Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali
Corso di Laurea Triennale in Informatica

SUPPORTO A EDITOR WEB-BASED IN AMBIENTE SERVICE-ORIENTED: MODELLAZIONE DELL'AMBIENTE DATI

Tesi di Laurea in Linguaggi di Programmazione

Relatore:
Chiar.mo Prof.
Gabbrielli Maurizio

Presentata da:
Andrea Monzali

Sessione II
Anno Accademico 2010/2011

Introduzione

Il Service Oriented Computing (SOC) è un emergente paradigma di programmazione per sistemi distribuiti, in cui la componente software elementare è il servizio. Una SOA (Service Oriented Architecture) è composta da un certo numero di servizi, i quali possono essere tra loro composti al fine di ottenere applicazioni complesse.

Per quanto riguarda tale composizione, esistono due possibili approcci: *choreography* e *orchestration*. Il primo prevede la progettazione dell'architettura con una visione globale, mentre la seconda utilizza moduli tipici del workflow (come la composizione sequenziale, parallela e condizionale) per coordinare tra loro i vari servizi.

I Web Services sono un importante campo di applicazione del SOC, in cui i servizi sono distribuiti nella rete e accedibili tramite protocollo HTTP; è in questo contesto che troviamo i progetti di JOLIE e jEye.

JOLIE (Java Orchestration Language and Interpreter Engine) è un linguaggio di programmazione orientato ai servizi, in cui è possibile creare da zero nuovi servizi o comporne altri già esistenti sfrutttando l'approccio dell'*orchestration*.

jEye nasce come editor web-based per JOLIE; l'utente ha la possibilità di comporre graficamente un workflow di esecuzione e spedirne la descrizione al server, il quale restituirà il codice sorgente JOLIE corrispondente.

Scopo di questo lavoro di tesi è stato la realizzazione del supporto relativo all'*Ambiente Dati* per jEye, ovvero la gestione dei dati disponibili all'interno del programma ad un certo punto dell'esecuzione. È possibile suddividere il

lavoro in due fasi principali: la *modellazione* e l'*implementazione*. Benché esse vadano di pari passo, hanno dato vita a due tesi distinte.

In particolare, questa esposne tutto quello che riguarda la modellazione teorica: benché siano presenti alcuni riferimenti all'implementazione, qui verrà definito e descritto il modello matematico che sta alla base del *DE* (*Data Environment*) di jEye. È suddiviso in tre sezioni: una prima parte di contestualizzazione (verrà descritto il Service Oriented Computing nelle sue linee generali con una particolare attenzione ai Web Services), una seconda sezione in cui verranno presentati gli strumenti di JOLIE e di jEye (il nostro campo di studio); infine, l'ultima sezione descriverà il modello formale su cui si basa il Data Environment di jEye.

Indice

Capitolo 1

Il Service Oriented Computing

Il *Service Oriented Computing* (SOC) [1] è un nuovo paradigma di programmazione che utilizza i *servizi* come costrutti base per realizzare applicazioni. I servizi [2] sono intesi come entità software autonome ed indipendenti dalla piattaforma su cui vegono realizzate; ad essi è associata un'interfaccia che li descrive e li rende accessibili dall'esterno; in questo modo più servizi possono essere coniugati tra loro al fine di creare applicazioni distribuite.

Queste applicazini risultano essere di veloce sviluppo e costi contenuti grazie al principio di riuso del codice [2]. L'utilizzo di servizi, inoltre, vanta benefici anche in ambienti eterogenei; data infatti l'indipendenza dalla piattaforma, vi è una maggiore facilità di composizione in ambienti che si distinguono per architettura, sistema operativo e ambiente di sviluppo.

Come anticipato, l'approccio del Service Oriented Computing è duplice: se da un lato un servizio può essere invocato restituendo il risultato di una operazione, dall'altro è possibile coniugare tra loro diversi servizi ottenendo applicazioni distribuite che massimzzano il riuso del codice. Per quanto riguarda le modalità per coniugare tra loro servizi, esistono approcci differenti, tra cui l'*Orchestrazione* e la *Coreografia*, brevemente spiegate in seguito.

Il *SOC* è alla base della realizzazione delle *Service Oriented Architecture* (*SOA*) [3], architetture software basate su un insieme di principi, concetti e

metodologie utili al design e allo sviluppo di software nella forma di servizi interoperabili.

Di seguito viene data una breve panoramica del SOC: verrà presentata la tecnologia dei *Web Services*, ad oggi la più utilizzata tra quelle basate su SOC, definendone i linguaggi che la governano. Vi sarà poi un'ultima sezione sulle modalità per coniugare tra loro servizi differenti.

1.1 I Servizi

Un servizio può essere definito come un'entità software che fornisce funzionalità a chi lo utilizza, attraverso lo scambio di messaggi. Più precisamente l'OASIS (Organization for the Advancement of Structured Information Standards) [4] definisce il servizio come "*a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by service description*". Da questa definizione si deduce che ogni servizio prevede un'*interfaccia* ed una *descrizione* di se stesso: essendo infatti necessaria l'indipendenza tra servizi, essi devono fornire descrizione ed interfaccia secondo regole standard (generalmente espresse tramite XML), al fine di accedere ed essere accessibili da altri servizi.

I servizi rispondono ai seguenti principi guida:

- **standardizzazione**: i servizi condividono un "contratto" comune per quanto riguarda le comunicazioni.
- **loose-coupling**: ogni servizio deve minimizzare la dipendenza da altri servizi; essi infatti si limitano ad essere a conoscenza della reciproca esistenza.
- **astrazione**: la logica del servizio deve essere nascosta.
- **riusabilità**: il codice di un servizio deve essere riutilizzabile, per questo la logica dell'esecuzione deve essere suddivisa tra i vari servizi.

- **autonomia**: il servizio deve avere il controllo sulla logica che implemta.
- **granularità**: attraverso considerazioni progettuali, il servizio deve avere uno scope ottimale per le operazioni che esegue.
- **assenza di stato**: un servizio deve minimizzare l'utilizzo di risorse, rinviando la gestione delle informazioni di stato.
- **reperibilità**: i servizi devono essere dotati di informazioni attraverso cui si possa reperire ed utilizzare il servizio stesso.
- **componibilità**: i servizi devono essere progettati con la finalità di prendere parte ad una più ampia esecuzione.

1.2 Web Service

I Web Service sono ad oggi, la più diffusa tecnologia basata sul Service Oriented Computing. Il W3C (*World Wide Web Consortium*) definisce un Web Service come [5] "*a software system designed to support interoperable machine-to-machine interaction over a network [...] It has an interface described in a machine-processable format (specifically Web Service Description Language, known by the acronym WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, tipically conveyed using HTTP with an XML serialization conjunction with other Web-related standards.*"

In altre parole, i *Web Services* sono [6] componenti software indipendenti da piattaforma hardware, linguaggio di programmazione e sistema operativo che comunicano tramite tecnologie Web standard come HTTP e XML. Oltre a questo, vi sono due tecnologie che "collaborano" alla definizione dei Web Services: WSDL (*Web Services Definition Language*) e SOAP (*Simple Object Access Protocol*).

Il primo consente di realizzare documenti per la definizione del Web Services, in termini di locazione e funzionalità che il servizio mette a disposizione.

Chi utilizza questi servizi fa uso delle informazioni definite nel documento WSDL per costruire messaggi SOAP da inviare al servizio tramite HTTP.

WSDL e SOAP, quindi, sono rispettivamente un linguaggio ed un protocollo per definire servizi e scambiare dati. Entrambi sono basati su XML; rendendo così l'approccio ai Web Services indipendente da hardware, linguaggio di programmazione e sistema operativo.

1.2.1 Web Services Description Language (WSDL)

WSDL è un formato XML definito dal W3C, utilizzato descrivere servizi di rete che operano tramite messaggi. Questo standard definisce in modo astratto le operazioni e i messaggi relativi per poi essere associati ad un protocollo di rete ed a un formato di messaggi concreti. In questo modo alla definizione astratta di servizio e operazione, potranno essere associate più definizioni concrete.

Un documento WSDL realtivo a un Web Service conterrà quindi informazioni su:

- operazioni utilizzabili
- utilizzo: protocollo specifico e formato dei messaggi in input e output
- locazione: tipicamente l'indirizzo (anche detto *endpoint* del servizio, in formato URI) su cui risiede il servizio

Queste informazioni verranno poi associate con la definizione astratta del servizio.

1.2.2 Simple Object Access (SOAP)

SOAP è un protocollo basato su XML (anch'esso definito dal W3C) per scambiare dati tramite HTTP. Prevede lo scambio di messaggi scomposti in quattro parti.

Envelope specifica che il documento in questione è un messaggio SOAP

Header (opzionale) informazioni relative al messaggio (ora/data, informazioni di autenticazione, ...)

Body contenuto del messaggio

Fault (opzionale) informazioni circa gli eventuali errori avvenuti durante la trasmissione

1.3 Service Oriented Architecture

Una SOA è un'architettura software che utilizza un paradigma di tipo SOC per l'organizzazione e l'utilizzazione di risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti.

1.3.1 Le Operazioni

Ciò che risulta più interessante dal nostro punto di vista sono le operazioni (*operations*) utilizzabili, cioé quelle messe a disposizione dal servizio tramite il relativo documento WSDL.

Sone previste quattro primitive di scambio messaggi:

- **One Way**: l'endpoint rimane in attesa di ricevere un messaggio
- **Request Response**: l'endpoint aspetta un messaggio e risponde a sua volta
- **Solicit Response**: l'endpoint invia un messaggio e rimane in ascolto per una risposta
- **Notification**: l'endpoint invia un messaggio, procedendo poi nella sua esecuzione

Le operazioni *One Way* e *Request Response* sono dette "operazioni in entrata" (*input operations*), mentre la *Solicit Response* e la *Notification* rientrano nella categoria delle "operazioni in uscita" (*output operations*)

1.3.2 Coniugazione di servizi

Avnendo a che fare con diversi servizi su differenti piattaforme, una modalità per sfruttare al massimo le SOA consiste nel comporre i vari servizi tra loro. Questa data la natura distribuita del SOC, questa composizione deve essere il più possibile indipendente da un particolare servizio.

Esistono due possibili approcci [7] per affrontare il problema: l' *Orchestrazione* (*orchestration*) e la *Coreografia* (*choreography*).

Orchestration

Sfruttando questo approccio, viene definito un processo centrale di controllo (il quale potrebbe essere a sua volta un servizio), che coordina e mette in esecusione le varie SOA. È bene notare che i vari servizi invocati non hanno "coscienza" del loro singolare scopo, ma sono semplicemente invocati a richiesta.

Questa tecnica è quella che più si avvicina all'approccio della consueta programmazione, poiché utilizza costrutti comuni ai linguaggi tradizionali come la composizione sequenziale, parallela e condizionale; mantenendo sempre il controllo sull'esecuzione delle istruzioni.

L'approccio in questione è quello su cui si basa il linuguaggio JOLIE, uno degli argomenti principali della presente trattazione.

Choreography

Questa metodologia, contrariamente alla precedente, prevede che tutti i servizi coinvolti prendano parte attiva all'esecuzione, coordinandosi tra loro. È intuitivo concludere che utilizzando questo approccio, ogni servizio deve avere ben presente il suo scopo all'interno dell'esecuzione globale. Non vi è infatti un processo che coordina tutti gli altri, ma sono i servizi stessi che interagiscono tra loro scambiandosi messaggi e contribuendo all'esecuzione.

In questo caso quindi si scorge una natura più collaborativa in cui non c'è un'entità singola che controlla l'esecuzione.

Capitolo 2

JOLIE e jEye

L'idea alla base delle *Service-Oriented Architecture* è che entità software indipendenti (i servizi) collaborino assieme al fine di sviluppare applicazioni complesse.

Un importante campo di applicazione delle SOA sono i *Web Services*: si tratta di piattaforme in cui i servizi sono distribuiti nella rete ed accedibili tramite protocollo Internet. Tendenzialmente, per comunicare tra loro, i Web Services fanno un ampio uso di XML per descrivere dati e operation; JOLIE (*a Java Orchestration Language and Interpreter Engine*) [8] si pone in questo contesto proponendo un approccio differente: presenta la possibilità di creare nuovi servizi da zero o comporne alcuni già esistenti, utilizzando una sintassi simile al quella di C (più intuitiva dell'XML).

Quanto alla composizione di servizi, l'approccio adottato da JOLIE è quello dell'*orchestration*, una modalità che consiste nel delineare un workflow di esecusione, sfruttando primitive di programmazione come esecuzione condizionale, sequenziale e parallela. Tale modalità si contrappone a quella del *choreography*: per una più accurata descrizione di questi due concetti si rimanda al capitolo precedente, in cui viene presentato il *Service Oriented Computing* nelle sue linee generali.

Il progetto di jEye [9], invece, nasce come supporto a JOLIE: si tratta di un editor web-based accedibile tramite browser; qui l'utente (il program-

matore JOLIE) ha la possibilità di definire graficamente un workflow; il lato server dell'applicazione si occuperà poi di tradurre questo workflow in codice JOLIE.

È bene precisare che l'intento di jEye non è quello di creare un corrispondente grafico per ogni strumento disponibile in JOLIE: alcune operazioni esprimibili "a mano" sono volutamente rese non disponibili in jEye. Benché questo faccia calare il potere espressivo dell'editor, sottolinea il suo scopo di strumento ad alto livello, in cui è possibile strutturare un workflow nelle sue linee generali, andando poi ad aggiungere i dettagli a mano nel codice.

In questo capitolo verranno presentati gli strumenti di JOLIE e jEye: siccome il presente lavoro di tesi è incentrato su un modulo di jEye (quello relativo all'Ambiente Dati), la maggior attenzione del capitolo sarà incentrata su jEye; JOLIE verrà comunque descritto in quanto framework di lavoro, ma non ne verrà descritto il funzionamento nei dettagli. Il capitolo è quindi suddiviso in due sezioni che descriveranno questi due strumenti.

2.1 Il linguaggio JOLIE

JOLIE è un linguaggio per orchestratori [8]: è basato su un solido modello matematico e, a differenza di altri linguaggi nati con stesso scopo, presenta una sintassi simile a quella del C (invece dell'XML come proposto dagli altri linguaggi), rendendolo così più intuitivo per il programmatore alle prime armi.

Lo scopo di un orchestratore è definire alcune primitive di comunicazione che permettano la coordinazione dei servizi all'interno di una SOA; l'idea di fondo è che in ogni comunicazione vi siano due estremi, che chiamiamo *ruoli*; ognuno di questi ha la possibilità di predisporre punti di ingresso e punti di uscita; i primi sono intesi come i punti su cui un ruolo attende di essere invocato dall'esterno, mentre i secondi sono i punti da cui un ruolo invoca servizi esterni. Entrambi i punti di ingresso e di uscita, sono definiti da una

locazione (indirizzo ip e porta) e gestiti da un interfaccia (all'interno della quale sono poi definite operation e dati).

Presi ad esempio due ruoli, A e B, supponiamo che essi definiscano rispettivamente un punto di ingresso IN_A e di uscita OUT_B; se tali punti coincidono (per locazione ed interfaccia), potrà instaurarsi una comunicazione da B verso A. Tale comunicazione è unilaterale a causa della definizione dei punti di ingresso/uscita: l'invocazione delle operation è inteso da B per verso A.

È bene precisare che è l'invocazione di operation ad avere una direzione, non il flusso di dati. Nell'esempio precedente, esiste un flusso di comunicazione da B ad A, quindi B può richiedere i servizi di A ma non viceversa. Tuttavia, se B richiede un'operation che prevede in risposta un dato, questo verrà spedito da A verso B tramite lo stesso canale. Questo proprio perché l'unilateralità della comunicazione riguarda l'invocazione di operation, non il flusso di dati.

Volendo poi rendere la comunicazione bilaterale (in cui cioé A possa invocare le operation di B), sarebbe necessario definire un punto di ingresso in B ed un uguale punto di uscita in A.

2.1.1 La grammatica di JOLIE

Viene qui definita la grammatica di un sorgente JOLIE: per semplicità di scrittura verrà fornita una definizione sommaria, ampliando poi alcuni punti: le parti in corsivo saranno quelle poi successivamente riprese; il resto sono parole chiave del linguaggio. La star di Kleene posta dopo un nome, è da intendersi come la ripetizione di quel nome per un numero (anche nullo) di volte. Le varie ripetizioni saranno serate da virgole.

jolie-src :=
locations { *location_definition** }
operations { *operation_declaration** }
variables { *variables_declaration* }
links { *links_declaration* }

*definition**
main { *process* }
*definition**

2.1.2 Identifier

Prima della trattazione dei singoli simboli *non-terminali* del linguaggio, è bene definire il concetto di *Identifier* (o semplicemente *id*). Esso è un nome univoco che JOLIE mantiene in memoria per identificare differenti entità; ad esso possono essere associati dati, operation, locazioni, ...

Sono accettate come *id* espressioni alfanumeriche, in cui il primo carattere è richiesto essere una lettera.

2.1.3 Location

Come anticipato, i punti di ingresso/uscita sono definiti da una locazione (oltre che da un'interfacccia); una locazione è intesa come una coppia costituita da un indirizzo ip ed una porta. Tale coppia identifica all'interno della rete il punto su cui inviare dati o attendere richieste.

L'indirizzo ip può anche essere espresso tramite il nome della macchina all'interno di una rete locale.

location_definition := *id*="*hostname*:*port*"

2.1.4 Operation

Esistono quattro tipologie di operation: Notification, Solicit Response, One Way e Requst Response; le prime due sono da intendersi come operation di output mentre le restanti come operation in input.

Questa dichiarazione consisterà nell'elencare i nomi delle varie operation, distinguendone la tipologia. Una precisazione sulle operation di output: è

possibile rinominare queste operation associandovi un nome "interno" come alias per l'effettivo nome dell'operation, nella forma *id_interno = id_esterno*.

La grammatica delle operation è così definita:

$$\begin{aligned}\text{operation\_declaration} := \text{OneWay: } &(id)^* \\ \mid \text{RequestResponse: } &(id)^* \\ \mid \text{Notification: } &(id = id)^* \\ \mid \text{SolicitResponse: } &(id = id)^*\end{aligned}$$

La star di Kleene sta ad indicare che per ogni tipologia di operation, potebbe esistere più di una dichiarazione, ciascuna separata da virgole.

2.1.5 Variables

JOLIE utilizza variabili non tipate; sono implicitamente supportati i tipi *string*, *int* e *double*. Implicitamente significa che le stringhe sono distinguibili dai dati numerici perché racchiuse tra doppi apici.

La grammatica per definire le variabili consiste semplicemente nell'elencarne i nomi (gli *id*).

$$\text{variables\_declaration} := (id)^*$$

2.1.6 Links

I links rappresentano una metodologia di JOLIE per la sincronizzazione di processi paralleli: questo avviene inviando ed attendendo segnali su di uno stesso link. La dichiarazione dei link prevede l'elenco degli *id* che li rappresenteranno.

$$\text{links\_declaration} := (id)^*$$

2.1.7 Definitions

Come molti linguaggi di programmazione imperativa, esiste la possibilità di definire delle procedure, blocchi di codice invocabili da altri punti dell'esecuzione. Essi prevedono un nome che li identifichi ed un insieme di statement racchiusi tra parentesi graffe.

La grammatica sarà quindi nella forma:

$$\text{definition} := id \ \{ \ process \ \}$$

Il *main* è anch'essa una definition identificata da questo nome, e contenente il codice eseguito una volta che il servizio diventa attivo.

Per quanto riguarda il *process*, è utile distinguere tra statement e compositori: tra essi è poi possibile individuare alcune categorie, che qui di seguito verranno spiegate.

Statement per il controllo del flusso

Modificano il procedere lineare nell'esecuzione, come in ogni linguaggio di programmazione imperativa, è presente un costrutto di scelta (if then else), uno di iterazione determinata (for) ed uno di iterazione indeterminata (while). Inoltre esiste un costrutto specifico per invocare procedure, passando il controllo dell'esecuzione ad altre *definition*.

- esecuzione condizionale
  if (*condition*) {*process*} else {*process*}

- esecuzione ciclica
  while (*condition*) {*process*}
  for(n) {*process*}

- invocazione
  call (*id*)

Statement per le Operation

Si tratta delle primitive per invocare e predisporre rispettivamente le operation (rispettivamente di output e di input). Le prime saranno identificate dall'*id* dell'operation e da quello dell'orchestrator che contiene l'operation invocata. Le seconde, soltanto dall'*id* dell'operation messa a disposzione.

Entrambe presupporranno una serie di *id* ad indicare i dati inviati e/o i dati su cui verranno memorizzate le risposte.

- One Way
  id ⟨ *id** ⟩
- Request Response
  id ⟨ *id** ⟩ ⟨ *id** ⟩ (*process*)
- Notification
  id@*id* ⟨ *id** ⟩
- Request Response
  id@*id* ⟨ *id** ⟩ ⟨ *id** ⟩

Statement per la sincronizzazione

Si tratta di due costrutti, *linkIn* e *linkOut*, che consentono a processi paralleli di sincronizzarsi tramite i link.

- linkIn (*id*)
- linkOut (*id*)

L'idea consiste nel bloccare l'esecuzione a seguito di una chiamata a *linkIn*; per procedere è necessario che un altro processo richieda un corrispondente *linkOut* sullo stesso *id*.

Trattando diversi processi paralleli, è possibile che più di uno richieda la *linkIn* (e quindi si blocchi) sullo stesso *id*; in questo caso, nel momento in cui verrebbe evocata la *linkOut*, i processi bloccati verrebbero "svegliati"

secondo una politica non deterministica. Ossia ne verrebbe riattivato solo uno e gli altri disattivati (a questo proposito si veda il paragrafo relativo alla composizione non deterministica degli statement).

Altri statement

Troviamo infine alcuni statement che il linguaggio mette a disposizione, come le primitive per effettuare input/output su console, l'attesa ed il processo nullo.

- in (*id*)
  L'esecuzione si blocca in attesa di un dato da tastiera. Al termine, tale dato verrà memorizzato nella variabile identificata dall'*id*.

- out (*espressione*)
  Viene stampato a video il contenuto dell'espressione.

- sleep (*msec*)
  L'esecuzione si blocca per il numero di millisecondi indicati nell'argomento.

- nullProcess
  Processo nullo, corrispondento al *no-op* statement di molti linguaggu di programmazione.

Composizione Sequenziale

Consiste nel comporre tra loro i vari statement attraverso il ";", assumendo che ognuno di essi cominci la propria esecuzione non prima che il precedente non sia temrinato.

$$s_0; s_1; \ldots; s_N$$

Composizione Parallela

Qui, i vari statement vengono lanciati in esecuzione in concorrenza. L'intero costrutto termina quando l'ultimo statement è terminato. Sintatticamente viene usato il simbolo "||" per collegare tra loro gli statement.

$$s_0 \parallel s_1 \parallel \ldots \parallel s_N$$

Composizione Non Deterministica

Questo modello di composizione prevede più entità di esecusione; tali entità sono tutte bloccate, in attesa di un segnale che le attivi. Di queste, solo una verrà sbloccata e comincerà la sua esecuzione, mentre le restanti verranno disattivate.

Tali entità sono composte da una guardia g (uno statement bloccante) ed un processo p e sono correlate dal simbolo "++".

$$[g_0]p_0 + +[g_1]p_1 + + \ldots + +[g_N]p_N$$

La guardia, trattandosi di un'istruzione bloccante, potrà essere o un'operation di input (*One Way* o *Request Response*) oppure un'istruzione di *linkIn*; mentre il processo potrà essere un qualunque statement o composizione di essi.

2.1.8 I Dati

In ultima analisi, riportiamo la definizione dei dati: come già detto, in JOLIE le variabili sono implicitamente tipate, e non esiste una vera e propria dichiarazione.

Esiste inoltre la possibilità di strutturare i dati creando nuovi tipi a partire da quelli già esistenti. Vi sono quattro tipi nativi: *void*, *int*, *string* e *double*; strutturando un nuovo tipo, esso possiede un tipo nativo ed un numero di campi; ogni campo possiede un nome, un tipo ed un vettore di dati di quel tipo.

2.2 jEye: un editor web-based

jEye [9] nasce come supporto ai linguaggi service-oriented e si definisce come un "*editor visuale web-based ad alto livello per il linguaggio JOLIE*". Analizziamo meglio questa definizione:

- editor visuale
  jEye è un applicazione che presenta una GUI intuitiva che consente di strutturare un sorgente componendo graficamente alcuni blocchi.

- web-based
  Si tratta di un'applicazione web, a cui è possibile accedere tramite browser.

- ad alto livello
  È adatto alla strutturazione di ampi progetti, in cui è necessario mantenere una visione globale, disinteressandosi di alcuni dettagli.

- per il lingiaggio JOLIE
  Possiede una componente Server che su richiesta, può analizzare quanto definito graficamente dall'utente e tradurlo in codice JOLIE funzionante.

Come si nota, la definizione di jEye appena fornita, ne mette in luce i punti principali: si tratta di un'applicazione che non richiede grandi conoscenze di JOLIE, è intuitiva e non necessita di particolari strumenti software (come compilatori o librerie). Inoltre, essa produce codice JOLIE corretto, rendendo così jEye uno strumento rivolto sia ad utenti alle prime armi che non conoscono il linguaggio, sia ad utenti esperti che necessitao di strutturare ampi progetti mantenendone una visione globale.

Una precisazione: jEye è uno strumento corretto, ma (volutamente) non completo: non esiste infatti una corrispondenza esatta per ogni costrutto di JOLIE in un elemento grafico di jEye. Questa scelta ha una duplice motivazione: da un lato si è cercato uno strumento in grado di delineare un

workflow nelle sue linee generali, astraendo alcuni dettagli (che potrebbero poi essere aggiunti a mano sul codice). D'altra parte, limitare la libertà del programmatore consente all'editor di offrire qualche strumento in più; primo tra tutti, il supporto per l'Ambiente Dati (oggetto di questa tesi), che è stato realizzato a seguito di alcune supposizioni che, benché siano un limite alla libertà del programmatore, hanno creato lo scenario giusto per realizzarlo.

2.2.1 Le activity e la Strutturazione del Workflow

In jEye, il workflow è pensato come una composizione sequenziale, alla quale vengono via via aggiunti in coda i vari componenti. Definiamo l'*activity* come il componente grafico di jEye a cui corrisponde un costrutto JOLIE.

Possiamo definire un'activity α come:

$$\begin{aligned}\alpha := & \mathit{LogActivity} \mid \mathit{PrepareMail} \\ & \mid \mathit{DataCreation} \\ & \mid \mathit{For} \mid \mathit{Loop} \mid \mathit{IfThenElse} \\ & \mid \mathit{Parallel} \\ & \mid \mathit{OneWay} \mid \mathit{RequestResponse} \mid \mathit{Notification} \mid \mathit{SolicitResponse}\end{aligned}$$

La suddivisione in righe riprende una suddivisione per categorie che ora andiamo brevemente a descrivere:

- LogActivity e PrepareMail

  Si tratta di due attività di messe a disposizione dall'editor per semplificare il lavoro al programmatore, ma sono casi particolari di altre attività (rispettivamente SolicitResponse e DataCreation). Esse consentono di stampare a video una stringa (SolicitResponse dell'operation *println@Console*) e di strutturare un dato idoneo ad assere poi spedito come e-mail (DataCreation).

- DataCreation
  Consente di strutturare un dato definendone i vari campi e valori.

- For, Loop e IfThenElse
  Si tratta dei comuni costrutti di iterazione e scelta dei linguaggi di programmazione di tipo-imperativo.

- Parallel
  Consente all'utente di preparare diversi processi che verrano poi eseguiti in parallelo.

- OneWay, RequestResponse, Notification e SolicitResponse
  Le quattro primitive per l'invocazione/predisposizione di operation.

Di queste, alcune prevedono la possibilità di ridefinire al loro interno un numero arbitrario di ulteriori sequenze di activity; in particolare:

For, Loop Contengono una sequenza: quella che verrà iterata.

RequestResponse Contiene una sequenza, quella adibita ed elaborare la risposta.

IfThenElse Contiene due sequenze: da eseguire nel caso in cui la condizione sia vera o falsa.

Parallel Contiene un numero arbitrario di sequenze: quelle che verranno eseguite in parallelo.

2.2.2 Interazione dell'utente

L'interfaccia di jEye è molto minimale: oltre ai bottoni per otterenere il sorgente JOLIE, il resto è dedicato alla strutturazione del workflow. È presente una Sequence (inizialmente vuota) a cui l'utente agiunge via via le activity.

Questo lavoro di tesi ha creato il supporto per generare l'Ambiente Dati; a livello di interfaccia, è stato aggiunto ad ogni activity un bottono per ottenere l'Ambiente. Con la possibilità quindi di mostrare su richiesta l'elenco dei dati a disposizione di quell'activity.

Capitolo 3

L'Ambiente Dati di jEye

Nell'ambito dei linguaggi di programmazione, con *Ambiente* si intende l'insieme delle associaizoni tra un nome e l'oggetto denotato da quel nome. Tali associazioni possono riguardare oggetti di differente natura (variabili, procedure, servizi, ...); il lavoro di questa tesi riguarderà le variabili.

Definiamo quindi l'*Ambiente Dati* (o *DE*, *Data Environment*) il sottinsieme dell'Ambiente che considera esclusivamente le associazioni rigurdanti variabili.

Siccome JOLIE non richiede che tutti i dati vengano dichiarati prima dell'esecuzione (a differenza di altri linguaggi come Pascal o l'ANSI C), l'Ambiente Dati va espandendosi con il procedere dell'esecuzione.

È per questo motivo che si è soliti considerare l'Ambiente Dati a partire da un punto del programma; considerando perciò solo le associazioni (nome - dato denotato) ottenuto fino a quel punto.

Conoscere i nomi delle variabili a disposizione può rivelarsi molto utile sia all'utente umano (il programmatore), sia a tool automatici (ad esempio tool di typechecking). Se immaginiamo di avere un'operazione che necessita di un dato di un certo tipo, sapere se esiste e (in caso affermativo) conoscere il nome di una variabile di quel tipo è di aiuto al programmatore. D'altro canto, partendo dall'Ambiente Dati è possibile sviluppare tool automatici di correttezza sintattica che verifichino la coerenza tra tipi (ad esempio durante

un assegnamento).

Volendo estrarre l'Ambiente Dati da un programma con un paradigma di tipo imperativo, in linea teorica sarebbe sufficiente attraversare il listato controllando tutte le istruzioni e, nel caso in cui queste generino uno o più dati, aggiungere via via i dati così generati all'Ambiente.

Analizzando meglio il problema tuttavia, ci si accorge che la situazione è più complessa: in primo luogo, a causa del procedere non lineare dell'esecuzione. Sebbene si tratti di programmazione imperativa, in cui vi è una serie di istruzione eseguite in sequenza, JOLIE (come tanti altri linuguaggi) dà la possibilità di operare scelte (tramite costrutti di tipo IfThenElse), ripetere istruzioni per un numero imprecisato di volte (cicli) ed eseguire più istruzioni contemporaneamente (in concorrenza). In generale, dato il listato del codice, non è possibile prevedere la sequenza delle istruzioni eseguite a run-time; e di conseguenza, di quali dati si disporrà in un preciso punto.

Un'altra questione (più legata allo specifico linguaggio JOLIE) riguarda la struttura dei dati: si tratta infatti di dati semi-strutturati, in cui la struttura stessa è parte dell'informazione trasportata. In un contesto del genere diventa necessario considerare una variabile sempre associata al suo tipo (o struttura). Durante la trattazione, i termini *tipo* e *struttura* verranno utilizzati con riferimento allo stesso concetto.

Il presente capitolo sarà così organizzato: in un prima sezione verrà fornita la descrizione formale della struttura di jEye sulla quale è stato elaborato un modello per i dati (descritto nella seconda sezione). In ultimo vengono affrontate alcune questioni di carattere pratico (vedi quelle relative alle operazioni insiemistiche appena anticipate) con qualche riferimento all'implementazione ed alle semplificazioni in essa adottate.

3.1 Il Modello di jEye

In questa sezione viene affrontata la descrizione formale di jEye, per una descrizione più approfondita si rimanda al capitolo precedente in cui

questo strumento viene analizzato più nel dettaglio. Qui vengono ripresi (e formalizzati) i soli concetti inerenti all'Ambiente Dati.

3.1.1 Il Dato

Come già detto, l'*Ambiente* di un programma è l'insieme delle associazioni tra un nome e l'oggetto denotato da quel nome. Trattando l'Ambiente Dati, verranno prese in considerazione solo quelle associazioni riguardanti le variabili; in cui viene associato un nome ad un dato.

Il linguaggio JOLIE possiede alcuni tipi nativi (int, double, string, ...) ma soprattutto la possibilità di strutturare i dati come alberi, rendendo così la struttura del dato parte integrante dell'informazione. Per intenderci: presa una variabile JOLIE con un certo numero di campi, sarebbe inutile considerarne solo i valori tralasciando i nomi dei campi associati ad essi; questo proprio perché in un dato JOLIE la struttura è essenziale quanto i valori in essa contenuti. Vi sono perfino situazioni in cui il valore di un campo non viene neanche preso in considerazione, ma è l'esistenza stessa del campo a definire il dato.

Definiamo quindi l'*Ambiente Dati* come le associazioni tra un nome e il dato da esso denotato; considerando poi l'ampia rilevanza del tipo nel trattamento dei dati, ai fini di questa trattazione è possibile considerare il *DE* di un programma come l'insieme delle coppie (nome - tipo).

$$DE := \{\delta \mid \delta := (nome, \tau)\}$$

Occorre ora dare una definizione formale di un tipo JOLIE:

$$\tau := void \mid int \mid double \mid string$$
$$\tau := (\tau, \Phi)$$

con:

$$\Phi = \{\phi \mid \phi := (nome, \tau)\}$$

In JOLIE, un tipo τ è un tipo nativo (void, int, double o string) con un numero (anche nullo) di campi ϕ; ciascun campo possiede a sua volta un tipo τ ed un nome che lo identifica.

Facciamo qualche esempio:

```
type t1: int
```

```
type t2: int {
        .field1: int
        .field2: string
}
```

```
type t3: t2 {
        .field3: double
        .field4: void {
                .field5: int
                .field6: int
        }
}
```

3.1.2 Le Activity

Presentiamo ora la componente elementare dell'elaborazione: l'attività (*activity*). Con activity si intende ogni costrutto che l'utente può istanziare in jEye e che prenderà poi parte al Workflow.

Sia α un activity:

$$\alpha := np \mid dc \mid n \mid sr \mid ow$$
$$\alpha := \gamma ? \alpha_T : \alpha_F$$
$$\alpha := for\,(\eta)\,\alpha$$
$$\alpha := while\,(\gamma)\,\alpha$$
$$\alpha := rr\,(\delta_{IN}, \delta_{OUT})\,\alpha$$
$$\alpha := \alpha; \alpha$$
$$\alpha := \alpha \parallel \alpha$$

Seguono ora le definizioni delle singole activity; alcune di esse sono espresse in funzione di δ, che ai fini di questa tesi è possibile trattare come una coppia (nome - tipo). Si veda a questo poposito la precedente sezione relativa al Dato.

Si ricorda inoltre che trattando le primitive di invocazione di servizi, il punto di vista è quello dell'operation invocata: quindi i δ_{IN} sono i dati "richiesta", mentre i δ_{OUT} i dati "risposta". Considerando le primitive di output (Notificatione e Solicit Response) il punto di vista sarà invertito: δ_{IN} è il dato uscente ("richiesta" per l'operation invocata) mentre δ_{OUT} il dato entrante (in "risposta" dall'operation).

- *np*
  NullProcess: activity nulla, equivalente al *nop* di altri linguaggi di programmazione.

- $\gamma ? \alpha_T : \alpha_F$
  IfThenElse: esecuzione condizionale: data un'espressione booleana γ e due activity α_T e α_F, solo una di queste verrà eseguita a seguito della valutazione di γ

- $for\,(\eta)\,\alpha$
  For: ciclo con iterazione determinata. È definito da un numero intero $\eta \in \mathbb{N}$ (il numero di iterazioni) e dall'activity α che verrà iterata.

- $while\,(\gamma)\,\alpha$
  Loop: ciclo con iterazione indeterminata. È definito da un'espressione booleana γ (la condizione di entrata nel ciclo) e dall'activity α che verrà iterata.

- *dc*
  DataCreation: strutturazione di una variabile, qui l'utente ha la possibilità di specificarne i vari campi ed i relativi valori. Per una definizione più accurata di dati e tipi, si rimanda alla sezione relativa al Dato. L'activity *DataCreation* è definita dall'insieme Φ dei campi definiti dall'utente.

$$\mathrm{dc} := \Phi$$

- n

  Notification: primitiva di output a senso unico: viene spedito un dato δ senza però attendere nessuna risposta. È semplicemente identificata dal nome della variabile inviata.

  $$\text{n} := (nome_{IN})$$

- sr

  SolicitResponse: primitiva di output in cui l'elaborazione rimane bloccata in attesa di una risposta. Tale activity è identificata dal nome della variabile spedita e dal dato δ_{OUT} che viene ricevuto.

  $$\text{sr} := (nome_{IN}, \delta_{OUT})$$

- ow

  OneWay: primitiva di input a senso unico: viene resa disponibile un'operation e si rimane in attesa che un servizio esterno richieda tale operation inviando un dato δ_{IN}. A quel punto l'esecuzione riprenderà.

  $$\text{ow} := (\delta_{IN})$$

- $rr\,(\delta_{IN}, nome_{OUT})\,\alpha$

  RequestResponse: primitiva di input che, una volta invocata con un dato δ_{IN}, esegue un'activity α elaborando così una risposta che verrà spedita al mittente.

- $\alpha_1; \alpha_2$

  Sequence: esecuzione sequenziale di due activity. α_2 verrà eseguita solo quando α_1 sarà terminata.

- $\alpha \parallel \alpha$

  Parallel: esecuzione concorrente di due activity. L'activity *Parallel* termina nel momento in cui sono terminate entrambe le activity in esso contenute.

3.2 Creazione del Data Environment

Data la definizione formale degli aspetti di JOLIE e jEye inerenti al nostro scopo, verranno ora indagate le problematiche relative a generazione e gestione dell'Ambiente Dati.

L'Ambiente Dati è la risposta alla domanda "Quali saranno i dati a disposizione in quello specifico momento dell'esecuzione?", che potrebbe anche essere riscritta come "Cosa sarà successo fino a quel punto?".

In altre parole, conoscere l'Ambiente Dati in un punto del programma, equivale a sapere quali istruzioni saranno state eseguite fino a quel punto.

Ci si accorge però che le istruzioni *eseguite* fino ad un punto non sempre corrispondono alle istruzioni *scritte nel codice* prima di quel punto; se ad esempio consideriamo un costrutto di scelta, è possibile essere a conoscenza delle istruzioni *scritte* in entrambi i rami; tuttavia verranno *eseguite* quelle presenti in uno solo di essi. Quindi, se per generare l'Ambiente Dati verrebbero considerate tutte le istruzioni *scritte*, questo risulterebbe composto di associazioni ($nome \rightarrow oggetto$) effettivamente non esistenti (perché derivanti da operazioni non eseguite).

È stato fatto l'esempio del costrutto di scelta, ma lo stesso problema si pone anche con costrutti di iterazione ed esecuzione in parallelo.

La previsione a tempo di compilazione del comportamento di un programma rientra nell'ampio campo dell'analisi statica dei programmi (Static Code Analysis, SCA); e ha tra gli obiettivi quello di fornire una previsione sull'esecuzione del codice di un programma senza farlo effettivamente eseguire.

Il presente lavoro si occupa di fornire un modello semplificato che in parte limita la libertà dell'utente nel comporre il workflow. Alcune operazioni consentite scrivendo il codice a mano, verranno impedite durante la strutturazione del workflow tramite jEye. Le assunzioni e le semplificazioni adottate verranno via via discusse nel momento in cui sarà richiesto.

L'approccio presentato prevede un meccanismo che, data un'activity α, identifichi nel workflow tutte le activity che possono dare un contributo

d'Ambiente ad α. Verrà poi discusso l'apporto di dati che ogni activity a disposizione in jEye può dare, definiendo alcune regole e vincoli.

3.2.1 Attraversamento del Workflow

Il problema consiste nell'attraversare il sorgente di un programma dall'inizio fino ad una specifica posizione.

Come già detto, stiamo trattando di progrannazione di tipo imperativo; quindi un primo approccio potrebbe essere quello di valutare le istruzioni del sorgente dalla prima fino a quella di interesse ed aggiungere all'Ambiente i dati eventualmente generati da queste istruzioni.

Tale approccio non risulta tuttavia funzionale in quanto il modello imperativo (a cui si rifà JOLIE) presenta costrutti di scelta, iterazione, ricorsione e concorrenza. Tutto ciò impedisce di conoscere a tempo di compilazione l'esatta sequenza di istruzioni che verranno effettivamente eseguite.

Il Modello

Il modello qui presentato si basa sulla definizione stessa di activity:

$$\alpha := np \mid dc \mid n \mid sr \mid ow$$
$$\alpha := \gamma ? \alpha_T : \alpha_F$$
$$\alpha := for\,(\eta)\,\alpha$$
$$\alpha := while\,(\gamma)\,\alpha$$
$$\alpha := rr\,(\delta_{IN}, nome_{OUT})\,\alpha$$
$$\alpha := \alpha ; \alpha$$
$$\alpha := \alpha \parallel \alpha$$

Si nota che è possibile definire le activity con una struttura nidificata come fossero alberi; in particolare: For, While e RequestResponse possiedono una sola activity figlia; mentre IfThenElse, Sequence e Parallel si ramificano in due activity.

Questo modello annidato permette, data un activity, di risalire ad un'altra che la contiene, e così fino all'inizio del codice; chiameremo ROOT_ACTIVITY

il nodo radice di questo albero, l'activity che non è contenuta in nessun'altra activity.

L'idea di fondo consiste nel risalire al contenitore di un'activity α e valutare se esso può contribuire, in qualche sua parte, all'Ambiente Dati di α; risalendo poi di contenitore in contenitore verso il nodo radice, verranno via via accumulate tutte le associazioni ($nome \rightarrow oggetto$) che andranno a copmporre l'Ambiente Dati.

Definiamo l'insieme E_α come l'Ambiente Dati disponibile all'activity α, ovvero tutte le associazioni prodotte dall'inizio del programma fino ad α esculsa.

Mentre indicheremo semplicemente con E l'intero Ambiente Dati del programma (calcolato cioé al termine del sorgente).

La funzione π

Supponiamo che esista una funzione

$$\pi : A \rightarrow A_\pi$$

tale che $\pi(\alpha)$ rappresenta l'activity che contentiene α (o il *parent* di α).

Benché sia α che $\pi(\alpha)$ siano entrambe activity, si può notare che il codominio della funzione π non è esattamente A, ma A_π. Tale insieme è costituito dagli elementi di A, privati delle activity semplici e con l'aggiunta di un valore costante $NO\_CONTAINER$.

$$A_\pi := A \setminus \{np, dc, n, sr, ow\} \cup \{NO\_CONTAINER\}$$

Intuitivamente, $\pi(\alpha)$ rappresenta l'activity contenitore (o genitore): per questo motivo sono escluse tutte le activity semplici (ovvero non definite in funzione di altre activity); inoltre, esiterà un'activity che non è contenuta in nessun'altra (il nodo radice dell'albero): come anticipato, si tratta della $ROOT\_ACTIVITY$, e si definisce:

$$\pi(ROOT\_ACTIVITY) = NO\_CONTAINER$$

La funzione ϵ

Definiamo ora la funzione ϵ, la quale si occuperà di generare l'Ambiente di un'activity risalendo l'albero attraverso oppoortune chiamate alla funzione π.

Per maggior semplicità di scrittura, indicheremo:
$ite := \gamma ? \alpha_T : \alpha_F$
$for := for\,(\eta)\,\alpha$
$loop := while\,(\gamma)\,\alpha$
$rr := rr\,(\delta_{IN}, nome_{OUT})\,\alpha$
$p := \alpha \parallel \alpha$
$s := \alpha_1; \alpha_2$ con $s_1 := \alpha_1$ e $s_2 := \alpha_2$

Si assume inoltre che esista una funzione

$$G : A \times E \to E$$

Che presa un'activity ed un Ambiente, restituisce lo stesso Ambiente, eventualmente arricchito dai dati generati all'interno dell'activity.

Tale funzione delinea le regole secondo cui un'activity genera dati e, per questo motivo, merita una più approfondita analisi che verrà afforontata nella sezione successiva. Per ora si assuma semplicemente che questa funzione esiste.

Discutiamo ora la funzione ϵ: l'idea di fondo è che l'Ambiente Dati disponibile ad un'activity α sia quello disponibile al contenitore di α con eventualmente qualche aggiunta (proprie, a seconda del contenitore).

Per questo motivo, la funzione avrà come argomenti un'activity α ed il suo contenitore α_π; e restituirà l'Ambiente disponibile ad α sfruttando (tramite chiamate ricorsive) l'Ambiente di α_π.

$$\epsilon : A \times A_\pi \to E$$

Si noti che il secondo argomento della funzione dovrà appartenere all'insieme A_π, ovvero l'insieme A privato delle activity semplici con l'aggiunta

della costante $NO\_CONTAINER$ (si veda il precedente paragrafo relativo alla funzione π).

La funzione ϵ si definisce così:

(1) $\epsilon(\alpha, NO\_CONTAINER) = \varnothing$

(2) $\epsilon(\alpha, \alpha_\pi) = \epsilon(\alpha_\pi, \pi(\alpha_\pi))$ con $\alpha_\pi := ite \mid for \mid loop \mid p$

(3) $\epsilon(\alpha, rr) = \epsilon(rr, \pi(rr)) \cup \delta_{IN}$

(4) $\epsilon(\alpha, s) = \begin{cases} \epsilon(s, \pi(s)) & \alpha = s_1 \\ G(s_1, \epsilon(s, \pi(s))) & \alpha = s_2 \end{cases}$

Analizziamo un caso per volta: (1) corrisponde al caso base, in cui si prende in considerazione il nodo radice dell'albero: non esistendo nessun contenitore di tale activity (e di conseguenza nessuna precedente activity), l'Ambiente restituito sarà vuoto.

Il caso (2) comprende i casi in cui α è parte di un costrutto di scelta, di iterazione o di esecuzione parallela. Qui ci si limita ad ottenere l'Ambiente del contenitore tramite una chiamata ricorsiva su α_π senza aggiungere niente. Questo perché tali costrutti, per loro natura, non "consegnano" nessun dato proprio alle activity che contengono.

Il caso (3) riguarda la *RequestResponse* che, a differenza delle activity appena discusse, consegna all'activity figlia il proprio Ambiente con l'aggiunta del suo δ_{IN}, che consiste in una nuova associazione ($nome \rightarrow dato$), considerata come una coppia ($nome - tipo$).

Infine, il caso (4) riguarda l'esecuzione sequenziale: qui è necessario distinguere a seconda che α venga eseguito per primo o per secondo. Nel primo caso, α eredita semplicemente l'Ambiente della sequenza (come nei precedenti casi, sarà sufficientre risalire di un livello ed ottenere l'Ambiente del contenitore). Invece, con $\alpha = s_2$ bisognerà tener conto di tutti dati generati in s_1 (in quanto eseguito prima di α); ottenendo prima di tutto l'Ambiente della Sequence tramite $\epsilon(s, \pi(s))$, al quale vengono aggiunte tutte le associazioni generate in s_1 (ottenute tramite la funzione G).

In conclusione

È stato stabilito che ottenere l'Ambiente Dati di un'activity equivale a chiedere al suo contenitore di fornigli il proprio Ambiente con l'eventuale aggiunta di qualche associazione "interna" al contenitore stesso. Questo concetto è stato realizzato dalla funzione ϵ, che partendo da un'activity α, valuta la natura dei suo contenitore e fornisce le eventuali coppie (nome - tipo) "interne", aggiungendole all'Ambiente che a sua volta eredita dal proprio contenitore.

È quindi possibile definire una funzione ξ

$$\xi : A \to E$$

Definita come

$$\xi(\alpha) = \epsilon(\alpha, \pi(\alpha))$$

Che restituisce l'Ambiente Dati di un'activity (E_α).

Questa sezione è quella che più di tutte, è stata semplificata in fase di implementazione, a seguito di numerose assunzioni legate all'implementazione delle strutture dati di jEye. Per una più approfondita discussione di tali assunzioni e semplificazioni si rimanda alla parte finale di questo capitolo o direttamente alla tesi riguardante gli aspetti implementativi dell'Ambiente Dati di jEye.

3.2.2 Generazione dei dati: la funzione *G*

Individuate le parti del workflow che andranno a contribuire all'Ambiente Dati di un activity, è ora necessario stabilire le modalità secondo cui queste possano generare coppie (nome-tipo). Come già anticipato, esiste una funzione generatrice (la funzione G) che, presi un'activity ed un Ambiente, restituisce lo stesso Ambiente, eventualmente arricchito dei dati generati dall'activity.

Avremo quindi:

$$G : A \times E \to E$$

Ovviamente, esisterà una funzione G propria per ogni tipologia di activity, le quali verrano qui di seguito discusse una per volta; inoltre, verranno parallelamente discussi i vincoli a cui alcune activity saranno soggette.

Come più volte detto, ottenere l'Ambiente Dati comporta una previsione (a tempo di compilazione) del comportamento di un programma, una volta che questo entra in esecuzione. Tale requisito non è stato affrontato in questo lavoro, il quale modella l'Ambiente Dati a seguito di alcune assunzioni che (limitando in parte la libertà dell'utente) semplificano lo scenario studiato.

Per questo motivo, l'Ambiente generato non sarà sempre corretto, ma comunque coerente alle assunzioni fatte.

In primo luogo verrano quindi discusse queste assunzioni, dopodiché verrà presa in esame la funzione generatrice G declinata nelle varie tipologie di activity.

Assunzioni e Vincoli sullo Scenario

Abbiamo stabilito che in questo lavoro verrà presentato un modello per l'estrazione dell'Ambiente Dati da un programma, tale Ambiente non sarà sempre corretto, questo perché per poter definire l'Ambiente Dati sarebbe necessario conoscere a tempo di compilazione l'effettiva esecuzione del programma, cosa che non sempre è possibile stabilire (si pensi solo ai costrutti di scelta o ai cicli).

Per questo motivo nel modello qui presentato sono state fatte alcune assunzioni per rendere l'Ambiente se non totalmente corretto, almeno coerente con alcune ipotesi. Tali ipotesi verranno qui brevemente spiegate e riprese poi durante la trattazione della funzione generatrice.

Si tratta di regole che vengono imposte all'utente, di qui in avanti il modello assumerà di lavorare con tali regole rispettate; sul lato implementativo, questo si tradurrà in alcune casi con delle notifiche o dei divieti.

- Esecuzione Condizionale

  Si assume che all'interno del costrutto di scelta, i due rami generino gli stessi dati. Come si vedrà in seguito, al termine di tale costrutto, l'Ambiente risulterà essere l'intersezione dei dati prodotti nei due rami.

  Nel caso in cui in un ramo venisse generato un dato che non trova il suo corrispondente nell'altro ramo, questo sarebbe notificato all'utente: al termine del costrutto, l'Ambiente non conterrà quel dato.

- Esecuzione Parallela

  Si assume che nelle varie activity che vengono eseguite in parallel non vengano generati dati che hanno lo stesso nome; questo per evitare ambiguità nel momento in cu si va a generare l'Ambiente.

  A differenza della regola precedente, questa consiste in un divieto: non verrà concesso all'utente di prosegurie se la regola non venisse rispettata.

- Esecuzione Ciclica

  Lo scope di visibilità di JOLIE prevede che una volta dichiarata, una variabile prosegue la sua esistenza anche fuori dal blocco in cui è stata definita. Se all'interno di un ciclo viene generato un dato, questo continua ad esistere anche dopo la fine del ciclo. Per motivi relativi all'impossibiltà di conoscere il numero di iterazioni di un ciclo, si è deciso di limitare la visibilità di un dato creato dall'activity, allo scope della stessa activity, ma non anche fuori.

  Come nel caso del costruto di scelta, anche qui sarà prevista una notifica che informerà l'utente che quel dato generato non sarà presente nell'Ambiente al termine del ciclo.

Null Process: *np*

Trattandosi dell'activity nulla, il suo apporto all'Ambiente sarà nullo.

$$G\left(np, E\right) = E$$

Esecuzione Sequenziale: $\alpha_1; \alpha_2$

Trattandosi di esecuzione sequenziale, α_2 non comincerà la propria esecuzione prima che α_1 non sia terminata; intuitivamente, la seconda activity avrà a disposizone l'Ambiente a disposizione della prima, con in più tutti le coppie in essa generate.

$$G\left(\alpha_1; \alpha_2, E\right) = G\left(\alpha_2, G\left(\alpha_1, E\right)\right)$$

Esecuzione Parallela: $\alpha_1 \parallel \alpha_2$

Le due activity α_1 e α_2 vengono eseguite in concorrenza: dal punto di vista dei dati, esse sono indipendenti, nel senso che i dati generati nell'una non sono visibili dall'altra. Al termine dell'esecuzione dell'intero costrutto (che terminerà quando l'ultima activity al suo interno sarà terminata) saranno disponibili i dati generati in entrambe le activity.

Essendo l'Ambiente definito come un insieme, non ammette duplicati: nel caso in cui un'activity generasse dati già creati dall'altra activity (cioé con lo stesso nome), verrebbe meno l'univocità dei nomi, che avrebbe come conseguenza il collasso dei due dati in uno unico. Per questo motivo, si assume che le due activity α_1 e α_2 non generino dati con lo stesso nome. A livello di implementazione, questo consisterà in una notifica all'utente nel momento in cui all'interno delle due activity venissero creati due dati con lo stesso nome.

A livello di modello, si assume semplicemente che ogni activity generi i propri dati senza ambiguità di nomi.

$$G\left(\alpha_1 \parallel \alpha_2, E\right) = G\left(\alpha_1, E\right) \cup G\left(\alpha_2, E\right)$$

Con $G\left(\alpha_1, \varnothing\right) \cap G\left(\alpha_2, \varnothing\right) = \varnothing$ (nessun dato comune tra le due activity)

Esecuzione condizionale: $\gamma ? \alpha_T : \alpha_F$

Questo costrutto permette di eseguire una sola delle activity specificate, a seguito della valutazione dell'espressione booleana γ. Siccome in linea di principio γ non è nota a priori (ad esempio perché mette a confronto due valori che diventeranno noti solo a tempo di esecusione), non è possibile prevedere staticamente quale delle due activity venga eseguita.

È possibile tuttavia fare alcune ipotesi: se una variabile viene allocata all'interno di un ramo, sarà del tutto superflua all'esecuzione dell'altro ramo. Il codice che seguirà, quindi, non potrà fare assunzioni sull'esistenza di tale variabile. Potremmo quindi trattare l'Ambiente Dati in maniera simile allo scope di visibilità, in cui un dato definito all'interno di un blocco, rimane visibile (o contentuto nell'Ambiente) solo all'interno di quel blocco.

Per quanto funzionale, questo modello non rispecchia lo scope di visibilità di JOLIE, che non distrugge una variabile una volta terminato il blocco nel quale era stata dichiarata.

Siano E_T e E_F gli insiemi di dati generati nei due rami:
$E_T = G(\alpha_T, \varnothing)$
$E_F = G(\alpha_F, \varnothing)$

Un'altra proposta è di generare un Ambiente che sia l'intersezione tra E_T e E_F. Per quanto questa ipotesi non generi comunque un ambiente corretto (mancheranno sempre quei dati presenti in un solo ramo), è giusto pensare che se un dato compare solo in un ramo, tutta l'esecuzione successiva non ne farà uso.

Possiamo quindi concludere con:

$$G(\gamma ? \alpha_T : \alpha_F, E) = E \cup (E_T \cap E_F)$$

Che potrebbe essere riscritta con:

$$G(\gamma ? \alpha_T : \alpha_F, E) = G(\alpha_T, E) \cap G(\alpha_F, E)$$

Cicli: $for\,(\eta)\,\alpha$ **e** $while\,(\gamma)\,\alpha$

I cicli consentono di eseguire un'activity per un numero variabile di volte, la differenza tra i due sta nel fatto che nel *For* viene esplicitato il numero di iterazioni, mentre nel *While* (o *Loop* secondo la denominazione di jEye) il programmatore indica la condizione per continuare ad iterare.

Se nel costrutto di scelta la questione era più di carattere qualitativa (non si sapeva *quale* activity tra due sarebbe stata eseguita), qui vi è un problema quantitativo (non è noto per *quante* volte verrà eseguita un'activity).

Il numero di cicli η e la condizione γ possono, in linea di principio, dipedere da altri dati; e quindi in generale, non è possibile stabilire a tempo di compilazione il numero di iterazioni che verranno eseguite e, come ripetuto più volte, il presente lavoro si limiterà ad ottenere un Ambiente Dati non sempre sarà corretto, in quanto non verrà affrontata l'Analisi Statica del codice.

Una prima ipotesi consiste nel considerare le associazioni ($nome \rightarrow dato$) generate dall'iterazione di un'activity, come fossero generate eseguendo l'activity una volta sola. Ci si limiterebbe così a calcolare $G\,(\alpha, E)$ con α l'activity contenuta nel ciclo.

Immaginamo di avere un'activity che genera un'unica coppia (nome-tipo); iterandola più volte ogni iterazione crea una coppia. È possibile distinguere due casi: (1) queste coppie hanno tutte lo stesso nome, ma tipi differenti, oppure (2) anche il nome della variabile cambia ad ogni ciclo.

Nel primo caso, l'approccio appena descritto risulterebbe accettabile, in quanto l'Ambiente verrebbe comunque arricchito di un solo dato (il nome della variabile è sempre quello), nel secondo, invece, l'ipotesi presentata non consente di fare previsioni sulla quantità di istanze generate.

A questo si aggiunga che è possibile che l'activity espliciatata nel ciclo non venga eseguita neanche una volta; in quel caso, la funzione generatrice produrrebbe dei dati che, in esecuzione, non verrebbero mai generati.

L'approccio qui adottato, invece, si basa sulla seguente idea: nel caso in cui l'esecuzione di un'activity generi nuovi dati, se c'è la necessità che questi

vengano utilizzati anche al termine del ciclo, dovevano già essere inizializzati prima ancora di cominciarlo; in altre parole, dovevano già essere presenti nell'Ambiente prima dell'iterazione dell'activity. Con questa modalità, l'Ambiente rimarrebbe invariato.

Inoltre, all'interno dell'activity, tutte le associazioni create sarebbero comunque localmente visibili, e verrebbero poi tolte dall'Ambiente una volta terminato il ciclo.

Questo approccio produce un Ambiente che non è corretto, in quanto dei dati generati nel ciclo, non ne viene considerato nessuno; a questo proposito si ricordi che le regole di scope di JOLIE non prevedono la distruzione di una variabile al termine del blocco in cui è stata dichiarata; quindi, volendo rispettare le regole di JOLIE, i dati creati all'interno di un ciclo, dovrebbero essere utilizzabili anche fuori da esso.

Nonostante questo, si è deciso di utilizzare l'approccio descritto, anche con l'accorgimento che jEye nasce come uno strumento ad alto livello, con cui è possibile strutturare a grandi linee il workflow di esecuzione; per poi andare ad agiungere a mano i dettagli.

$$G\left(for\left(\eta\right)\alpha, E\right) = E$$
$$G\left(while\left(\gamma\right)\alpha, E\right) = E$$

Data Creation: *dc*

Si tratta dell'activity che consente all'utente di strutturare un dato nei suoi vari campi. Intuitivamente al termine di questa activity, l'Ambiente dati sarà arricchito da una nuova associazione tra il nome dela variabile appena strutturata ed il dato stesso.

Siccome la *DataCreation*, per sua definizione, stabilisce i vari campi ϕ, il contributo all'Ambiente Dati di questa activity è la coppia $(nome, \{\phi\})$.

$$G\left(dc, E\right) = E \cup \left(nome, \{\phi\}\right)$$

Notification: *n*

Si tratta dell'invocazione di un'operation con relativo invio di un dato. Dopo l'invio l'esecuzione procede e non è prevista nessuna rispota; per questo motivo a seguito di una *Notification* l'Ambiente rimane invariato. Si assume infatti che il dato inviato faccia già parte dell'Ambiente Dati.

$$G(n, E) = E$$

Solicit response: *sr*

Invocazione di un'operation con relativo invio di un dato ed attesa della risposta. Una volta che l'operation ivnocata risponde, l'Ambiente viene arricchito dal dato ricevuto come risposta

La funzione generatrice per la *Solicit Response* sarà:

$$G(sr, E) = E \cup \delta_{OUT}$$

Si ricorda che trattando servizi, il punto di vista è quello dell'operation invocata: per questo δ_{OUT} è il dato di output. Siccome la *Solicit Response* invoca un'operation esterna, δ_{OUT} è il dato che ritorna.

One Way: *ow*

Primitiva di input a senso unico: L'esecuzione rimane bloccato aspettando di essere invocati da un servizio esterno. Tale servizio, invierà anche un dato, il δ_{IN} dell'operation. È tale dato che verrà aggiunto all'Ambiente.

$$G(ow, E) = E \cup \delta_{IN}$$

Request response: *rr*

Questa operation è simile alla *OneWay*, con la differenza che è prevista l'elaborazione e l'invio di una risposta. Oltre al δ_{IN} ricevuto dal servzio invocante, verrà aggiunto all'Ambiente dutti i dati eventualmente generati dall'activity che elabora la risposta.

$$G(rr, E) = G(\alpha, E \cup \delta_{IN})$$

3.3 Dal modello all'implementazione

In questa sezione vengono descrite tutte quelle questioni di carattere pratico che partendo dal modello formale, mettono le basi per l'implementazione. È bene premettere che se la modellazione può considerarsi un lavoro a sé, il lavoro di implementazione è da considerarsi come continuazione all'implementazione di jEye; quindi molte delle scelte adottate per la gestione del Data Environment sono scaturite dalla natura dell'architettura di jEye.

Per una descrizione approfondita degli aspetti implementativi, si rimanda alla tesi parallela a questa, il cui scopo è proprio descrivere l'implementazione del modulo di jEye relativo al *DE*. Qui verranno solo trattai gli aspetti che riguardano strettamente il modello.

3.3.1 Architettura di jEye e Semplificazione al Modello

L'architettura di jEye prevede una definizione di activity leggermente diversa rispetto a quella fornita qui, considerando questa nuova definizione, l'algoritmo per ottenere l'Ambiente Dati rimane comunque corretto, anzi, è possibile semplificarlo

Riprendiamo la definizione di activity:

$$\alpha := np \mid dc \mid n \mid sr \mid ow$$
$$\alpha := \gamma ? \alpha_T : \alpha_F$$
$$\alpha := for\,(\eta)\,\alpha$$
$$\alpha := while\,(\gamma)\,\alpha$$
$$\alpha := rr\,(\delta_{IN}, \delta_{OUT})\,\alpha$$
$$\alpha := \alpha; \alpha$$
$$\alpha := \alpha \parallel \alpha$$

In fase di implementazione, l'entità elementare non è più l'activity, quanto la Sequence; definiamo una Sequence σ come:

$$\sigma := \{\alpha_i\}$$
$$\alpha := np \mid dc \mid n \mid sr \mid ow$$

$$\alpha := \gamma ? \sigma_T : \sigma_F$$
$$\alpha := for\,(\eta)\,\sigma$$
$$\alpha := while\,(\gamma)\,\sigma$$
$$\alpha := rr\,(\delta_{IN}, \delta_{OUT})\,\sigma$$
$$\alpha := \sigma_1 \parallel \sigma_2$$

In pratica: le definizioni di activity rimangono pressocché invariate, con l'unica differenza che le activity composte (IfThenElse, For, Loop, RequestResponse e Parallel) vengono definite in base ad una o più Sequence (σ, σ_T, σ_F, σ_1, σ_2); la Sequence, dal canto suo, è definita come un insieme (ordinato) di activity con arietà variabile.

Mantenere l'ordinamento nella Sequence è essenziale: si tratta dell'ordine con cui le activity vengono definite ed aggiunte in coda alla Sequence; trattando di generazione di dati è importante sapere "quali istruzioni vengono generate prima (quali dati vengono generati prima)".

In questo conteso, anche la funzione ξ per generare l'Ambiente Dati sarà diversa: consideriamo infatti la Sequence come un grande contenitore di activity, alcune delle quali, a loro volta, possono contenenre altre Sequence. È possibile ottenere, data un'activity α, tutte le activity contenute nella Sequence che contiene α, che precedono α nell'ordinamento.

$$\Sigma\,(\alpha) = \{\beta \mid \beta \in \pi\,(\alpha) \wedge \beta < \alpha\}$$
con $\beta < \alpha$ inteso come "β precede α nell'ordinamento"

Data la funzione Σ, è possibilie ridefinire ξ come:

$$\xi\,(\alpha) = \begin{cases} \varnothing & \Pi\,(\alpha) = NO\_CONTAINER \\ G\,(\Sigma\,(\alpha)\,, \xi\,(\Pi\,(\alpha))) & altrimenti \end{cases}$$
$$\text{con } \Pi\,(\alpha) = \pi\,(\pi\,(\alpha))$$

In pratica: consideriamo $\sigma = \pi\,(\alpha)$, l'Ambiente di α dovrà tener conto delle activity di σ precedenti ad α, cioé $\Sigma\,(\alpha)$; ma non solo. La Sequence σ potrebbe essere contenuta a sua volta in un'activity α'.

$$\alpha' = \pi(\sigma) = \pi(\pi(\alpha)) = \Pi(\alpha)$$

Si ripete quindi la stessa prcedura su α' fino a risalire alla Sequence principale.

3.3.2 Operazioni Insiemistiche

L'Ambiente Dati è definito come un insieme e per definizione stessa di insieme, non sono ammessi duplicati all'intero di esso. Questo concetto è coerente con quello di Ambiente Dati, in cui non potranno esistere due dati uguali. Il nome di una variabile è un identificativo univoco della variabile stessa all'interno dell'Ambiente.

Tuttavia, le operazioni insiemistiche (unione, intersezione e differenza) si basano sulla relazione di ugaglianza; l'Ambiente Dati è un insieme di associazioni, quindi è necessario definire una relazione di uguaglianza per associazioni.

Queste associazioni riguardano variabili, quindi è sensato pensare che due associazioni ($nome \rightarrow oggetto$) si possano considerare uguali, se sono uguali i nomi. In questa tesi, l'associazione è stata trattata come una coppia (nome, tipo), quindi si può dire:

$$(nome, \tau)_1 = (nome, \tau)_2 \Leftrightarrow nome_1 = nome_2$$

Conclusioni

Questa tesi ha affrontato il tema dell'Ambiente Dati per editor di linguaggi servce-oriented (in particolare jEye e JOLIE). È stato realizzato (ed implementato) un modello per estrarre le associazioni (nome → dato) generate in un programma, dall'inizio fino ad un preciso punto.

Il supporto realizzato vincola l'utente ad alcune regole, limitando il potere espressivo del linguaggio. Tuttavia un requisito di progettazione dell'editor jEye è la realizzazione di un tool ad alto livello, con cui il programmatore può strutturare un workflow di esecuzione, inserendo poi alcuni dettagli a mano nel codice. Le limitazioni introdotte riguardano alcuni aspetti interni dell'esecuzione del servizio descritto, ma non l'interazione che esso dovrà avere con altri servizi esterni.

È quindi possibile concludere che le assunzioni imposte a jEye non ne intaccano l'idea di fondo.

È bene ripetere che l'elenco di nomi che espone l'Ambiente Dati, non corrisponde esattamente alle associazioni create fino a quel momento. Questo fatto non deriva da una mancanza in fase di modellazione; ma è anch'esso una conseguenza dei requisiti di jEye: tutte le scelte adottate vanno nella direzione di dare al programmatore la conoscenza sufficiente ad impostare l'interazione che il servzio avrà con l'esterno.

Parlando di sviluppi futuri, è possibile delineare due aspetti: un servizio (lato Client o lato Server) di *typechecking* e la realizzazione del supporto relativo all'estrazione di nomi dall'Ambiente. Se finora sono state definite le modalità per ottenere le associazioni (nome → oggetto) ed inserirle nell'Am-

biente, è intuitivo pensare che questi nomi possano essere utilizzati all'interno della strutturazione del servizio (feature finora non esistente). In tale contesto, sarebbe utile disporre di un tool di typechecking che preso un nome e la descrizione di un tipo, verifichi l'esistenza di un dato con quel nome e, in caso affermativo, l'appartenenza di tale dato al tipo specificato.

Allo stato attuale, le operazioni di jEye che prevedono l'utilizzo di nomi già appartenenti all'Ambiente, consistono semplicemente nello specificare un nome; non viene tuttavia effettuato nessun controllo su di esso.

Bibliografia

[1] Dimitrios Georgakopoulos, Michael P. Papazoglou: Service-Oriented Computing

[2] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann: Service-Oriented Computing: State of Art and Research Challenges

[3] SOA: http://en.wikipedia.org/wiki/Service-oriented_architecture

[4] OASIS: http://en.wikipedia.org/wiki/OASIS_organization

[5] Hugo Haas, Allen Brown: Web Services Glossary

[6] Erin Cavanaugh: Web Services: Benefits, challenges, and a unique, visual development solution, Altova

[7] Abdaladhem Albreshne, Patrick Fuhrer, Jacques Pasquier: Web Services Orchestration and Composition

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

[9] Federico Roffi: Progettazione e Realizzazione di un Editor web-based per il libìguaggio JOLIE, Alma Mater Studiorum - Università di Bologna