara, la struttura generale del workflow viene composta annidando graficamente gli elementi. Ogni annidamento corrisponde nella maggior parte dei casi ad un blocco di codice JOLIE racchiuso da parentesi graffe.

Oltre alla rappresentazione dei singoli elementi si è dovuto pensare all'introduzione dei controlli necessari per l'eliminazione e lo spostamento degli elementi, operazioni che scrivendo codice con un editor tradizionale si effettuerebbero da tastiera attraverso i comandi taglia/incolla.

Nei seguenti paragrafi sono descritti nel dettaglio i singoli elementi grafici, alcune traduzioni in codice Jolie e quali informazioni sono necessarie per la composizione dei corrispondenti tipi JOLIE che sono inseriti nel WorkflowTree.

3.4.2.1 Workflow

Workflow è il nodo radice della struttura ad albero che rappresenta ogni workflow creato con jEye.

Al caricamento di jEye viene inserito all'interno del corpo centrale della pagina, mostrando solo il controllo della Sequence che contiene.

All'interno di questo elemento sono salvati gli elementi e le informazioni temporanei di supporto ad alcuni visitor.

Codice Jolie corrispondente al Workflow:

```
include "console.iol"
include "standard_types.iol"
execution{ concurrent }

init {
  loopindex = 0;
  println@Console("Service␣TEST␣is␣running...")()
}

main {

}
```

Tipo Workflow:

```
type Workflow: void {
  .name: string
  .roles*: string
  .activity*: Activity
}
```

3.4.2.2 Sequence

Sequence è l'elemento che rappresenta lo statement di sequenza.

Cliccando sul bottone *Add Activity* comparirà il menu di scelta delle Activity che possono essere aggiunte alla Sequence selezionata. Siccome ogni riga della Sequence può espandersi orizzontalmente e verticalmente in seguito a ripetuti annidamenti, per maggiore chiarezza é assegnata una colorazione diversa dello sfondo a righe alterne.

In alto a sinistra di ogni riga troviamo (partendo da sinistra) i controlli per l'eliminazione, lo spostamento e l'ordinamento sequenziale delle singole Activity. L'ordine delle righe determina la sequenza di esecuzione delle Activity, che inizia dall'Activity contenuta nella riga posizionata più in alto, fino ad arrivare all'ultima in fondo alla pagina.

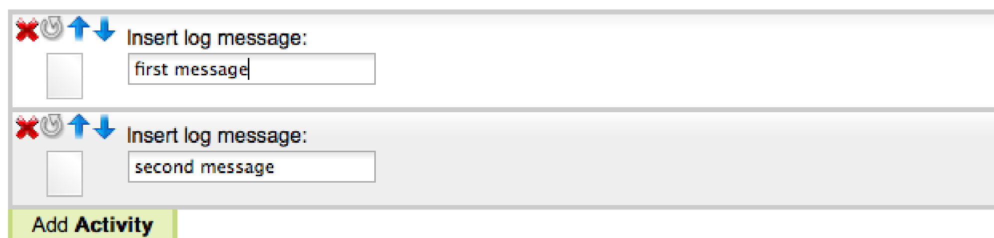

Figura 3.1: Sequence

Codice Jolie corrispondente alla Sequence:

```
main {
  println@Console("first␣message")();
  println@Console("second␣message")()
}
```

Tipo SequenceActivity:

```
type SequenceActivity: void {
  .name: string
  .activity*: Activity
}
```

3.4.2.3 Parallel

Parallel è l'elemento che rappresenta l'esecuzione in parallelo di processi.

Cliccando sul bottone *Add Sequence* è possibile aggiungere una nuova colonna contenente una Sequence.

In alto a sinistra di ogni colonna troviamo (partendo da sinistra) i controlli per l'eliminazione e per l'ordinamento orizzontale delle Sequence. L'ordinamento non è rilevante ai fini dell'esecuzione, in quanto avviene in parallelo per ogni Sequence, ma permette all'utente di rendere il workflow più chiaro e meglio organizzato.

L'esecuzione della Parallel termina quando tutte le Sequence terminano.

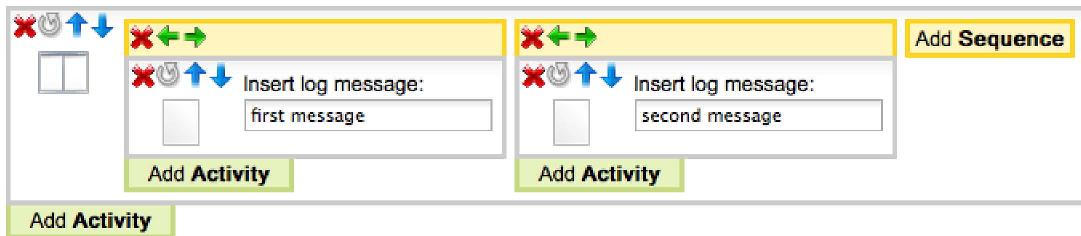

Figura 3.2: Parallel

Codice Jolie corrispondente alla Parallel:

```
main {
  { println@Console("first␣message")() }
  |
  { println@Console("second␣message")() }
  |
  { println@Console("third␣message")() }
}
```

Tipo ParallelActivity:

```
type ParallelActivity: void {
  .name: string
  .activity*: Activity
}
```

3.4.2.4 LogActivity

LogActivity svolge una funzione di logging.

Questa Activity permette all'utente di specificare un messaggio che verrà stampato sulla console durante l'esecuzione.

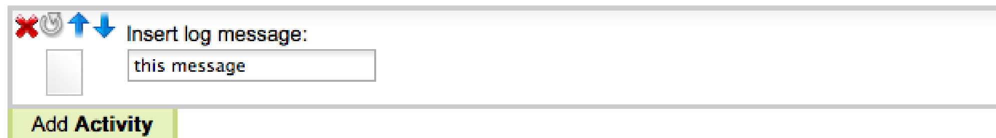

Figura 3.3: LogActivity

Codice Jolie corrispondente alla LogActivity:

```
1 main {
2   println@Console("this␣message")()
3 }
```

Tipo LogActivity:

```
1 type LogActivity: void {
2   .message: string
3 }
```

3.4.2.5 IfThenElse

IfThenElse rappresenta uno degli statement di controllo.

Questo elemento è formato da due Sequence e una Condition. La valutazione della condizione descritta dalla Condition determina se verrà eseguita la Sequence di sinistra, in caso di valutazione positiva, o di destra, in caso di valutazione negativa.

Codice Jolie corrispondente alla IfThenElse:

```
1 main {
2   if ( a == b) {
3     { println@Console("true")() }
```

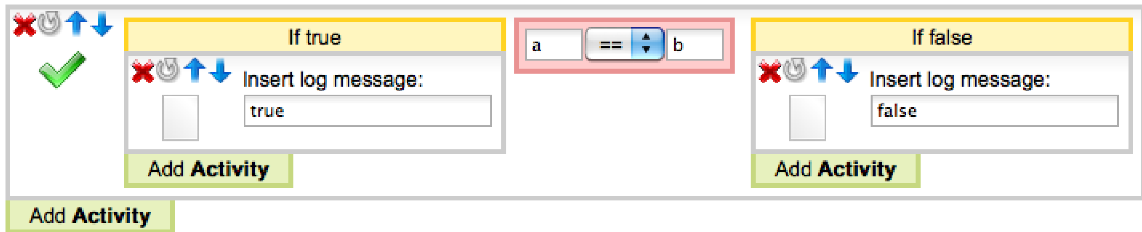

Figura 3.4: IfThenElse

```
4   } else {
5     { println@Console("false")() }
6   }
7 }
```

Tipo IfThenElseActivity:

```
1 type IfThenElseActivity: void {
2   .name: string
3   .activity_true: Activity
4   .activity_false: Activity
5   .condition: Condition
6 }
```

3.4.2.6 Loop

Loop rappresenta lo statement di controllo *while*.

Questo elemento ripete l'esecuzione della Sequence fintanto che la valutazione dell'espressione logica della Condition risulta positiva.

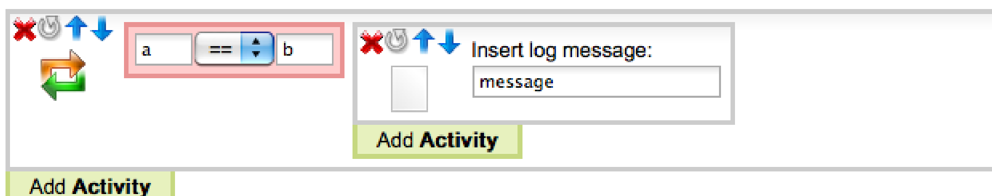

Figura 3.5: Loop

Codice Jolie corrispondente al Loop:

```
main {
  while( a==b) {
    { println@Console("message")() }
  }
}
```

Il tipo corrispondente al Loop è la WhileActivity:

```
type WhileActivity: void {
  .name: string
  .condition: Condition
  .activity: Activity
}
```

3.4.2.7 For

L'elemento For rappresenta l'ultimo statement di controllo.

Nel riquadro di sinistra è possibile specificare il numero di ripetizioni che si vogliono compiere della Sequence corrispondente. Il valore che rappresenta il numero di ripetizioni può essere incrementato o diminuito utilizzando, rispettivamente, i bottoni + e - o alternativamente cliccando sul bottone Edit che mostrerà un pannello per la digitazione da tastiera dei numeri.

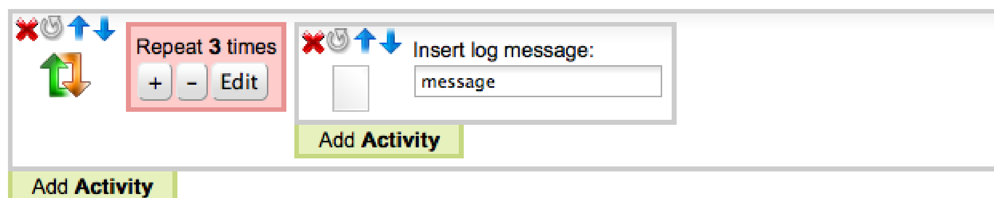

Figura 3.6: For

Codice Jolie corrispondente al For:

```
main {
  loopindex++;
```

```
3   for( loop[ loopindex ] = 0, loop[ loopindex ] < 3,
4         loop[ loopindex ]++) {
5     {
6       println@Console("message")()
7     }
8   }
9 }
```

Tipo ForActivity:

```
1 type ForActivity: void {
2   .name: string
3   .times: int
4   .activity: Activity
5 }
```

3.4.2.8 Notification

L'elemento Notification permette l'invocazione di un altro servizio senza attesa di risposta.

Come previsto per tutti gli elementi che rappresentano statement delle operation che richiedono dati di input, i ruoli (Role) e le operazioni (Operation) disponibili vengono forniti dal WorkflowManager attraverso i metodi **getRoles** e **getRole**.

Figura 3.7: Notification

Tipo NotificationActivity:

```
type NotificationActivity: void {
  .name: string
  .operation: string
  .role: string
  .input*: Datum
}
```

3.4.2.9 OneWay

L'elemento OneWay permette la ricezione di messaggi.

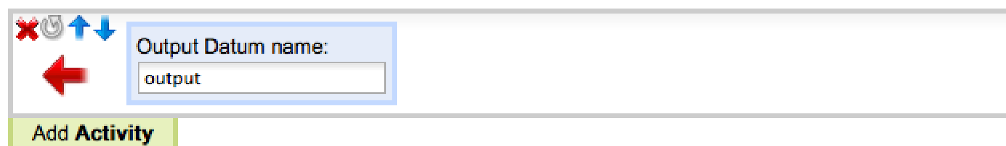

Figura 3.8: OneWay

Tipo OneWayActivity:

```
type OneWayActivity: void {
  .name: string
  .operation: string
  .role?:string
  .output*: Datum
}
```

3.4.2.10 RequestResponse

L'elemento RequestResponse permette l'invocazione di un altro servizio e la ricezione di un messaggio di risposta. Tra la richiesta e la risposta vengono eseguite le Activity specificate nella Sequence.

Tipo RequestResponseActivity:

```
type RequestResponseActivity: void {
```

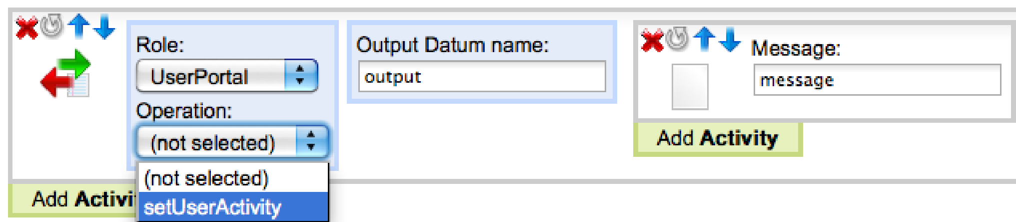

Figura 3.9: RequestResponse

```
2   .name: string
3   .operation: string
4   .activity: Activity
5   .role?: string
6   .output*: Datum
7   .input*: Datum
8 }
```

3.4.2.11 SolicitResponse

L'elemento SolicitResponse permette l'invocazione di un altro servizio e la ricezione di un messaggio di risposta.

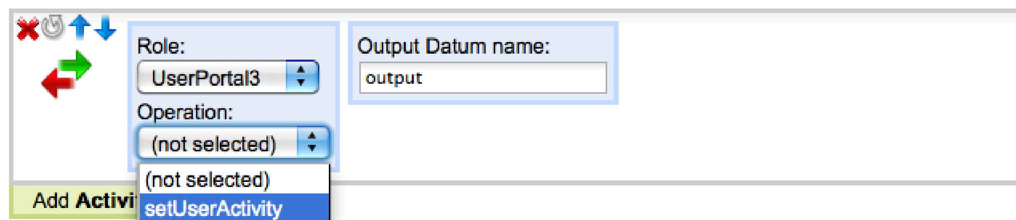

Figura 3.10: SolicitResponse

Tipo SolicitResponseActivity:

```
1 type SolicitResponseActivity: void {
2   .name: string
3   .operation: string
4   .role: string
```

```
5   .input*: Datum
6   .output*: Datum
7 }
```

3.4.2.12 Condition

L'elemento Condition permette la composizione di semplici espressioni logiche selezionando l'operatore logico dal menu a tendina e inserendo i dati nelle due caselle di testo.

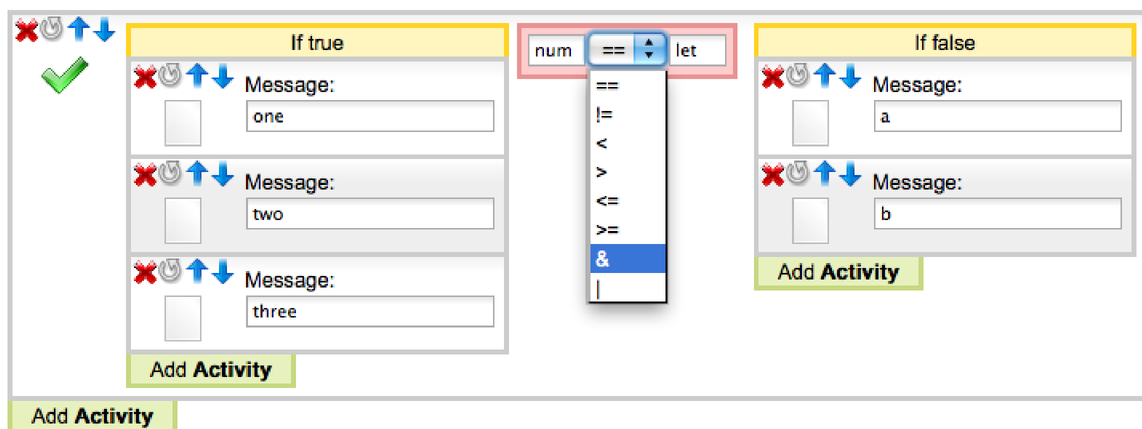

Figura 3.11: Condition

Tipi Condition e LogicalExpression:

```
1 type Condition: void {
2   .expression?: LogicalExpression
3   .leaf?: Datum
4 }
5
6 type LogicalExpression: void {
7   .left: Condition
8   .operator: string //LogicalOperator
9   .right: Condition
10 }
```

3.4.2.13 DataCreation

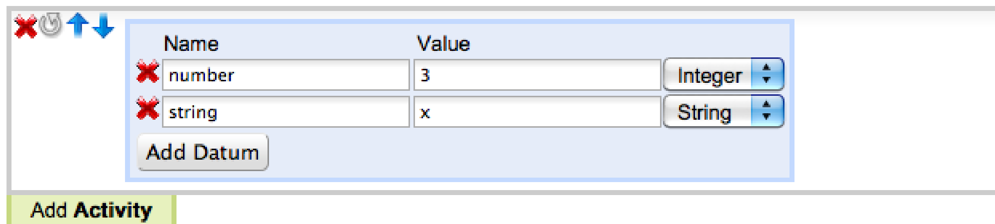

Figura 3.12: DataCreation

Tipo DataCreationActivity:

```
type DataCreationActivity: void {
  .name: string
  .tree: Datum
}
```

3.4.3 Visitor pattern

Le interfacce delle applicazioni GWT sono composte di *widget*. Alcuni widget, come ad esempio i pannelli che implementano l'interfaccia *Complex-Panel*, possono contenere a loro volta uno o più widget. Nell'implementazione di jEye questa proprietà è stata sfruttata per ottenere nel browser una struttura, chiamata **VisualWorkflowTree**, ad albero con corrispondenza quasi diretta con il WorkflowTree.

Ogni elemento del *Jolie Visual Model* visto finora è una estensione del widget *VerticalPanel*. Questo ha permesso l'aggiunta di decorazioni, controlli e l'eventuale annidamento di sotto-elementi.

L'implementazione dell'interfaccia **JolieElement** rende la classe un elemento visitabile da un **JolieVisitor**. **JolieActivity**, che è implementata da ogni Activity, invece è l'estensione dell'interfaccia JolieElement e permette la determinazione della classe delle Activity da visitare.

In jEye sono stati implementati tre visitor concreti: **JolieDebugVisitor**, **JolieMoveActivityVisitor** e **JolieWorkflowTreeComposerVisitor**.

3.4.3.1 JolieMoveActivityVisitor

Per implementare la funzione di spostamento di una Activity è stato necessario implementare il *JolieMoveActivityVisitor*.

Cliccando sul bottone di spostamento di una Activity, l'elemento selezionato viene rimosso dal VisualWorkflowTree e immagazzinato temporaneamente all'interno dell'elemento Workflow. La visita di un JolieMoveActivityVisitor (istanziato con parametro $show = true$) renderà visibili i bottoni di selezione per ogni Sequence, aggiungendo un messaggio di avvertimento nella posizione di origine. Una volta scelta la Sequence di destinazione, cliccando sul pulsante *Move Activity here*, l'elemento è inserito nella sua nuova posizione e rimosso dalla variabile temporanea del Workflow. Per nascondere i controlli resi precedentemente visibili viene compiuta una seconda visita di un JolieMoveActivityVisitor (instanziato con parametro $show = false$).

E' possibile spostare un solo elemento alla volta. La pressione consecutiva di due bottoni di spostamento, infatti, causerà come unico effetto la visualizzazione di un messaggio di istruzione sulla procedura corretta rivolta all'utente.

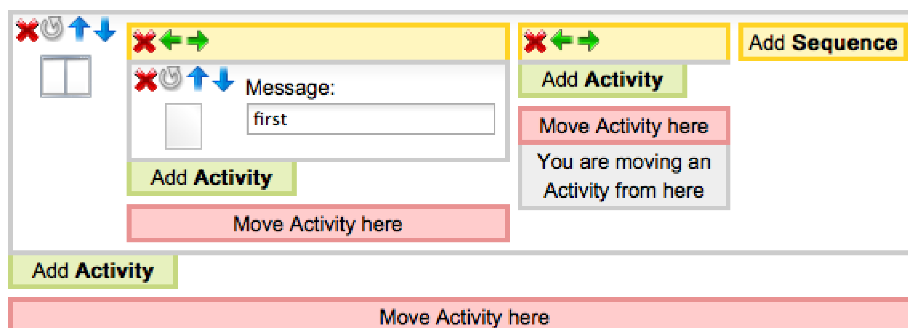

Figura 3.13: Spostamento di una Activity

3.4.3.2 JolieDebugVisitor

JolieDebugVisitor genera una stringa di debug che descrive la struttura del VisualWorkflowTree. Partendo dal nodo radice Workflow, il VisualWork-

flowTree è visitato in profondità. Durante la visita, ogni nodo aggiunge alla stringa risultato il proprio tipo seguito da un blocco di parentesi graffe che conterranno, se presenti, le informazioni estrapolate dal nodo corrente e la stringa risultante dalla visita dei sottonodi.

Esempio di una stringa restituita dal JolieDebugVisitor:

```
1  Debug:
2  Workflow {
3    Sequence {
4      Parallel {
5        Sequence {
6          LogActivity { first }
7        }
8        Sequence {
9          LogActivity { second }
10       }
11     }
12   }
13 }
```

3.4.3.3 JolieWorkflowTreeComposerVisitor

JolieWorkflowTreeComposerVisitor è il visitor che permette la composizione automatica del WorkflowTree.

La variabile che descrive la struttura del WorkflowTree è un *Value* chiamato *message*. Ogni nodo del VisualWorkflowTree aggiunge a message, nel momento della visita, i nodi necessari alla composizione di uno o più tipi JOLIE corrispondenti all'elemento visitato. Partendo dal nodo radice Workflow, il VisualWorkflowTree viene visitato in profondità e, al termine, message viene inviato come parametro al WorkflowManager attraverso l'utilizzo di una JolieCallback che richiama il metodo *parseWorkflow*. La risposta sarà una stringa contenente la traduzione in linguaggio JOLIE.

Per visualizzare la traduzione è sufficiente cliccare sul bottone *get JOLIE code* presente nell'intestazione della pagina principale di jEye. Un popup dinamico presenterà il codice all'utente, permettendone la copia o la semplice visualizzazione.

```
include "console.iol"
include "standard_types.iol"
execution{ concurrent }

init {
  loopindex = 0;
  println@Console("Service Sequence1298820604510 is running...")
}

main {
  println@Console("one")()

;
  println@Console("two")()

;
  println@Console("")()
```

Figura 3.14: Visualizzazione del codice JOLIE

Conclusioni

Nel corso di questa tesi è stato affrontato il tema della Service-Oriented Architecture, un importante paradigma per lo sviluppo di sistemi distribuiti. In questo ambito è stato analizzato il progetto JOLIE, un linguaggio che permette la manipolazione di servizi per costruire sistemi distribuiti service-oriented. Il mio contributo a questo progetto è stato quello di progettare e implementare l'applicazione jEye, un editor visuale web-based per la progettazione di workflow.

L'interfaccia dell'editor è accessibile tramite browser e non richiede altri strumenti di programmazione. La fase di progettazione è totalmente visuale richiede un numero di digitazioni da tastiera minime. E' stato elaborato un modello di rappresentazione visuale del linguaggio che permette all'utente solo un numero finito di operazioni. I workflow progettati possono essere convertiti in qualsiasi momento in linguaggio JOLIE.

L'espandibile struttura di jEye e le tecnologie utilizzate lasciano molto spazio a possibili sviluppi futuri. Grazie all'utilizzo di GWT, un potente framework Ajax, l'accesso all'interfaccia di progettazione è garantito anche su dispositivi Android e iOS . Approfondire lo studio dell'usabilità dell'interfaccia di jEye su tablet e smartphone lo renderebbe uno strumento di sviluppo multipiattaforma anche per dispositivi mobili.

I servizi relativi alla conversione sono forniti da una parte server realizzata completamente in linguaggio JOLIE. Grazie a questa architettura gli sviluppi futuri includono l'implementazione un servizio di memorizzazione lato server dei workflow progettati. Una volta memorizzati, i workflow po-

trebbero essere utilizzati per l'istanziazione in tempo reale di servizi JOLIE. L'introduzione in jEye di un pannello di controllo dei servizi JOLIE istanziati ne consentirebbe la gestione e la configurazione.

Si può quindi affermare che la realizzazione del progetto jEye soddisfa gli obiettivi prefissati.

Bibliografia

[GeoSch09] Dimitrios Georgakopoulos and Michael P. Papazoglou, *Service-Oriented Computing*, 2009, The MIT Press.

[Tho05] Thomas Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*, 2005, Prentice Hall.

[Jos07] Nicloai M. Josuttis. *SOA in Practice: The Art of Distributed System Design (Theory in Practice)*, 2007, O'Reilly Media.

[MonGui06] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, Gianluigi Zavattaro, *JOLIE: a Java Orchestration Language Interpreter Engine*, 2006, Department of Computer Science, University of Bologna.

[Gui07] Claudio Guidi, *Formalizing languages for Service Oriented Computing*, 2007, Department of Computer Science, University of Bologna.

[Gwt10] Google Web Toolkit, http://code.google.com/webtoolkit/