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: Implementazione dell'Ambiente Dati

Tesi di Laurea in Paradigmi di Programmazione

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Matteo Bonazzi

Sessione II
Anno Accademico 2010/11

Alla mia famigli, in particolare a mio nonno deceduto, all'altro mio nonno che non vedeva l'ora di avere un laureato in famiglia e ai miei genitori che hanno "sponsorizzato" la mia esperienza universitaria.

Introduzione

Il Service-Oriented Computing (SOC) [1] è un emergente paradigma informatico per sistemi distribuiti, in cui il servizio ha il ruolo di componente basilare. La SOA [1] (Service Oriented Architecture) rappresenta una metodologia di applicazione dei principi del SOC.

Questa architettura contempla due possibili approcci per comporre tra loro i servizi: *choreography* e *orchestration*. Il primo affronta la progettazione con una visione globale del sistema, mentre attraverso l'utilizzo dell'approccio orchestration, il sistema viene gestito utilizzando moduli tipici del workflow per coordinare tra loro un numero arbitrario di servizi.

È proprio questo il contesto in cui risiedono i progetti di JOLIE e jEye e principalmente quello dei web services, un diffuso campo di applicazione del SOC in cui i servizi sono distribuiti nella rete e accessibili attraverso il protocollo HTTP (principalmente).

JOLIE (Java Orchestration Language Interpreter Engine) è un linguaggio di programmazione che utilizza l'approccio dell'orchestration per creare nuovi servizi o anche comporne di già esistenti per ottenere nuovi servizi a loro volta.

jEye è un tool web-based per JOLIE che permette di creare graficamente un workflow e, una volta elaborato dal server, ottenere il sorgente corrispondente in linguaggio JOLIE.

Lo scopo del progetto trattato in questa tesi è stato la realizzazione di una nuova feature per jEye consistente nella generazione dell'*Ambiente Dati* ovvero l'insieme dei dati (intesi come variabili) disponibili in un determinato

punto del workflow. Questo lavoro si è sviluppato in due fasi: quella di *modellazione* e quella di *implementazione*.

Questa tratta ciò che riguarda l'*implementazione* con riferimenti al modello matematico che si è realizzato. La trattazione si svilupperà in una prima parte di contestualizzazione in cui verrà illustrato il concetto di Service Oriented Computing, una seconda sezione per introdurre JOLIE e jEye (la base del nostro lavoro) e un ultimo capitolo che tratterà, per l'appunto, la fase di ingegnerizzazione e implementazione concreta del Data Environment per jEye.

Indice

Elenco delle figure

Capitolo 1

Il Service-Oriented Computing

Il *Service-Oriented Computing* (da ora in poi SOC) è un nuovo paradigma informatico che utilizza i servizi come costrutti base per la realizzazione di software. Le applicazioni così realizzate risultano essere di veloce sviluppo e costi contenuti grazie al principio di riuso del codice. Vantano benefici anche nello sviluppo in ambienti eterogenei poichè la caratteristica di indipendenza dei servizi porta una maggiore facilità di composizione degli stessi anche in queste realtà.

Con il termine *Service-Oriented Computing* ci si riferisce ai principi, concetti e metodi che delineano un paradigma utile alla realizzazione di applicazioni basate su *Service-Oriented Architecture*. SOA è un paradigma di design e sviluppo nella forma di *servizi* interoperabili.

1.1 I Servizi

Un servizio può essere definito come un'entità che fornisce funzionalità agli utilizzatori attraverso uno scambio di messaggi. Più presicamente l'OASIS [2, 6] (Organization for the Advancement of Structured Information Standards) lo definisce 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 ser-*

vice description". Da questa definizione si deduce che ogni servizio prevede un'interfaccia e una descrizione di se stesso.

I servizi rispondono ai segueint principi guida:

- **loose-coupling**: ogni servizio deve minimizzare la dipendenza da altri servizi
- **astrazione**: la logica del servizio deve essere nascosta
- **riusabilità**: un servizio deve essere riutilizzabile
- **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 delegando, quando necessario, le informazioni di stato
- **reperibilità**: i servizi devono essere forniti con informazioni attraverso le quali si possano reperire ed utilizzare

1.2 Web Service

I Web Service [3] sono, al giorno d'oggi, la più diffusa tecnologia di ispirazione SOC. Il W3C [5] definisce un web service come "*a software system designed to support interoperable machine-to-machine interaction over a network*"; continuando si legge anche "*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 [8] messages, tipically conveyed using HTTP with an XML serialization conjunction with other Web-related standards.*"

In sintesi un web service è un servizio che espone una interfaccia in formato WSDL grazie alla quale gli utilizzatori possono interagire con "messaggi" SOAP in formato XML incapsulati e consegnati tramite protocollo HTTP.

1.2.1 Web Services Description Language (WSDL)

WSDL [7] è un formato XML per descrivere servizi di rete che operano tramite messaggi. Tale 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 un servizio od 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 [9]) su cui risiede il servizio

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

1.2.2 Le operazioni

Ciò che risulta più interessante dal nostro punto di vista sono le operazioni (in inglese *operations*) utilizzabili, cioè quelle messe a disposizione dal servizio.

WSDL prevede quattro primitive di invio 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**: in questo caso l'endpoint invia un messaggio e rimane in ascolto per una risposta
- **Notification**: l'endpoint attende un messaggio

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 Service-Oriented Architecture (SOA)

La SOA è *un paradigma per l'organizzazione e l'utilizzazione delle risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti. Fornisce un mezzo uniforme per offrire, scoprire, interagire ed usare le capacità di produrre gli effetti voluti consistentemente con presupposti e aspettative misurabili.* Questo paradigma contempla due possibili approcci per affrontare il problema: *orchestration* [10] e *choreography* [11, 12]

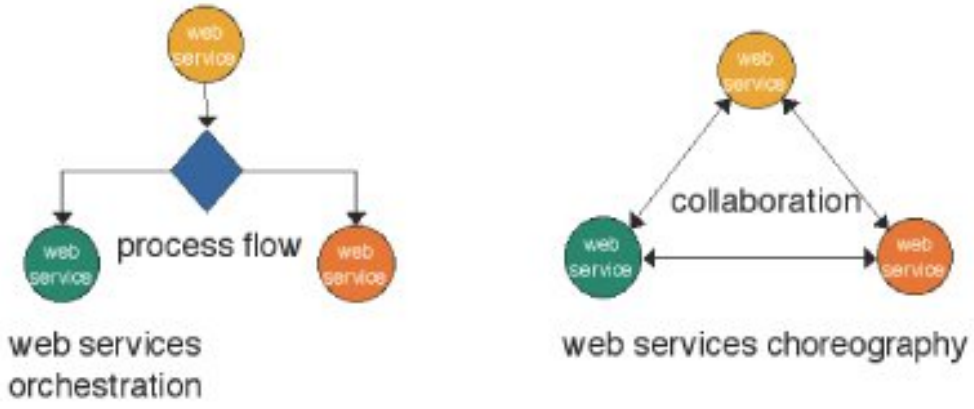

Figura 1.1: Approcci alla SOA

1.3.1 Orchestration

Questa tecnica è quella che si avvicina di più all'approccio della consueta programmazione poichè utilizza costrutti comuni ai linguaggi trdizionali come la composizione sequenziale, parallela e condizionale. Questo metodo approccia il problema attraverso un processo/servizio principale che coordina l'interazione di altri servizi 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 di questa trattazione

1.3.2 Choreography

Con questa metodologia, contrariamente a quella precedente, non vi è un processo che coordina tutti gli altri, ma sono i servizi stessi che interagiscono tra loro scambiandosi messaggi e contribuendo all'esecuzione delle istruzioni che compongono il programma o il nuovo servizio complesso. 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

Il fulcro dell'architettura Service-Oriented è la collaborazione tra entità *software indipendenti* (servizi) che vengono composti per dare vita a nuovi servizi od applicativi complessi.

Lo scenario più diffuso in cui si applica la SOA sono i Web-service. Questo particolare tipo di servizi offre la possibilità di essere utilizzato tramite protocollo HTTP e comunica, di norma, attraverso messaggi in fomao xml (secondo una convenzione SOAP).

JOLIE [13] si caratterizza come linguaggio per comporre o creare da zero nuovi servizi: ha una sintassi molto simile al linguaggio C ed utilizza un approccio di tipo *orchestration* per "governare" la complessità del software. Questo approccio, approfondito nel capitolo precedente, prevede che il controllo del flusso di esecuzione sia nelle mani di un processo principale che coordina tutti i servizi coinvolti.

jEye [14] è un tool grafico pensato per l'utilizzo in rete tramite browser che offre la possibilità di creare, visivamente, workflow. Il diagramma di flusso viene poi elaborato e inviato a un server che lo traduce in codice sorgente JOLIE. Ovviamente le potenzialità di jEye sono minori di quelle del linguaggio per cui è stoto progettato, ma offre comunque uno strumento per la strutturazione primaria del codice, lasciando al programmatore la possibilità di integrare il codice in modo tradizionale.

2.1 JOLIE

Jolie è un linguaggio basato sull'approccio dell'orchestrazione e la sua sintassi è molto simile a quella del linguaggio C che lo rende adatto anche ai programmatori alle prime armi.

Il compito di un orchestratore all'interno della SOA è quello di definire metodi di comunicazione e coordinazione tra servizi. In una comunicazione basilare vi sono sempre almeno due estremi coinvolti che vengono chiamati *ruoli* ognuno dei quali dovrà poter spedire e ricevere messaggi. Per fare ciò, ognuno di questi avrà la possibilità di predisporre *punti di ingresso* e *punti di uscita*. Questi punti saranno identificati tramite la loro *locazione* che consiste in una coppia (indirizzo ip, porta). Tornando alla suddivisione tra punti, quelli di ingresso sono quelli su cui il ruolo attende di essere invocato dall'esterno, mentre i punti di uscita sono quei punti attraverso i quali il ruolo può invocare servizi esterni.

Per fare un esempio: se prendiamo due ruoli A e B che definiscono rispettivamente due punti, uno di ingresso IN_A e uno di uscita OUT_B. Se i due punti coincidono in locazione (e *interfaccia*), la comunicazione potrà instaurarsi.

La comunicazione così creata sarà *unilaterale* da B verso A per definizione dei punti di ingresso/uscita, quindi si deve considerare il "verso" all'atto dell'invocazione dell'operation e non l'andamento del flusso dei dati/messaggi.

Questa considerazione merita una piccola precisazione: ricordando che le operations primitive utilizzabili sono quattro (one-way, request-response, solicit-response e notification), come visto nel capitolo precedente, si nota facilmente che vi sono due operazioni che prevedono risposta all'invocazione. Nella situazione dell'esempio, se necessario, A potrà rispondere a B senza problemi; la restrizione che impone la comunicazione unilaterale è solo a livello di invocazione delle operations. Quindi ciò che non potrà fare A sarà l'invocazione delle operations di B, a meno di non definire pinti di ingresso in B e instaurare una nuova "linea" di comunicazione da A verso B. In questo modo verrebbe a crearsi una comunicazione *bilaterale* composta concretamente da

due unilaterali a verso opposto.

2.1.1 Grammatica JOLIE

jolie-src :=

locations { *location_definition** }
operations { *operation_declaration** }
variables { *variables_declaration* }
links { *links_declaration* }
*definition**
main { *process* }
*definition**

L'asterisco posto dopo un nome significa la possibile ripetizione di questo per un numero qualsiasi (anche zero) di volte. Le parole in corsivo sono *simboli non terminali* che saranno oggetto della nostra analisi per sommi capi, mentre le altre sono *parole chiave* del linguaggio stesso.

Prima di continuare deve essere definito il concetto di *identifier* (*id*) come nome univoco (identificatore per l'appunto) che il linguaggio utilizza e mantiene per identificare entità quali dati, operation,...

2.1.2 Location

La location, come ormai sappiamo dai paragrafi precedenti, definisce i punti di input e output in coppia con la definizione de interfaccia.

Una location è costituita da una coppia (indirizzo ip, porta) in cui il primo parametro identifica l'endpoint su una rete e il secondo definisce il "canale" su cui deve "transitare" la richiesta.

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

2.1.3 Operations

Le principali operazioni di JOLIE si dividono in *input operations* che sono la one-way e la request-response, e *output operations* che consistono nella notification e nella solicit-response come già esaminato nel capitolo precedente.

La dichiarazione di un operation consiste nella definizione del nome dell'operazione specifica e della sua tipologia.

Una particolare caratteristica di JOLIE permette di poter assegnare un alias locale (ad uso prettamente interno al servizio) per le operazioni in uscita del tipo "id_interno"= " id_esterno".

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

L'asterisco indica che per ogni tipologia di operazione si potrebbe definire più di una dichiarazione e ciascuna andrebbe separata da una virgola.

2.1.4 Variables

JOLIE supporta implicitamente i tipi nativi *string*, *double* e *int* ma utilizza variabili non tipate. Più banalmente JOLIE può distinguere le stringhe dai dati numerici poichè racchiuse tra doppi apici.

La dichiarazione di variabili viene fatta semplicemente attraverso un elenco di nomi:

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

2.1.5 Links

I links sono canali di comunicazione dedicati per la sincronizzazione di processi paralleli. Questa sincronizzazione avviene attraverso l'invio e l'attesa di ricezione di determinati segnali sullo stesso link.

La dichiarazione avviene definendo l'elenco dei nomi che li rappresentano:

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

2.1.6 Definitions

Un'altro punto in comune di JOLIE con altri linguaggi di programmazione di tipo imperativo è la possibilità di creare sotto-procedure richiamabili più volte durante l'esecuzione della procedura principale.

Queste sotto-procedure sono blocchi di codice indentificate da un nome e sono molto utili specialmente quando una certa elaborazione deve essere ripetuta più volte durante l'esecuzione dell'applicazione.

La dichiarazione avviene in questo modo:

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

Il *process*è il corpo della sotto-procedura che viene racchiuso tra parentesi graffe e si compone di *statement* opportunamente composti che tratteremo tra poco.

Esiste inoltre una sotto-procedura particolare chiamata *main* che viene invocata una volta che il servizio viene attivato e rappresenta quindi il corpo vero e proprio del servizio.

2.1.7 Statement

Gli statement sono costrutti per la programmazione. Questi costrutti si possono suddividere sulla base del loro campo di applicazione:

- **statement per il controllo del flusso**

  Questi sono i costrutti condizionali e per i cicli (con iterazioni determinate o indeterminate) che, per la loro natura, modificano il lineare flusso di esecuzione. In questa categoria rientra anche il costrutto che permette l'invocazione di altre definition:

  - esecuzione condizionale
    if (*condition*) {*process*} else {*process*}
  - esecuzione ciclica
    while (*condition*) {*process*}
    for(n) {*process*}
  - invocazione
    call (*id*)

- **statement per le operations**

  Sono statement che consentono l'invocazione delle input eøoutput operations. Per le operazioni in uscita dovranno essere fornite informazioni sull'id dell'operation e sull'orchestratore a cui fare riferimento, mentre per le operazioni in ingresso basterà specificare soltanto l'id

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

- **statement per la sincronizzazione**

  Sono costrutti che consentono a processi paralleli di potersi sincronizzare:

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

Il primo, una volta invocato, sospende l'esecuzione finchè un altro processo invocherà un linkOut sullo stesso *id*.

Nel caso in cui più processi fossero in attesa (a seguito di una chiamata linkIn) sullo stesso id e un ulteriore processo invocasse un linkOut su quello stesso id, solo uno dei processi suddetti verrebbe "risvegliato"secondo una *politica non deterministica*.

- **altri statement** quali:
  - in (*id*)
    L'esecuzione viene sospesa fino all'arrivo di un dato da tastiera che verrà memorizzato nella variabile id.
  - out (*espressione*)
    Stampa a video.
  - sleep (*msec*)
    Interrompe l'esecuzione per il numero di millisecondi fornito come argomento.
  - nullProcess
    Il processo nullo, equivalente al *no-op* di altri linguaggi di programmazione.

Questi costrutti possono essere composti in vari modi:

- **composizione sequenziale**

  Attraverso il simbolo ";" ci si assicura che ogni statement cominci la propria esecuzione solo quando il precedente abbia terminato la sua.

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

- **composizione parallela**

  L'esecuzione degli statement avviene in concorrenza. Questo si ottiene attraverso il simbolo "||".

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

- **composizione non deterministica**

  Questo scenario si verifica quando più entità sono bloccate in attesa di un segnale per poter riprendere l'esecuzione. All'arrivo del segnale, solo una di queste verrà sbloccata e potrà continuare la propria esecuzione mentre le altre rimarranno ancora bloccate.

  Queste entità si compongono di una guardia g (uno statement bloccante) ed un processo p e sono correlate dal simbolo "++".

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

  Per quanto detto sopra g potrà essere una input operation (one-way o request-response) o un'istruzione di sincronizzazione (linkIn); quanto al processo, potrà essere una qualunque composizione di statement o anche un singolo statement.

2.1.8 I dati

In JOLIE le variabili vengono considerate implicitamente tipate e non esiste una dichiarazione esplicita. Tuttavia il lingiaggio in questione consente di strutturare i dati creando nuovi tipi partendo da quelli già esistenti.

I quattro tipi nativi disponibili in JOLIE sono int, string, double e void. La strutturazione di un nuovo tipo prevede la specifica di un tipo nativo e può contenere vari campi caratterizzati a loro volta da nome, tipo e un array di dati di quel tipo.

2.2 jEye

JEye si definisce come "*editor visuale web-based di alto livello per il linguaggio JOLIE*".

Più nello specifico jEye è un tool grafico dotato quindi di una GUI intuitiva che permette di strutturare software (anche di notevole complessità) attrvarso blocchi. Tutto ciò viene fatto tramite web browser ed è coadiuvato da una componente server che, su richiesta può processare l'elaborato grafico e produrne il codice JOLIE equivalente.

L'utilizzo di questo tool, viste le sue peculiarità, non richiede particolari conoscenze di JOLIE e non necessita nemmeno di strumenti software specifici come librerie. Tutto ciò rende jEye uno strumento ideale per un ampio spettro di utilizzatori: dai novizi alla programmazione con una conoscenza minimale del linguaggio, fino ai programmatori più avanzati che potranno beneficiare di una strutturazione grafica per progetti anche ragguardevoli.

Una doverosa precisazione è da fare: questo strumento è volutamente incompleto, cioè non vi è una corrispondenza di una caratteristica di jEye e per ogni costrutto presente in JOLIE. Questo impone ovviamente dei limiti ma da un altro punto di vista facilita l'esperienza d'utilizzo; inoltre questo strumento è stato pensato per strutturare workflow, ed astrarre su alcuni dettagli può consentire l'introduzione di altre features molto comode come, ad esempio l'Ambiente Dati (oggetto della trattazione) che si basa su considerazioni che potranno sembrare limitanti ma hanno comunque dato modo di sviluppare adeguatamente questa caratteristica.

2.2.1 Le activity e il workflow

Il workflow, in jEye, è strutturato come composizione sequenziale di componenti chiamate *activity* le quali vengono man mano accodate. L'activity la definiremo il componente grafico di jEye che rappresenta un costrutto JOLIE.

La definizione di un activity α sarà quindi:

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

$| \ OneWay \ | \ RequestResponse \ | \ Notification \ | \ SolicitResponse$

La suddivisione in righe propone una categorizzazione delle activity:

- LogActivity e PrepareMail
  Casi particolari di altre activity, consentono di stampare a video (solicit-response dell'operation println@console) e di strutturare un dato idoneo all'invio tramite mail. Sono state create per comodità d'uso del programmatore.

- DataCreation
  Consente la strutturazione di dati definendone campi e valori.

- For, Loop e IfThenElse
  Gli equivalenti degli statement per il controllo di flusso presenti, praticamente, in ogni linguaggio imperativo.

- Parallel
  Si riferisce alla composizione parallela e consente all'utente di preparare i vari processi da eseguire concorrenzialmente.

- OneWay, RequestResponse, Notification e SolicitResponse
  I corrispondenti "grafci" per le quattro operazioni primitive di input/output.

Alcune di queste attività, proprio per loro natura, consentono di definire internamente un numero arbitrario di sequenze di activity:

For, Loop Contengono una sequenza che contiene il codice da iterare.

RequestResponse Contiene una sequenza che rappresenta l'elaborazione della risposta da generare.

IfThenElse Contiene le due sequenze rispettivamente da eseguire o meno in funzione del risultato booleano della condizione.

Parallel Contiene un numero arbitrario di sequenze che verranno eseguite in concorrenza.

2.2.2 Interazione dell'utente

L'interfaccia minimale di jEye consiste quasi completamente in elementi atti al controllo della strutturazione del workflow.

Un progetto vuoto si presenta come una *sequence* vuota in cui è possibile inserire activity.

Il lavoro oggetto di questa trattazione ha riguardato lo sviluppo della caratteristica aggiuntiva per la generazione dell'Ambiente Dati che ha coinvolto l'interfaccia solo minimamente attraverso l'aggiunta di un pulsante in ogni activity.

Capitolo 3

Implementazione del Data Environment

Questo lavoro di tesi mira a creare un ulteriore aiuto alla programmazione che già offre jEye. Il progetto consiste nell'introduzione del supporto all'ambiente dati in un punto del workflow. Grazie a questa feature aggiuntiva sarà possibile all'utente la consultazione di tutte le variabili e relativo tipo nel punto desiderato del workflow. Questo è solo uno dei vantaggi e possibili applicazioni che questo lavoro comporta, una ulteriore realizzazione sarà la validazione del dato in ingresso o in uscita nelle invocazioni di servizi.

Il processo di implementazione è stato condotto su un applicativo già realizzato e funzionante basato su librerie Google Web Toolkit (GWT) e rivolto all'utilizzo via browser. Questo ha comportato scelte implementative non sempre ottimali per poter "convivere" con l'ingegnerizzazione già in essere.

Questo capitolo tratterà anche aspetti riguardanti l'ingegnerizzazione del software, cioè quel processo decisionale e implementativo che sta alla base della realizzazione di un prodotto che abbia buone caratteristiche per l'utilizzo intelligente delle risorse, buona usabilità nonchè una buona implementazione leggera e flessibile, cioè aperta a sviluppi ulteriori in futuro.

3.1 Design Pattern

Dato che questa trattazione esamina gli aspetti implementativi, vale la pena spendere qualche riga per chiarire il concetto di *design patter*.

I design pattern sono metodi di risoluzione generale per problemi ricorrenti, possono essere classificati in *creazionali*, *strutturali* e *comportamentali* a seconda del campo di applicazione.

I primi risolvono situazioni in cui il problema principale è la creazione o istanziazione di variabili anche complesse: tra questi è stato utilizzato il pattern *singleton*. Lo scopo del metodo è quello di garantire che di una determinata classe venga creata una ed una sola istanza, e di fornire un punto di accesso globale a tale istanza.

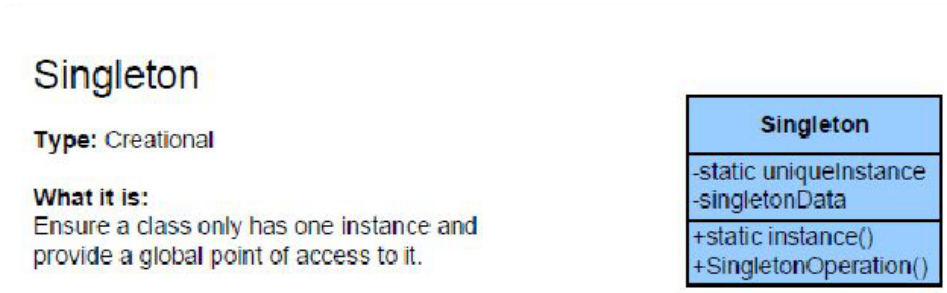

Figura 3.1: Schema del design pattern singleton.

La classe *strutturale* invece mira al riuso di oggetti esistenti, fornendone una interfaccia più adatta agli utilizzi del caso. Nel progetto in esame non si è ritenuto opportuno introdurre nessun pattern di questo tipo, poichè si sarebbe rischiato di sovra ingegnerizzare.

I pattern *comportamentali* infine sono quelli che forniscono soluzioni alle più comuni interazioni fra oggetti. Come vedremo in seguito, il pattern *visitor* già adottato nella creazione iniziale del tool grafico, ha dato modo di poter introdurre un nuovo comportamento alla struttura in modo molto agevole. Questo metodo di risoluzione permette di separare un algoritmo

dalla struttura dati a cui viene applicato, in modo da poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa.

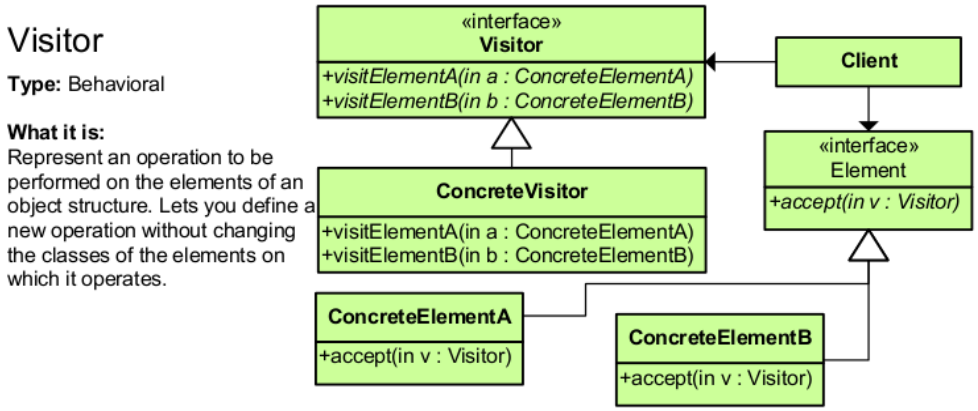

Figura 3.2: Schema del design pattern visitor

3.2 Ambiente Dati

Il concetto di *ambiente* [15] in un programma è definito come l'insieme delle associazioni tra un nome e l'oggetto denotato da quel nome. Queste associazioni possono riguardare vari tipi di oggetti (variabili, funzioni,...) ma ciò che viene considerato ai nostri fini saranno solo le variabili.

Ciò che si è realizzato può quindi essere definito come un sottoinsieme dell'ambiente che contiene solamente le associazioni nome-oggetto denotato riguardanti le variabili.

3.2.1 Il dato

JOLIE contempla alcuni tipi di dato nativi tipici dei più comuni linguaggi di programmazione (int, string,...) e, sempre comunemente a questi linguaggi, offre la possibilità di strutturare i dati. Quest'ultima caratteristica contraddistingue però JOLIE poichè in questo caso per il linguaggio, non è importante solo il valore del campo, ma anche il campo stesso.

Questo ha comportato l'adozione di una particolare attenzione nella gestione dei dati, sia per quanto riguarda il loro tipo, ma soprattutto per le questioni riguardanti la loro rappresentazione.

Per approfondimenti ulteriori su questo argomento, rimando alla tesi del mio collega Monzali Andrea.

3.2.2 Sequence

Questa pseudo-attività non viene sempre esplicitamente creata quando necessario ma merita un piccolo approfondimento per il suo ruolo di spicco.

Ogni attività (che approfondiremo nel prossimo paragrafo) è sempre contenuta in una sequence, quindi vi sarà una sequence principale che rappresenterà il workflow e altre sequence contenute in quelle attività che a loro volta ne possono contenere altre.

3.2.3 Le activity

L'*activity* (attività) è, come detto nel capitolo precedente, il costrutto base di jEye. Per la realizzazione della nuova caratteristica si è ritenuto opportuno operare una suddivisione concettuale tra activity container e non.

Le prime sono attività che possono a loro volta contenere una sequence che, come detto nel paragrafo precedente, contiene altre attività (come for, loop, parallel,...) e quindi devono essere trattate in modo specifico per ottenere il risultato richiesto.

Le attività non container invece sono quelle activity diciamo "semplici" che operano solo la loro elaborazione senza l'ausilio di altre attività all'interno di esse. Per tale "natura" queste richiedono un'elaborazione basilare per estrapolare l'ambiente.

Per una definizione più formale si rimanda alla trattazione di Monzali Andrea.

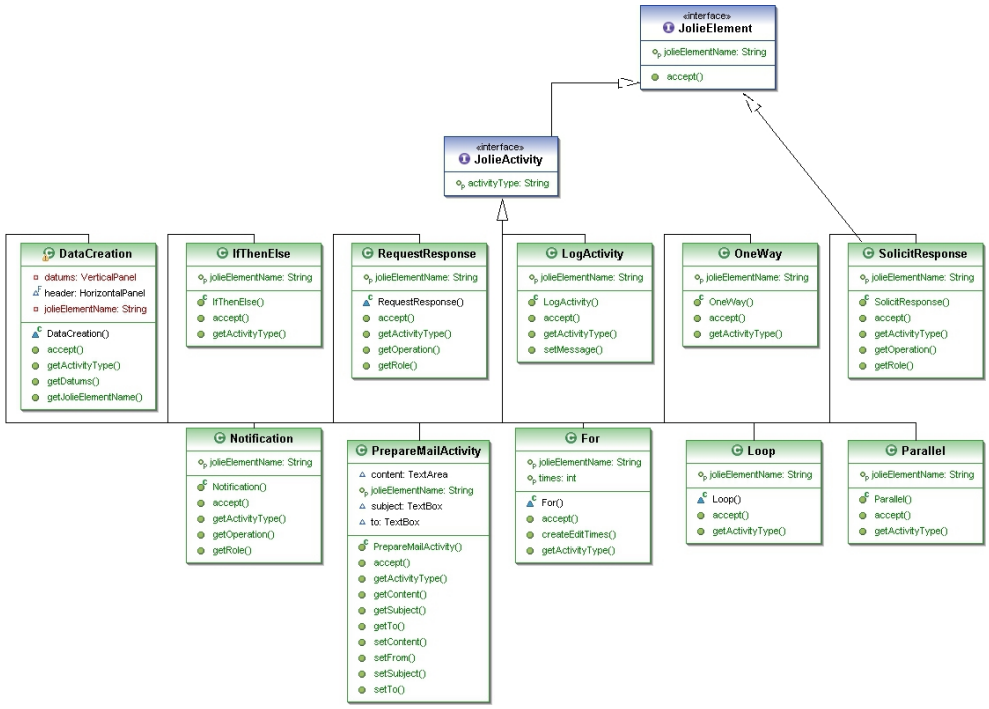

Figura 3.3: Activity presenti in jEye

3.3 Scelte progettuali

A monte del processo implementativo sono state prese decisioni sulla base di considerazioni fatte sul "panorama" di jEye. Le scelte principali hanno riguardato il metodo per la scansione del flusso di esecuzione (*workflow*), il salvataggio dei dati di insteresse e il recupero e presentazione dei dati all'utente.

3.3.1 Scansione del workflow

La scelta del metodo di scansione del workflow è diretta conseguenza della categorizzazione concettuale delle *activity* in activity container e activity semplici. Come già detto sopra, le prime sono activity che possono contenere, a loro volta, una *sequence* e quindi altre activity e che, contribuiscono all'arricchimento dell'environment anche con le attività contenute in esse. Le

altre invece sono normali activity che "ampliano" l'ambiente solo con il loro contributo.

Altro aspetto, non secondario, che ha influenzato la decisione finale sulla scansione del flusso, è stato il ragionamento sul concetto stesso di workflow che si può notoriamente sviluppare ad albero. La diretta conseguenza di questo sviluppo si manifesta come il noto problema della predizione del flusso di esecuzione effettivo, quindi si è scartata l'ipotesi di visitare l'albero dalla radice alle foglie, ma si è necessariamente convenuto sulla "risalita" dall'elemento alla radice.

3.3.2 Salvataggio dati

La decisione sulle modalità di salvataggio dei dati riguardanti l'ambiente è stata di facile risoluzione pur richiedendo un breve studio del meccanismo e del possibile utilizzo da parte dell'utente. La scelta finale è stata quella di rigenerare, ad ogni richiesta, il data environment. Questo sia per non introdurre un meccanismo di monitoraggio sui cambiamenti in tempo reale sia per intaccare il meno possibile il codice originale. Tutto questo ha contribuito a non incrementare il carico di lavoro nel normale utilizzo del tool.

Un secondo aspetto considerato è stato la gestione dei duplicati che, data la "giovane età" dello strumento, non ha una politica ben definita. Ciò ha portato all'introduzione di un vincolo come citato nella tesi "parallela" del mio collega Monzali Andrea sull'univocità dei nomi di variabile.

3.3.3 Recupero e presentazione dati

L'ultimo punto saliente di questo progetto è stato, non tanto il recupero dei dati, quanto la rappresentazione da proporre all'utente per poter utilizzare questa caratteristica aggiuntiva. Come si è detto sopra il data environment viene rigenerato ad ogni richiesta, richiesta che viene generata dalla pressione di un pulsante posto in corrispondenza di ogni attività. Quando l'ultilizzatore vorrà visualizzare l'ambiente a disposizione cliccherà questo pulsante che

innescherà la scansione del workflow in cerca dei dati interessati, il salvataggio di questi in un "database" e, infine, questo "database" sarà scandito contribuendo alla generazione del popup che darà modo di esporre all'utente il risultato dell'elaborazione.

3.4 Scelte implementative

Il processo di sviluppo procede con la fase di design che prevede la scelta del metodo con cui introdurre la nuova funzionalità con un approccio *divide et impera*.

In questa fase della progettazione vengono introdotti i sopra citati design pattern più adatti alle esigenze e che meglio si integrano (in questo caso) con quelli già in essere. In questo delicato e importante passaggio vengono introdotti implicitamente anche vincoli e modalità per il successivo sviluppo di caratteristiche che dovranno, appunto, "convivere" con le nostre scelte.

3.4.1 Scansione del workflow

L'implementazione della "risalita" dell'albero di esecuzione è stata introdotta tramite l'utilizzo del paradigma di design (*design pattern*) *visitor*. Si ricorda che questo design pattern comportamentale permette di separare un algoritmo dalla struttura dati a cui deve essere applicato. Ciò porta all'ovvio vantaggio di poter aggiungere funzionalità ad una struttura senza doverla necessariamente modificare.

A rafforzare l'idea che fosse la giusta strategia in questo caso, è stato il fatto che ogni activity fosse già predisposta per questo approccio. In questo scenario è quindi bastato qualche minimo adattamento (realizzato tramite una classe astratta) e l'implementazione ha richiesto uno sforzo minimo.

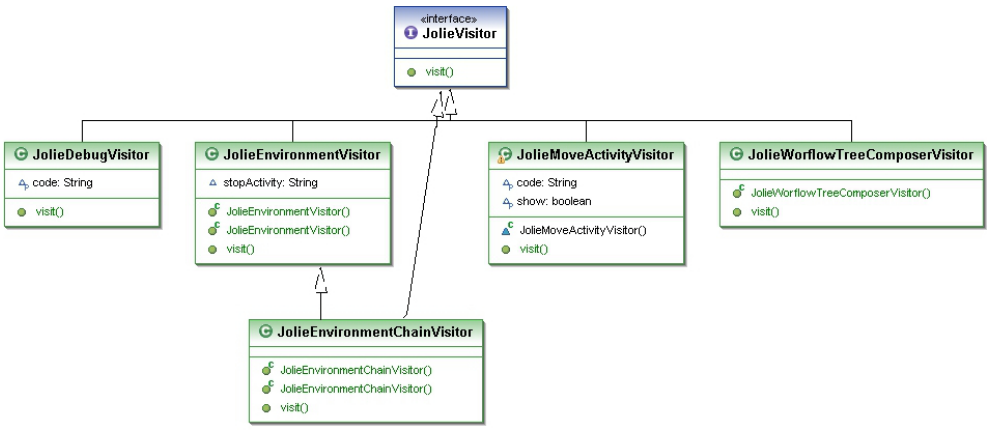

Figura 3.4: Le classi che compongono il visitor per l'Ambiente Dati insieme agli altri visitor già presenti

3.4.2 Salvataggio dati

In primo luogo è sorta la necessità di trasporre il modello dei ruoli (cioè i tipi di dato) in una rappresentazione attraverso classi java. Il pacchetto DataType sopperisce appunto a questa necessità e consente di ricreare tipi di dato anche complessi per poi essere riferiti sucessivamente.

Per la generazione della "base di dati" si è optato per il design pattern *singleton*, per il concetto di ambiente unico a cui ogni attività aggiunge il proprio "contributo".

3.5 Implementazione

Infine l'ultima fase di sviluppo, la scrittura concreta del codice sorgente. Ovviamente quest'ultimo passaggio comprende anche il *debug* e il *testing* del lavoro svolto in modo da poter essere fruibile.

3.5.1 Scansione del workflow

Prima di tutto si è ritenuto opportuno creare l'interfaccia *EnvironmentComposer* per poter raggruppare tutte le classi (prevalentemente le attività) in modo da poter avere un riferimento comune. Realizzato questo, per essere in linea con le scelte progettuali prese, si è creata la classe astratta *AbstractEnvironmentComposer* (che implementa l'interfaccia suddetta) che introduce la proprietà *container* e relativi *getter* e *setter* la quale servirà per tenere traccia della *sequence* in cui è contenuta ogni activity realizzando così il concetto di activity container.

La risalita del workflow viene implementato con queso meccanismo:

1. l'utente richiede l'environment cliccando sul pulsante
2. se il mio container è nullo allora passo al punto 3 altrimenti al punto 4
3. (è stato richiesto l'ambiente alla root sequence) genero l'ambiente partendo dalla prima attività fino all'attività richiedente (esclusa) (all'ultima compresa se non si è passati per il punto 4)
4. (è stato richiesto l'ambiente a una attività diversa dalla root sequence) richiedo la generazione dell'ambiente fino all'attività richiedente (esclusa) al mio container che a sua volta eseguirà il controllo al punto 2 una volta soddisfatta la richiesta precedente

Ovviamente il principio della risalita del workflow è rispettato ma viene implementato con salite e discese continue.

3.5.2 Salvataggio dati

Durante la scansione dell'albero di esecuzione, il visitor *JolieEnvironmentChainVisitor* si occpua anche di recuperare le informazioni necessarie alla generazione dell'environment secondo il modello esplicitato nella tesi del mio collega Monzali Andrea. I dati necessari vengono salvati nella tabella Hash *envDataMap* che contiene le coppie (DataInstance, tipo).

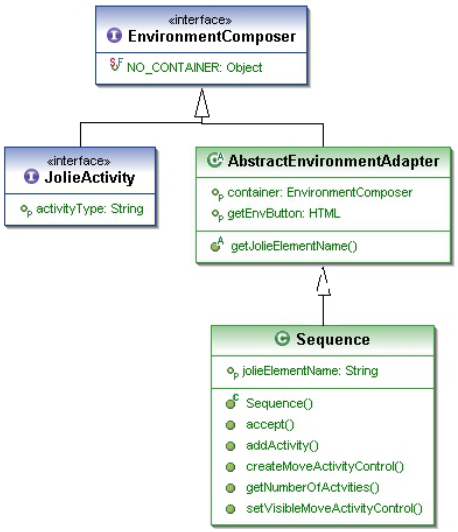

Figura 3.5: Classi aggiuntive per la compatibilità

Il pacchetto DataType è stato implementato per mappare i tipi di dato in strutture Java e si compone delle seguenti classi principali:

- **Type**: rappresenta il tipo di dato, caratterizzato da un NativeType (dato nativo) da cui deriva e un array di possibili SubType (sottotipi)

- **Cardinality**: rappresenta la cardinalità di un elemento all'interno di un tipo.

- **NativeType**: rappresentato da una enumerazione java, corrisponde ai tipi nativi presenti in Jolie

- **SubType**: implementa la possibilità di creare tipi non nativi all'interno di tipi non nativi a loro volta

- **DataInstance**: corrisponde alla trasposizione dell'istanza di dato, quindi caratterizzato dal suo nome, indice di vettore e tipo

3.5.3 Recupero e presentazione dati

Nella classe astratta *AbstractEnvironmentComposer* è stato integrato anche il bottone per la richiesta dell'ambiente. A questo, per poter ottenere l'effetto desiderato, è stato collegato un *handler* al click che inizializza il visitor per la generazione dell'ambiente e ne "lancia" l'esecuzione. Terminato il lavoro del visitor, sempre all'interno dell'handler, viene reperita la tabella hash contenete i dati relativi all'ambiente e, uno ad uno attraverso un ciclo, vengono "impaginati" in un popup che viene presentato all'utente.

Conclusioni

Il lavoro si è svolto come si sarebbe sviluppata una commessa nel mondo del lavoro. Ciò ha dato la possibilità di confrontarsi con uno scenario reale, così si sono svolti incontri con uno dei titolari di Italiana Software per la definizione dei bisogni e per presentare le fasi del prodotto.

Attraverso il primo incontro abbiamo definito i bisogni e i possibili scenari di utilizzo che il committente avrebbe desiderato. Questi sono stati elaborati e schematizzati in un *modello matematico* [4] per uno studio più agevole, fino ad ottenere i tre problemi principali su cui operare. Il tutto è stato considerato anche unitamente al progetto jEye già realizzato in precedenza.

Successivamente alla definizione delle problematiche principali si è cercato il metodo per la risoluzione ottimale per ognuno degli aspetti. In questo passaggio (solitamente a carico degli sviluppatori, aspetto non interessante dal punto di vista del cliente) lo scopo primario è stato la progettazione della feature ed il metodo di integrazione nell'applicativo già in essere, per poter offrire alla committenza un prodotto non finito ma utile alla conferma o smentita del buon andamento del progetto.

Ottenuta l'approvazione del committente si è proceduti nella fase finale di ottimizzazione, debug e test da parte degli sviluppatori per procedere alla presentazione del prodotto finito in linea con le aspettative del cliente.

Questo progetto, rispetto ad altri in cui la creazione fosse cominciata da zero, è stato particolarmente stimolante poichè ha unito la fase di progettazione ad una di integrazione. Tutto ciò ha ricreato una situazione non solo di rapporto con il cliente nella fase iniziale del lavoro ma ha anche simulato

la fase di mantenimento e aggiornamento di un prodotto software, fase che, come noto, ricopre una grande importanza.

Bibliografia

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

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

[3] Articolo wikipedia Web Service:
http://en.wikipedia.org/wiki/Web_service

[4] Tesi di Monzali Andrea: "Supporto a edito web-based in ambiente service oriented: Modellazione dell'ambiente dati"

[5] W3C working group note on Web Service:
http://www.w3.org/TR/ws-arch/

[6] SOA Reference Model OASIS:
http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

[7] W3C note WSDL:
http://www.w3.org/TR/wsdl

[8] W3C Reccomendation SOAP:
http://www.w3.org/TR/soap/

[9] W3C note URI:
http://www.w3.org/TR/uri-clarification/

[10] Articolo wikipedia Orchestration:
http://en.wikipedia.org/wiki/Orchestration_(computing)

[11] Articolo wikipedia Choreography:
http://en.wikipedia.org/wiki/Service_choreography

[12] Articolo wikipedia Choreography:
http://en.wikipedia.org/wiki/Web_Service_Choreography

[13] Sito internet progetto JOLIE:
http://www.jolie-lang.org/

[14] Sito del progetto jEye:
http://sourceforge.net/projects/jeye/

[15] Articolo wikipedia:
http://it.wikipedia.org/wiki/Ambiente_(programmazione)

Ringraziamenti

Ringrazio Monzali Andrea per la collaborazione e il prof. Gabbrielli Maurizio (relatore) e Claudio Guidi per la disponibilità e il il tempo che mi hanno dedicato. Ovviamente non può mancare un sentito grazie alla studentessa Neri Michela che mi ha aiutato e incoraggiato. Infine, ultimo ma non ultimo, il Dott. Lerose Davide con cui ho affrontato una buona parte di questo percorso di studi.