Realizzazione di un motore d'esecuzione per un linguaggio di orchestrazione di servizi

Blite ovvero BPEL in versione lite

Paolo Panconi

8 febbraio 2009

Indice

1 Architettura interna **7**
1.1 Modello per l'Attività 9
1.2 Esecuzione e parallelismo 11
1.3 Eventi e comunicazione 11

Elenco delle figure

Capitolo 1

Architettura interna

In questo capitolo illustriamo l'architettura interna dell'Engine, cioè del componente software predisposto alla messa in esecuzione dei programmi Blite.

Ricordiamo che nello scenario delineato avremo un Engine per locazione, o se si vuole per nodo di rete, e su ognuno di questi componenti sarà possibile installare o rimuovere definizioni di processi Blite. In pratica un engine gestirà un insieme di definizioni, creando da queste istanze di processi e utilizzerà l'Environment per interagire con gli altri Engine. Dall'environment stesso l'engine verrà notificato riguardo l'accadere di eventi, quali l'arrivo di messaggi indirizzati alle porte delle sue definizioni.

Dal punto di vista logico relazionale abbiamo già individuato le seguenti macro entità e relazioni

Per ogni definizione istallata sull'engine sarà presente un oggetto istanza della classe ProcessManager, che avrà il compito di gestire, nel loro ciclo di vita, le istanze di processo derivate dalla definizione.

Prima di entrare nel dettaglio delle scelte architetturali ricapitoliamo quali sono le caratteristiche peculiari di un sistema che deve gestire programmi per l'orchestrazione di servizi, in modo che sia più facile da una parte comprendere e dall'altra giustificare le scelte fatte.

A nostro vantaggio:

- Un Engine contiene in generale un numero contenuto di definizioni, per cui non ci interessa la scalabilità rispetto alla quantità di definizioni istallate su singolo engine. Tale scalabilità al contrario può essere ottenuta aggiungendo altri engine e installando definizioni su engine diversi.

- Una definizione (o programma) Blite avendo principalmente funzionalità di integrazione avrà una lunghezza generalmente limitata.
- Poiché le operazione fondamentali di un programma di questo genere sono invocazioni remote, le durate delle esecuzioni hanno ordini di grandezza minimi dettati dai tempi caratteristici della rete. Per questo motivo non risulta determinante l'efficienza di esecuzione delle operazioni interne di un processo. Il nostro engine non necessiterà di una particolare ottimizzazione rispetto all'efficienza di esecuzione interna.

al contrario risultano particolarmente critici i seguenti aspetti:

- Per ciascuna definizione potrà essere richiesta la creazione di innumerevoli istanze. La scalabilità rispetto al numero delle richieste remote e quindi di istanze di processo risulta essere un prerequisito fondamentale.
- Se da un lato abbiamo detto che l'efficienza di esecuzione non è una aspetto particolarmente critico, dall'altro però ogni attività interna necessita di un elevato grado di controllo e tranciabilità. Ogni attività deve potere essere eventualmente terminata o abortita. Poiché in generale ogni istanza potrebbe avere un immagine persistente o perlomeno essere soggetta ad una attività di monitoring, l'engine necessiterà di un grado di controllo a livello di singola attività Blite.

Tenendo conto di queste iniziali considerazioni sono state fatte alcune scelte basilari di organizzazione del progetto e l'architettura software è stata basata sulle seguenti specifiche fondamentali:

1. La compilazione di una definizione Blite (che eventualmente in un ambiente distribuito può essere fatta in fase di deploy) produce un modello statico della definizione stessa. Tale modello può essere implementato con una struttura ad oggetti che si può pensare di mantenere in memoria presso l'engine, tale struttura sarà navigata a runtime per ricavare il flusso e la logica di esecuzione. Sempre in fase di deploy l'engine può ricavare tutte le informazioni per popolare le strutture dati in cui sono memorizzati i binding fra i nomi delle porte e le definizioni; anche tali strutture dati possono essere mantenute in memoria.
2. Le richieste che giungono all'Engine non devono produrre un aumento delle risorse complessive mantenute dall'Engine. Ogni istanza di processo nel suo svolgersi deve, man mano che procede, rilasciare le risorse

0

di memoria acquisite. Anche il numero dei thread complessivo deve essere limitato superiormente (generalmente dell'ordine dell'unita). La realizzazione del parallelismo di attività deve essere attuata tramite il pattern "Resources Pool". Ogni Engine deve disporre di un pool di thread con cui eseguire in parallelo le attività secondo le definizioni Blite.

3. Il modello di esecuzione deve essere Activity Centric. L'engine deve trattare ogni attività secondo una astrazione generica che possa permettere di fattorizzare i comportamenti comuni e mantenere semplice e pulita l'implementazione della semantica di esecuzione del linguaggio.

1.1 Modello per l'Attività

A questo punto dopo aver esposto a grandi linee quelle che devono essere le caratteristiche fondamentali di un engine entriamo nel dettaglio del disegno della architettura. Nella realizzazione di questa abbiamo scelto di utilizzare il formalismo degli oggetti e delle classi secondo il consueto paradigma "Object Oriented". Inoltre abbiamo preso come fonte di ispirazione il "Composite Pattern" [GANGo4] cercandone una trasposizione nella problematica dell'esecuzione di un programma Blite. In particolare l'astrazione di componente è stata applicata all'entità attività. Come i componenti contribuiscono alla realizzazione di un documento o di una interfaccia utente le singole attività contribuiscono allo svolgersi dell'esecuzione del processo Blite.

Inoltre la tipica struttura gerarchica presente staticamente negli elementi sintattici di una definizione può essere naturalmente riprodotta a runtime tra i singoli step di esecuzione, andato a completare l'analogia con le strutture gerarchiche ad albero tipiche dei tradizionali domini di applicazione del Composite Pattern.

L'entità fondamentale del nostro dominio applicativo è stata quindi individuata nella ActivityComponent trasposizione a runtime dell'elemento sintattico Activity definito dalla grammatica di Blite. Ogni ActivityComponent è rappresentabile tramite la seguente interfaccia

Listing 1.1: L'interfaccia base del modello di escuzione del Blite Engine

```
package it.unifi.dsi.blitese.engine.runtime;

import it.unifi.dsi.blitese.parser.BltDefBaseNode;

/**
 * The base unit of runtime execution of a Runtime Process Instance.
 * The method <tt>doActivity</tt> it the key of the execution model.
 *
```

```
 * @author panks
 */
public interface ActivityComponent {

    public boolean doActivity();

    public ActivityComponent getParentComponent();

    public BltDefBaseNode getBltDefNode();
}
```

ActivityComponent	
boolean **doActivity()**	Costituisce il metodo centrale per lo svolgersi dell'esecuzione del programma. L'invocazione di tale metodo su un oggetto attività fa si che essa possa eseguirsi. Il valore booleano ritornato sarà il discriminate del fatto che il flusso di esecuzione corrente dovrà o meno interrompersi. Ogni attività oltre che eseguire se stesa sarà quindi anche responsabile nel guidare il flusso nel passo successivo. Utilizzando la gerarchia a lei nota imposterà la nuova attività corrente da eseguire (l'attività padre o un figlio) e ritornerà il valore true. Al contraio potrà interrompere il flusso corrente ritornando false.
ActivityComponent **getParentComponent()**	Tale metodo restituisce se presente l'elemento padre dell'attività corrente. In questo modo si realizza la struttura gerarchica fra i veri componenti dell'esecuzione.
BltDefBaseNode **getBltDefNode()**	Ogni attività componente dell'esecuzione è strettamente associata ad un elemento sintattico del programma. Con questo metodo ogni oggetto attività restituisce il nodo che la definisce nell'albero sintattico ricavato dal parsing del codice Blite.

Lo scenario che si va a delineare è quindi quello di due strutture gerarchiche associate: una costituita dall'AST (Abstract Syntax Tree) ricavato dal parsing del codice Blite, e che come si detto è mantenuta nella sua interezza, l'altra costituita dall'albero dinamico delle ActivityComponent che realizzano l'esecuzione a runtime. Quest'ultima struttura, una per ogni istanza, non è pero costruita in un unico momento in fase di inizializzazione del processo, ma al contrario è istanziata man mano che l'esecuzione procede. Come già accennato le attività stesse saranno responsabili di creare i loro successori e di metterli in esecuzione. Inoltre gli oggetti attività già eseguiti dovranno essere rilasciati il prima possibile in modo da poter essere

collezionati dal Garbage Collector e rilasciare le risorse di memoria.

Per ogni tipologia di attività prevista dalla grammatica di Blite esisterà una sotto classe specifica implementante l'interfaccia ActivityComponent e che realizzerà in maniera opportuna in rispetto della semantica il metodo boolean **doActivity()**. Per ottimizzare il disegno e fattorizzare il codice comune è stata ovviamente introdotta una classe astratta ActivityComponentBase da cui ogni altra implementazione di ActivityComponent erediterà le funzionalità comuni di base.

Anche la classe ProcessInstance, che modellerà con i suoi oggetti le varie istanze di processo nell'engine, implemetra l'interfaccia ActivityComponent uniformando la struttura gerarchica di esecuzione.

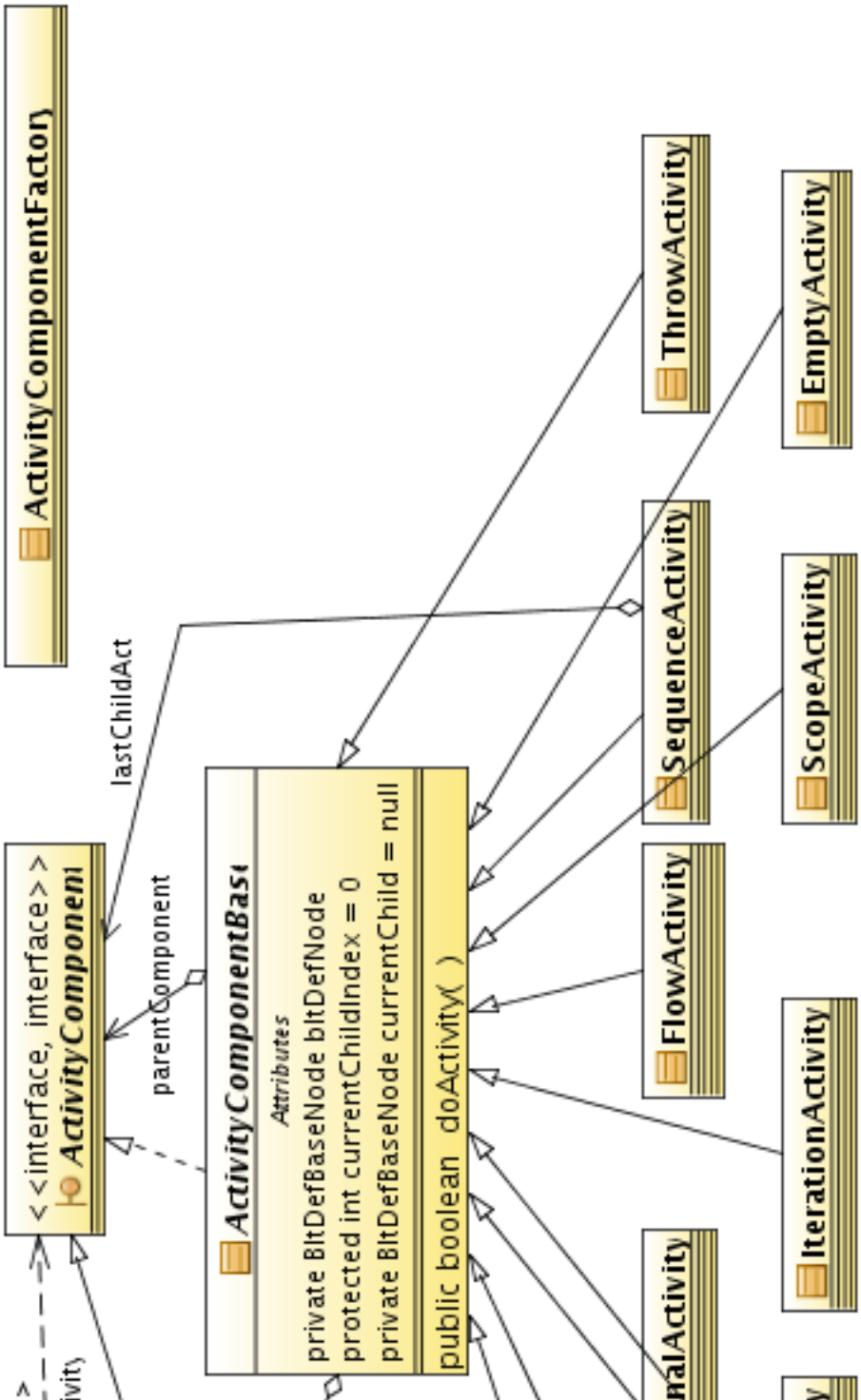
Le varie instanze di ActivityComponent del tipo specializzato verranno create tramite una classe di Factory ActivityComponentFactory che espone il FactotyMathod ActivityComponent **makeRuntimeActivity**(BltDefBaseNode bltDefNode,...)

Listing 1.2: ActivityComponentFactory la factory per le ActivityComponent label

```
/**
 * Factory class to create different ActivityComponent implemetation Objects.
 * @author panks
 */
public class ActivityComponentFactory {

    private static final ActivityComponentFactory SINGLETON =
                                                    new ActivityComponentFactory();

    private ActivityComponentFactory() {}

    /**
     * gets singleton instance
     * @return ActivityComponentFactory
     */
    public static ActivityComponentFactory getInstance() {
        return SINGLETON;
    }

    public ActivityComponent makeRuntimeActivity(BltDefBaseNode bltDefNode,
                                                 ExecutionContext context,
                                                 ActivityComponent parentComponent,
                                                 FlowExecutor executor) {
        ...
    }
}
```

1.2 Esecuzione e parallelismo

1.3 Eventi e comunicazione


ActivityComponentFactory
lastChildAct
parentComponent
<<interface, interface>>
ActivityComponent
ActivityComponentBase
Attributes
private BltDefBaseNode bltDefNode
protected int currentChildIndex = 0
private BltDefBaseNode currentChild = null
public boolean doActivity( )
ThrowActivity
EmptyActivity
SequenceActivity
ScopeActivity
FlowActivity
IterationActivity