Politecnico di Milano
V Facoltà di Ingegneria
Corso di laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

COMPATIBILITÀ, SIMILARITÀ E QUALITÀ NELLA COMPOSIZIONE DEI WEB MASHUP

Relatore: Prof. Maristella MATERA
Correlatore: Prof. Cinzia CAPPIELLO
Correlatore: Dott. Matteo PICOZZI

Tesi di Laurea di:
Massimo GATTI Matr. 755459
Maddalena LOSI Matr. 749539

Anno Accademico 2010-2011

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

La diffusione del Service Orientied Computing (SOC) ha fornito una grande quantità di servizi distribuiti e riutilizzabili. SOC ha dunque introdotto uno spostamento del paradigma di sviluppo software, specialmente nel dominio del Web, da uno sviluppo software orientato ai prodotti, ad una composizione di servizi orientata al consumatore. Questo cambiamento (o evoluzione) promuove la composizione di nuove applicazioni partendo da una grande disponibilità di servizi riutilizzabili. La composizione di servizi è stata tradizionalmente corredata da standard e tecnologie potenti (ad esempio BPEL, WSCDL, ecc) che comunque possono essere padroneggiate esclusivamente da esperti IT, capaci di programmare delle applicazioni. Soluzioni per una composizione di servizi orientate all'utente finale sono state scarsamente esaminate.

Recentemente, nel contesto del Web, è emerso un nuovo paradigma di composizione: i *Web mashup*. I mashup sono un particolare tipo di applicazioni Web che consentono l'integrazione, a differenti livelli, di servizi o API messi a disposizione in rete. I mashup sono composti da parti atomiche chiamate componenti che, sincronizzati e debitamente orchestrati, consentono l'integrazione di diversi contenuti generando un'applicazione a valore aggiunto.

I componenti possono essere di diverse nature e tipologie. In particolare i componenti possono essere tipicamente Web services, API remote, servizi proprietari (locali o remoti), per esempio servizi di accesso a database. Per poter essere integrato nelle pagine Web in generale, ed in particolare nelle piattaforme dedicate alla composizione dei mashup, ogni componente tipicamente necessita di un *wrapper*,

in grado di valutare l'interfaccia per l'invocazione dei suoi metodi ed eventuali descrizioni e annotazioni. Un wrapper è tipicamente uno script JavaScript che serve a mappare le funzionalità di un servizio in modo che possano essere effettivamente utilizzate. Attualmente nei mashup che si possono trovare in rete, per esempio su ProgrammableWeb.com [32], i componenti non hanno una struttura anche dal punto di vista dell'implementazione e il codice dei wrapper è *nascosto* all'interno dell'applicazione o pagina Web.

I mashup stanno progressivamente diventando una soluzione alternativa alla composizione di servizi, che può aiutare a realizzare il sogno di un web programmabile persino da utenti che non sono programmatori (cioè da parte di coloro che solitamente sono ascritti alla categoria di utenti finali, che si servono dei contenuti, ma che non ne creano, in quanto non posseggono le conoscenze tecniche adeguate). Ci sono tante ricerche sugli strumenti di mashup, i così detti *mash-makers*, che forniscono interfacce grafiche per comporre servizi di mashup senza chiedere all'utente di scrivere del codice.

Dal punto di vista della programmazione orientata all'utente finale, permettere ad una classe allargata di utenti (non solo gli sviluppatori esperti) di creare le proprie applicazioni richiede la disponibilità di astrazioni intuitive e strumenti di sviluppo semplici, e un alto livello di assistenza. Alcuni progetti (ad esempio ServFace [18] e SOA4All [30]) si sono focalizzati sul semplificare la creazione di un'efficace presentazione dei servizi, per fornire un canale diretto tra l'utente e il servizio (offrendo spesso un'interfaccia di servizio programmabile). Questo approccio non ha comunque permesso la composizione di servizi multipli in un'applicazione integrata.

Anche in un contesto aziendale, dove la tendenza attuale è di fornire all'utente strumenti per la costruzione di applicazioni chiamate *situational application* ad esempio applicazioni che svolgono una funzione analitica ben precisa e sono sviluppate per un orizzonte temporale limitato, si trovano approcci analoghi a quelli descritti poco sopra. I mashup aziendali sono il supporto per l'attuale approccio mashup nelle internet aziendali, permettendo ai membri dell'impresa l'utilizzo di servizi interni, definiti dal dipartimento di IT per accedere alle informazioni finanziarie dell'impresa stessa, e di servizi pubblici, nonché di poterli comporre in modo innovativo e utile ad esempio per automatizzare alcune procedure buro-

cratiche. Se supportato da strumenti adeguati, il paradigma di mashup consente la programmazione orientata all'utente: utenti non esperti possono creare la loro applicazione analitica senza l'aiuto di uno sviluppatore esperto. Questo *modus operandi* incrementa la produttività. Inoltre ci sono molte funzionalità di filtraggio e composizione di informazioni che non sono supportate da applicazioni aziendali di lungo periodo, questo è dovuto per esempio a specifiche necessità e/o preferenze che caratterizzano le attività dei singoli individui, oppure per problemi di business inaspettati.

Strumenti adeguati possono dunque fare uso di raccomandazioni che possano assistere l'utente finale durante la composizione del mashup, ad esempio suggerendo l'inclusione di determinati servizi, che possano migliorare la qualità del mashup. Questa tesi propone una soluzione a questo problema e definisce metodi per la valutazione della similarità e della qualità di ogni singolo componente con lo scopo di favorire e agevolare l'utente durante la composizione.

Una ricerca precedente si è focalizzata sullo sviluppo di un ambiente di mashup, DashMash [13], per l'integrazione di componenti eterogenei (non solo dati e Rss feeds) a livello di presentazione. Il progetto di mashup è basato su una notazione visuale, attraverso la quale l'utente DashMash può visivamente, in modo intuitivo, creare la composizione del mashup. La composizione visuale genera automaticamente un modello di composizione che poi dirige l'esecuzione del mashup.

Nel contesto di questo progetto, il nostro lavoro di tesi si è concentrato sulla definizione di algoritmi per la generazione di raccomandazioni a supporto della composizione. Tali raccomandazioni hanno l'obiettivo di supportare l'utente nella scelta di componenti *adeguati*, e cioè *compatibili* sintatticamente e semanticamente con gli altri componenti già presenti nella composizione, e in grado di massimizzare la *qualità* della composizione.

1.1 La semantica dei componenti

Per quanto riguarda l'ambito della semantica si può dire che i Web service semantici [31] rappresentano la prossima generazione di Web service [1], creati per supportare automaticamente la scoperta, la corrispondenza e la composizione dei

Web service stessi. In assenza di una componente semantica è decisamente complesso eseguire questi compiti automaticamente, dal momento che le descrizioni dell'interfaccia dei Web service sono spesso complesse e criptiche, specialmente quando questi servizi sono generati wrappando codice di tipo legacy. La presente tesi si pone l'obiettivo di migliorare e approfondire la qualità della corrispondenza e della descrizione automatiche dei Web service. Le pratiche correnti prevedono di descrivere un servizio come una collezione di operazioni, dove nella maggior parte dei casi, un'operazione viene definita tramite il suo identificativo (ad esempio nome dell'operazione, nomi e tipi dei parametri in input ed in output). In questo contesto si considera che due componenti siano corrispondenti se e soltanto se lo sono le loro rispettive operazioni; a loro volta sono corrispondenti le operazioni se corrispondono i loro nomi e i loro parametri in input ed in output. Pertanto il problema della corrispondenza di Web service si riduce al problema della corrispondenza tra nomi (ad esempio gli specifici tag), tipi e strutture.

All'attuale stato delle cose si individuano tre principali approcci al problema: il primo prevede l'esatta corrispondenza tra i tag, il secondo si basa sulla corrispondenza tra tag indipendenti dal dominio (ad esempio tramite l'utilizzo di dizionari), mentre il terzo prevede la corrispondenza dei tag dipendenti dal dominio (ad esempio coadiuvandosi con le ontologie [43]). Il primo approccio consiste nel cercare di individuare Web service che usino esattamente le medesime parole per descrivere concetti similari. La qualità dei Web service che sono corrispondenti è molto elevata, tuttavia le possibilità di individuare questa sorta di corrispondenze perfette sono molto basse. La seconda e la terza tecnica permettono invece l'uso di parole correlate (ad esempio sinonimi e concetti correlati come definito dai dizionari e dalle ontologie rispettivamente) per descrivere gli stessi concetti. Risulta perciò chiaro che le possibilità di individuare Web service corrispondenti sono decisamente più elevate, anche se la qualità dei Web service corrispondenti ne risente e non è sempre così perfetta.

Queste tecniche sono dunque risultate fruibili in alcuni ambiti del web semantico, ma inefficienti e limitate perché trattano ogni singolo tag come un gruppo di parole, senza però considerare le relazioni intercorrenti tra le parole. Questa tesi è volta per prima cosa a descrivere la metodologia basato sull'utilizzo delle ontologie, che si pone anche il compito di raffinare l'approccio basato sull'utilizzo

di un dizionario catturando le relazioni tra i token all'interno dai tag. L'idea è quella di disambiguare le corrispondenze di questi tag cogliendo le relazioni tra le parole all'interno di una ontologia e successivamente utilizzare l'ontologia stessa per guidare il processo di corrispondenza.

Inoltre gli obiettivi si rendono più specifici nel momento in cui entrano in gioco molteplici aree semantiche connesse a loro volta ad altrettante ontologie. Si rende pertanto necessario coordinare l'analisi dei componenti servendosi sia del dizionario WordNet [36] sia di un range di ontologie facilmente reperibili in rete in modo tale da rendere il più generale possibile tutta l'analisi e da realizzare un'architettura che sia agevolmente estensibile. È importante condurre una adeguata analisi semantica per favorire la componentizzazione e per costituire una base per l'analisi di qualità.

1.2 La qualità del mashup

A fronte di una opportuna analisi semantica, che individui componenti tra loro simili, si può ulteriormente guidare l'utente in scelte positive fornendo una dettagliata metodologia di analisi qualitativa dei componenti stessi. Fino ad ora i lavori di ricerca presenti in letteratura si sono focalizzati sulla definizione delle tecnologie che abilitano la composizione di mashup, mentre pochi sforzi sono stati rivolti ai concetti inerenti alla qualità di questo tipo di applicazioni. Da un lato i mashup non sono altro che applicazioni Web, quindi si potrebbe assumere che la loro qualità possa essere studiata con i ben consolidati metodi dell'ingegneria del Web. Nonostante tutte queste osservazioni, noi crediamo che questo specifico tipo di applicazioni derivi da particolari dinamiche che ne caratterizzano il processo di sviluppo e che richiedono, pertanto, un'analisi specifica e dedicata. Una delle caratteristiche fondamentali dei mashup è l'integrazione di servizi pronti all'uso (e possibilmente ad accesso pubblico), costituenti i mattoni su cui è costruito questo tipo di applicazioni. L'integrazione è quindi un fattore chiave, che introduce innovazione e, molto spesso, determina il successo nell'utilizzo di un mashup. Tuttavia, come ogni altro sistema *integrato*, la qualità dei singoli servizi incide fortemente sulla qualità della composizione finale. Questo aspetto si riscontra chiaramente se si considera che, omettendo tutte le logiche di sincronismo e coreografia, le fun-

zionalità dell'applicazione finale e il suo comportamento derivano direttamente dal comportamento dei singoli servizi. Per questo la valutazione della qualità del mashup deve partire dalla valutazione qualitativa di ogni singolo componente. La qualità della composizione necessità anche di particolare attenzione [21]: quando integrati in un mashup infatti, i servizi possono giocare differenti ruoli che incidono sulla percezione della qualità globale da parte dell'utente; ciò fa capire che la qualità dell'integrazione non può essere una banale aggregazione della qualità dei singoli componenti, ma deve tenere conto del loro ruolo e del loro peso nel concerto collettivo di tutti i componenti dell'applicazione.

In questo lavoro di tesi verrà presentato un approccio alla valutazione della qualità di un mashup in cui differenti prospettive di qualità, e la relativa valutazione, sono integrati nel ciclo di vita dei mashup stessi, costruendo un processo di sviluppo che tenga conto del dato qualitativo. Descriveremo quindi inizialmente un modello di qualità per i componenti di un mashup e identificheremo come la qualità dei singoli componenti, rappresentata da un'insieme di attributi di qualità, possa essere sfruttata per la selezione di servizi che producano mashup di un certo livello di qualità e stiano alla base della valutazione della qualità del mashup finale. In generale, osservando le pratiche correnti, è possibile notare come durante la composizione di un mashup la selezione di servizi adeguati sia basata su requisiti funzionali, non considerando la qualità del servizio stesso. In questo lavoro di tesi, mostreremo invece che la qualità dei servizi selezionati, congiuntamente al modo in cui sono composti, ha un impatto diretto sulla qualità totale del mashup finale. In accordo con ciò, mostreremo come i criteri di qualità che caratterizzano i mashup possano guidare la selezione dei componenti nei mashup stessi, durante la componentizzazione dell'utente finale.

1.3 Struttura della tesi

In questo paragrafo si descrive brevemente la struttura che si intende dare alla presente trattazione. Procediamo dunque a dare un elenco e una breve descrizione di tutti i capitoli:

- **Capitolo 2** - in questo capitolo viene descritto lo stato dell'arte, cioè lo scenario attuale dei mashup, dell'integrazione dell'UI, delle tecnologie di

composizione e dei tool di sviluppo esistenti, che costituiscono il background della presente tesi;

- **Capitolo 3** - il capitolo terzo si occupa di descrivere nel dettaglio la piattaforma per la quale sono stati realizzati gli algoritmi di compatibilità, similarità e qualità, cioè *Dash Mash*. L'attenzione e posta sulla struttura interna della piattaforma in modo da poter successivamente comprendere come la presente tesi costituisca un intervento volto a migliorarla e arricchirla;

- **Capitolo 4** - in questo capitolo ci si dedica invece a descrivere il funzionamento vero e proprio dell'algoritmo per il calcolo di compatibilità, similarità e qualità e in particolare tutte le sue componenti (compatibilità, similarità e qualità), la sua complessità, le ottimizzazioni ottenute e gli elementi di innovazione, quali l'analisi topologica legata alla teoria delle reti e dei grafi, che supporta la valutazione di qualità per i mashup, visti come un grafo, i cui nodi sono componenti;

- **Capitolo 5** - questa sezione è dedicata invece, a descrivere l'implementazione degli algoritmi a partire dalla descrizione dell'architettura sottostante e da un cenno alle tecnologie utilizzate (le più salienti delle quali saranno poi più accuratamente descritte nell'Appendice A), fino a giungere alla descrizione di alcune prove pratiche volte a valorizzare il front-end dell'applicativo;

- **Capitolo 6** - si dedica una breve sezione alla portabilità del metodo, cioè a come riutilizzare gli algoritmi introdotti anche in altri domini applicativi ed in altre piattaforme;

- **Capitolo 7** - si conclude con un ultima parte dedicata a importanti riflessioni, sul tema trattato, che portano a trarre delle conclusioni ben precise e a porsi altrettanto precisi propositi di futuri sviluppi e miglioramenti. Questa sezione si sviluppa su due binari paralleli:

    1. l'argomentazione a supporto dell'utilizzo delle ontologie, che risulta spesso fortemente osteggiato, ma nel nostro caso si è rivelato imprescindibile;

2. i proponimenti per ulteriori sviluppi futuri, sia nell'approccio ontologico, sia nelle valutazioni relative al modello di qualità e al suo utilizzo;

- **Appendice A** - questa appendice raccoglie una breve rassegna delle tecnologie più importanti, sia per peculiarità, che per innovatività, che hanno condotto il nostro studio.

Capitolo 2

Stato dell'arte

Questo capitolo introduce e descrive brevemente cosa siano i mashup, cosa siano i componenti e come si strutturi la loro coreografia nel mashup. Tutte queste nozioni vengono contestualizzate ampiamente nei primi paragrafi di questo capitolo in cui ci dedichiamo al problema di integrazione dell'interfaccia utente, alle tecnologie di sviluppo e ai tool di composizione preesistenti; infine ci si dedica a esporre con un certo livello di generalità cosa si intenda per annotazioni semantiche, per coi concludere spiegando come si generano raccomandazioni a supporto della composizione dei mashup da parte degli utenti finali.

2.1 I mashup

Nel corso degli anni sono stati effettuati molteplici ricerche e sviluppi nel campo dell'integrazione di applicazioni e, più recentemente, di servizi secondo il paradigma SOA (Service Oriented Architecture). Il riutilizzo e l'integrazione di contenuti Web e di servizi forniti da terzi è già una realtà comunemente conosciuta con il nome di Web mashup [17]. I Web mashup [4, 27] sono siti o applicazioni Web sviluppate integrando interfacce utente, logica applicativa e contenuti (dati) ai quali è possibile accedere dal Web e che possono essere forniti, come già detto, da soggetti terzi. I primi mashup comparsi sulla scena non potevano fare affidamento su API come interfacce di programmazione, dal momento che gli effettivi fornitori di contenuti non erano nemmeno consapevoli del fatto che i loro siti Web fossero inseriti in altre applicazioni. I primi mashup con Google Maps [41], per esempio,

sono precedenti al rilascio ufficiale delle API di Google Maps. Le API per mashup disponibili pubblicamente sono ancora rare, ma in aumento. L'integrazione può essere eseguita "ad hoc", sfruttando qualsiasi linguaggio di programmazione supportato dalla fonte dei contenuti, sia lato client (e.g. AJAX) sia lato server (e.g. PHP, Java, ASP.NET). Il contenuto è tipicamente fornito in codice markup e integrato inserendo il rispettivo sito Web. Dal momento che tali tipi di "interfacce di componenti" non sono in genere molto stabili (si ricorda che il fornitore dei contenuti potrebbe non sapere che la sua applicazione è stata riutilizzata), lo sforzo maggiore nello sviluppo di mashup è dovuto al testing manuale e alla manutenzione. A causa della mancanza di framework di supporto adatti, la stabilità del codice non è assicurata (cioè ci potrebbero essere collisioni tra due codici sorgenti JavaScript), e potrebbero verificarsi conflitti tra componenti UI, cioè dotati di interfaccia grafica. Quindi la costruzione di un mashup Web resta un compito ad hoc lungo e impegnativo e la necessità di modelli di programmazione appropriati è evidente.

2.2 Tecnologie per l'integrazione dell'UI

Nell'integrazione dell'interfaccia utente è possibile individuare quattro temi in particolare:

- definire modelli e linguaggi per le specifiche dei componenti;
- definire modelli e linguaggi per le specifiche della composizione dei componenti;
- scegliere stili di comunicazione per l'interazione dei componenti;
- definire discovery e meccanismo di binding (anche durante il runtime) per l'identificazione dei componenti.

Anche tenendo conto di determinati principi guida, cioè, la semplicità, il formalismo, la leggibilità e la modularità delle specifiche, è possibile individuare alcune possibili soluzioni per la gestione dei componenti e delle interazioni con il loro livello di presentazione, ognuno caratterizzato da un determinato grado di complessità, tecnologia e programmazione cosiddetta *amichevole*:

- modello del componente: le applicazioni di integrazione dei componenti sono caratterizzate da una API (Application Programming Interface) e anche da un modello del componente. Nell'integrazione dei dati, lo schema della sorgente dei dati descrive il componente. In effetti, nell'integrazione dell'interfaccia utente a livello di presentazione, oltre al riutilizzo di classi di librerie, si rivela necessario un modello a componenti in grado di supportare le interazioni complesse e di coordinamento. Ogni singolo componente è descritto da una invocazione di servizi software di abilitazione dell'interfaccia e da un'interfaccia utente che consente l'interazione. Un'interfaccia utente del componente consente diversi livelli di interoperabilità;

- solo-GUI: tutte le interazioni con i componenti GUI di sola esecuzione sono effettuate mediante una componente logica per l'interfaccia utente. L'unico metodo per integrare un componente solo-GUI è conoscere la sua interfaccia utente ed essere in grado di tracciare i tratti di posizione del mouse e di capire cosa la UI del componente mette in evidenza, così da rendere possibile eseguire le azioni che causano la modifica dell'interfaccia utente;

- interfaccia nascosta: in molte applicazioni Web, il componente è dotato di un'interfaccia specifica che consente di controllare la sua interfaccia utente, ma non è pubblicamente descritta. Applicazioni Web di interazione permettono l'accesso e la manipolazione di contenuti e la visualizzazione della risposta mediante l'invio di richieste HTTP. Queste applicazioni obbediscono a un protocollo generale per l'interazione tra client e applicazioni, di solito difficili da identificare;

- interfaccia pubblicata: in questo caso ideale, il componente fornisce una descrizione pubblicata della sua interfaccia utente e una API in modo da poter effettuare manipolazioni a runtime. Un basso livello API potrebbe permettere il controllo dei singoli elementi dell'interfaccia utente. Una API di alto livello potrebbe esporre un set di soggetti e oggetti controllabile, così come le operazioni per modificare lo stato dell'entità.

2.2.1 Linguaggio di composizione

Un linguaggio di composizione sorge a sostegno dell'identificazione e della specificazione degli elementi coinvolti in una orchestrazione e della loro interazione. Per quanto riguarda l'integrazione dei dati, la composizione avviene spesso attraverso una vista SQL che permette di esprimere uno schema globale come un insieme di viste su uno schema locale. Nella integrazione delle applicazioni la composizione può essere descritta sia tramite linguaggi di programmazione general-purpose, come Java, sia attraverso linguaggi dedicati all'integrazione delle applicazioni, come i linguaggi di composizione del workflow o di servizio. Nella composizione dell'interfaccia utente ci potrebbero essere due tipi di soluzioni:

- linguaggi di programmazione general-purpose: gli sviluppatori possono adottare una terza generazione di linguaggi per la composizione delle applicazioni. Tali linguaggi sono molto flessibili, ma mancano di astrazione dei componenti a grana grossa (come strutture per la discovery e per il binding dei componenti o primitive di alto livello per la sincronizzazione dei componenti che l'interfaccia utente mostra);

- linguaggi di composizione specializzati: i linguaggi di alto livello sono tipicamente usati con una sintassi XML su misura per la composizione dei componenti dell'interfaccia utente a livello di descrizioni astratte o esterne. Il vantaggio principale di questi linguaggi è di avere un livello superiore di programmazione per il supporto delle composizioni, che sfrutta le caratteristiche del modello del componente.

2.2.2 Stile di comunicazione

Abbiamo già sottolineato l'importanza della comunicazione per l'integrazione dell'interfaccia utente: la necessità è quella di monitorare gli eventi all'interno di alcuni componenti dell'interfaccia utente, che possono aggiornare lo stato (e quindi, l'interfaccia utente) di altri componenti. È possibile distinguere tra due tipi di comunicazione:

- comunicazione mediata centralmente, in cui l'applicazione composita ha un coordinatore centrale che riceve gli eventi e le istruzioni riguardanti

le richieste per manipolare le interfacce utente dei componenti, dai singoli componenti;

- comunicazione diretta componente a componente, in cui l'applicazione composita è una coalizione di singoli componenti.

Queste soluzioni si differenziano nettamente dall'integrazione dei dati in cui i componenti sono di solito passivi e non avviano la comunicazione con l'applicazione che opera l'integrazione. L'integrazione dell'applicazione mostra la stessa differenza: un soggetto centralizzato (l'applicazione composita) richiama i componenti, se necessario. Un'ulteriore distinzione tra questi tipi di comunicazione in materia di integrazione dell'interfaccia utente è quella tra interazione RPC-style, in cui lo scambio di informazioni attraverso i componenti genera chiamate di metodo e dei dati restituiti, e interazione publish-subscribe, in cui le applicazioni comunicano in modo debolmente accoppiato tramite messaggi scambiati attraverso i messaggi dei broker.

2.2.3 Discovery e Binding

Un problema rilevante in materia di integrazione dell'interfaccia utente è la scoperta (*discovery*) dei componenti coinvolti nella composizione e capire come ottenere il binding, sempre dei medesimi componenti. Ci sono due diversi modi per farlo: il primo consiste nel definire la composizione staticamente in fase di progettazione o nel momento dell'implementazione e la seconda consiste nel fare questo in modo dinamico in fase di runtime.

Osserviamo che nell'integrazione dei dati e delle applicazioni l'associazione tra diverse fonti di dati in genere si verifica in fase di progettazione quando i dati dello schema globale sono definiti e la discovery viene eseguita dinamicamente da un middleware di integrazione. Tuttavia questo approccio manca di flessibilità a causa della difficoltà di interagire con componenti che siano stati recentemente soggetti a discovery e solo successivamente aggiunti all'integrazione iniziale.

Una soluzione ibrida di binding è utilizzata anche nel caso in cui il progettista dell'applicazione identifichi e testi un insieme di componenti potenziali e in seguito gli utenti selezionino un sottoinsieme di essi in fase di runtime in base al

compito che devono affrontare: in questo caso la discovery è statica, ma il riferimento è dinamico. Questo approccio ibrido rende anche possibile l'integrazione dell'interfaccia utente.

2.2.4 Visualizzazione dei componenti

Ci sono diverse soluzioni per la visualizzazione dei componenti, la cui distinzione è basata sul paradigma di rendering dell'interfaccia utente. Infatti, il componente può visualizzare la sua interfaccia utente in un primo tipo di soluzione, mentre in un secondo tipo di soluzione, l'applicazione composita riceve il codice di markup dell'interfaccia utente dei componenti e li renderizza. La seconda soluzione ha bisogno di una descrizione di markup, che deve essere interpretata con un motore di rendering in modo da ottenere una traduzione in elementi grafici: la specifica di markup descrive spesso le proprietà statiche dell'interfaccia utente, mentre i linguaggi di scripting dinamico ne forniscono il comportamento. Tale descrizione può essere fatta con i linguaggi orientati ai documenti o i linguaggi di interfaccia utente che si interfacciano appunto con un sofisticato modello di applicazione. Esempi di tale linguaggio sono XAML [49], XUL [51], UIXML [52], XIML [50]. È possibile avere due diverse soluzioni di rendering dell'interfaccia utente:

- component-rendering dell'interfaccia utente: il componente gestisce il rendering dell'interfaccia utente e il display, quindi l'applicazione composita è una collezione di interfacce utente di componenti. Questo è il caso delle applicazioni desktop classiche che individuano componenti eseguibili legati a librerie grafiche;

- un'interfaccia utente basata su markup: il componente restituisce il codice di interfaccia utente e delega il rendering dell'utente finale sia alle applicazioni composite sia all'ambiente funzionante, in grado di interpretare il codice dell'interfaccia utente dei componenti e di allocare spazio adeguato di layout per il rendering dei componenti stessi. In questo caso, l'interazione dell'utente con il componente può essere gestita direttamente attraverso una logica di creazione di script adatti incorporati nel codice del componente, oppure tramite l'applicazione composita, che è in grado di intercettare gli eventi generati dell'UI e li trasmette al componente per l'interpretazione.

2.3 Tecnologie di composizione

Illustriamo in tabella 2.1 un confronto tra le diverse tecnologie di interfaccia utente considerate nel contesto della composizione UI. Questo confronto è incentrato sui principali aspetti di integrazione UI precedentemente presentati. Le tecnologie a confronto sono brevemente descritte nel resto di questa sezione.

	Desktop UI components	Browser plug-in components	Web portals and portlets
Component model	Published, Programmable API	Basic interface	Public API
Composition language	General-purpose	Document markup code, Javascript	General-purpose
Communication style	Mediated intercomponent	Centrally mediated	Centrally mediated
Discovery and binding	static and dynamic	static	static and dynamic
Component visualization	rendered	rendered	markup-based

Tabella 2.1: Tecnologie di composizione a confronto

2.3.1 Componenti del desktop

Storicamente la composizione di interfaccia utente è nata per le applicazioni desktop, per creare un ambiente in cui le applicazioni sviluppate con linguaggi eterogenei potessero interagire. Pensiamo ad esempio ad ActiveX, che sfrutta la tecnologia Microsoft COM per incorporare una completa interfaccia utente nelle applicazioni host, e l'applicazione composita UI Block(CAB), che è un framework di interfaccia utente per la composizione in .NET con un servizio di contenitore che consente agli sviluppatori di creare applicazioni su moduli caricabili o plugin. In particolare, i componenti CAB possono essere utilizzati con qualsiasi .NET per costruire contenitori compositi ed effettuare comunicazioni componente-container. CAB fornisce un broker di evento per molti-a-molti, con accoppiamento di comunicazione tra i componenti sulla base di un modello publish-subscribe di eventi di runtime. Un altro esempio è Eclipse Rich Client Platform (RCP), che fornisce un simile framework, ma include anche una shell di applicazione di servizi di interfaccia utente. Esso offre inoltre un modulo basato su API che consente

agli sviluppatori di creare applicazioni all'interno di questa shell. Eclipse consente inoltre agli sviluppatori di personalizzare ed estendere i componenti dell'interfaccia utente tramite i cosiddetti punti di estensione, una combinazione di interfacce in linguaggi Java e di markup, come XML. I componenti desktop dell'interfaccia utente in genere utilizzano i linguaggi di programmazione general-purpose per componenti integrati (C # e Java per il CAB per RCP) perché le interfacce dei componenti sono API di programmazione specifiche del linguaggio. I componenti possono gestire i loro propri rendering dell'interfaccia utente, ma potrebbero anche sostenere stili di comunicazione flessibile. Entrambi i binding in fase di progettazione e di esecuzione sono supportati, basandosi su meccanismi di reazione specifici del linguaggio. Molte delle tecnologie per i componenti desktop dell'interfaccia utente sono dipendenti dal sistema operativo. Sebbene CAB e RCP non dipendano direttamente dal sistema operativo, si affidano rispettivamente al loro ambiente di runtime. La mancanza di interfacce che riconoscano la tecnologie e siano descrittive, rende l'interoperabilità tra i componenti attuata con diverse tecnologie difficile da raggiungere.

2.3.2 Componenti del browser

L'esperienza di navigazione nel browser spesso coinvolge caratteristiche di interfaccia utente avanzate. Le principali tecnologie utilizzate per creare componenti di interfaccia utente nascosti in interfacce basate sul markup sono applet Java con controlli di tipo ActiveX. Dopo la definizione del binding dei componenti durante la scrittura della pagina da parte del web designer, il browser scarica i componenti a runtime e li istanzia. Questi componenti spesso rendono disponibile il loro proprio rendering, con una piccola comunicazione ulteriore tra di loro e la pagina web che li contiene. L'interfaccia esterna di questo tipo di componenti è molto semplice e solitamente richiede solo la configurazione di opportuni parametri quando si incorporano i componenti all'interno del codice di markup.

2.3.3 Portali web

Lo sviluppo di portali web distingue esplicitamente tra componenti UI (portlets) e applicazioni composite (portals). I portlets sono componenti di applicazioni

web; generano frammenti di documenti di markup che aderiscono a regole fissate, in modo da facilitare l'aggregazione di contenuti nei portali server per formare in ultimo documenti compositi. I portali server tipicamente fanno sì che gli utenti possano customizzare pagine composite e rendono disponibili svariati tipi di personalizzazioni. Analogamente alle servlet Java, i portlet implementano una specifica interfaccia Java per l'API standard dei portlet, che è stata intesa per aiutare gli sviluppatori a creare portlet che possano conformarsi ai portali server. Per i portlet Java, le applicazioni dei portali sono basate sul linguaggio di programmazione Java, sebbene con le parti web, uno sviluppatore web programma le proprie applicazioni in .NET. L'applicazione portale aggrega tutti gli output di markup dei suoi portlet e gestisce la comunicazione in una tipologia centralmente mediata. I portlet permettono anche sia il binding statico che quallo dinamico; a runtime, l'applicazione portale può rendere i portlet disponibili in un registro per la selezione e il posizionamento da parte degli utenti. Sebbene i portlet e i portali web abbiano obiettivi e architetture similari, non sono interoperabili. I web service per portlet remoti (WSRP) indirizzano questa richiesta al protocollo di livello esponendo portlet remoti come servizi web; la comunicazione tra portali e portlet avviene tramite SOAP, il che significa che gli sviluppatori possono costruire portali e portlet con differenti linguaggi e framework a runtime.

2.4 I Web mashup

I mashup sono particolari applicazioni Web che consentono l'integrazione, a differenti livelli, di servizi o API messi a disposizione in rete. I mashup sono composti da parti atomiche chiamate componenti che, sincronizzati e debitamente orchestrati, consentono l'integrazione di diversi contenuti generando un'applicazione a valore aggiunto.

I componenti possono essere di diverse nature e tipologie. In particolare i componenti possono essere tipicamente Web Services, API remote, Servizi proprietari (locali o remoti) [28] per esempio servizi di accesso a database. Ogni componente necessita di un wrapper per poter essere integrato nelle pagine Web in generale, ed in particolare nella piattaforma.

Per quanto riguarda il modello dell'interfaccia utente del componente e la specificazione esterna i web mashup offrono una API pubblicata, ma un'interfaccia nascosta. Il linguaggio di composizione e costituito da linguaggi di programmazione general-purpose. Lo stile di comunicazione è mediato in maniera centralizzata. In relazione a discovery e binding si evidenzia un binding di tipo statico, mentre circa la visualizzazione dei componenti si può asserire che generalmente essa è basata sul markup.

In quanto alle caratteristiche peculiari di questi componenti enumeriamo compatibilità, similarità e qualità. La compatibilità è una corrispondenza prettamente sintattica e riguarda i tipi dei parametri di input e output e delle operazioni/eventi di ogni componente. Se c'è matching esatto allora i componenti sono compatibili. La similarità valuta sempre i medesimi parametri di cui sopra, però dal punto di vista semantico e viene calcolata tramite l'utilizzo di WordNet e delle ontologie. Infine la qualità è data da un insieme di caratteristiche che vengono assegnate al componente e che ne costituiscono una sorta di strumento valutativo. Tutte queste caratteristiche saranno più ampiamente descritte in seguito.

2.5 Tool di sviluppo per i mashup

Per velocizzare l'intero processo di sviluppo dei mashup, ma anche per consentire agli utenti finali dei mashup (anche inesperti) di creare proprie applicazioni Web, di recente sono emersi numerosi strumenti e framework di sviluppo. Questi strumenti vengono solitamente forniti con una varietà di caratteristiche e una miscela di approcci di composizione. Uno sguardo da vicino ad essi ci permette di identificare le questioni aperte e le sfide della ricerca che caratterizzano il fenomeno mashup. Ai fini della presentazione, abbiamo selezionato gli approcci più popolari o rappresentanti strumenti per creare mashup volti all'utente finale.

2.5.1 Yahoo Pipes

Yahoo Pipes [8] prende uno o più componenti RSS, li analizza e restituisce un nuovo componente come risultato. Ciò permette di avere uno strumento di rappresentazione grafica che fornisce agli sviluppatori la possibilità di procedere

manipolando il singolo componente. Yahoo Pipes si concentra in particolare sull'integrazione dei dati piuttosto che sull'integrazione dell'interfaccia utente. Ne mostriamo un'immagine in figura 2.1.

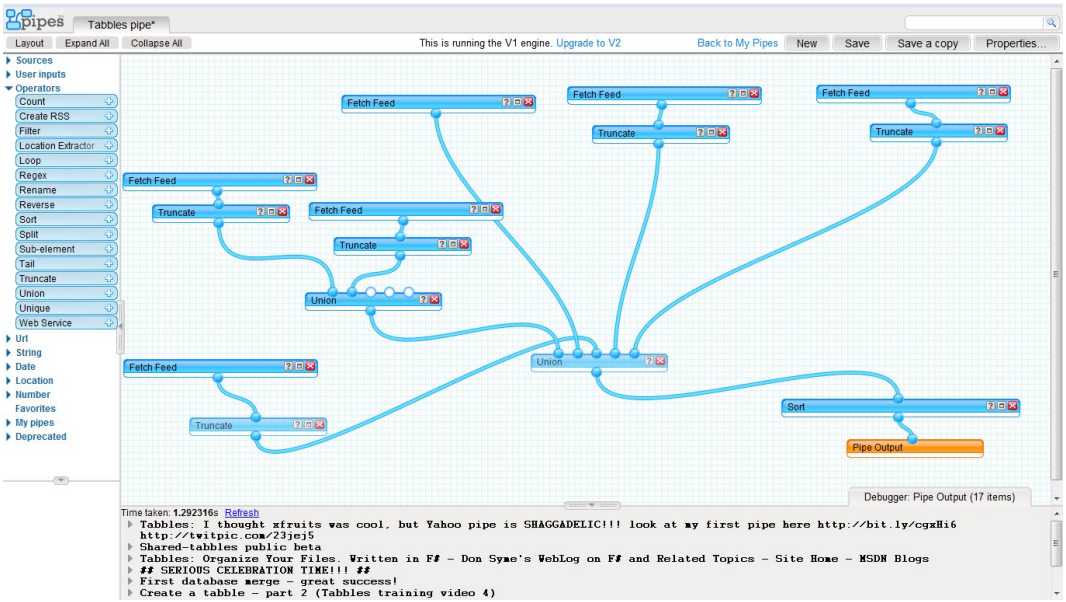

Figura 2.1: Yahoo Pipes

2.5.2 Intel Mash Maker

Intel Mash Maker [24] è un browser per il Web Semantico. Può essere utilizzato come un browser web, eccettuato il fatto che Intel Mash Maker comprende il significato semantico delle pagine web. Questo fatto rende possibile la presenza di alternative, più dispendiose, rappresentazione di dati e anche l'*intelligente* combinazione dei dati con altre informazioni. Con Intel Mash Maker, e similmente con altre applicazioni, è possibile creare Web Mashup, ma Intel Mash Maker (figura 2.2) è provvisto di un tool per la creazione di un web mashup predisposto, basato sull'interpretazione semantica delle pagine web. Risulta inoltre provvisto di strumenti che visualizzano i contenuti semantici. In conclusione, questo framework ha l'obiettivo di ottenere una soluzione che renda disponibile un livello di astrazione che a sua volta permette la componentizzazione di componenti eterogenei tramite il collegamento di eventi di alto livello e di operazioni, in modo tale che l'uten-

te finale possa pensare alla composizione come ad una operazione funzionale ed astratta, senza la necessità di doversi occupare dell'implementazione dei componenti o degli adattatori e della tecnologia di comunicazione che è introdotta da essi. D'altro canto, chi mette a disposizione un componente ha soltanto la necessità di esporre i propri eventi e metodi con un semplice adattatore mappando gli eventi e i metodi stessi con un concetto di livello superiore.

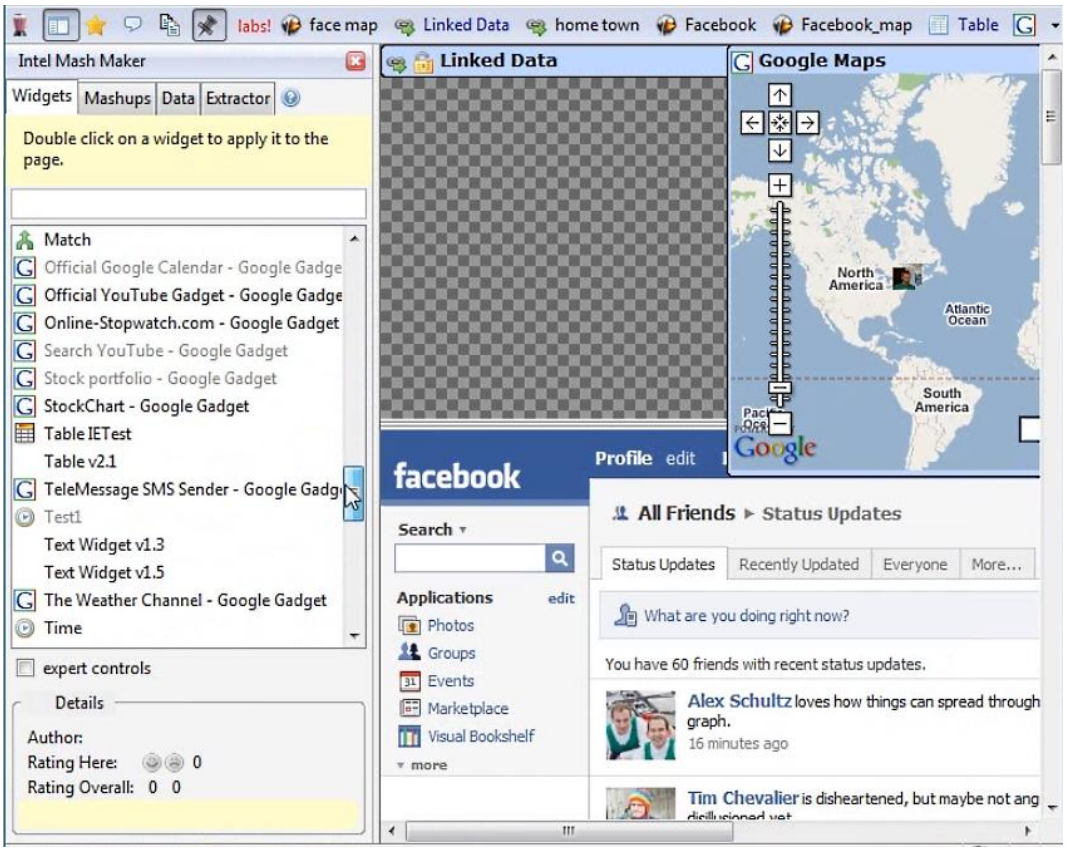

Figura 2.2: Intel Mash Maker

2.5.3 Quick and Easily Done Wiki

QEDWiki [22] è la proposta IBM per un *creatore di mashup* wiki-based, completamente in esecuzione all'interno del browser e con un client di accesso che permette l'interazione con il mashup Hub di IBM. L'Hub supporta la creazione dei data-feed e widget di interfaccia utente e incorpora dati del Mashup per ap-

plicazioni Intranet (Damia) per l'assemblaggio dei dati e la loro manipolazione. Avendo tutte le caratteristiche tipiche di un ambiente wiki, permette agli utenti di modificare, visualizzare immediatamente e condividere facilmente mashup. I mashup sono assemblati da JavaScript da Widget basati su PHP, il cui cablaggio determina il comportamento del mashup. I Widget rappresentano componenti applicativi che potrebbero o meno avere la propria interfaccia utente. Per assemblare un mashup, un utente seleziona un layout di pagina (un modello HTML) e poi trascina degli widget sulla griglia della pagina (figura 2.3) e li configura in modo interattivo.

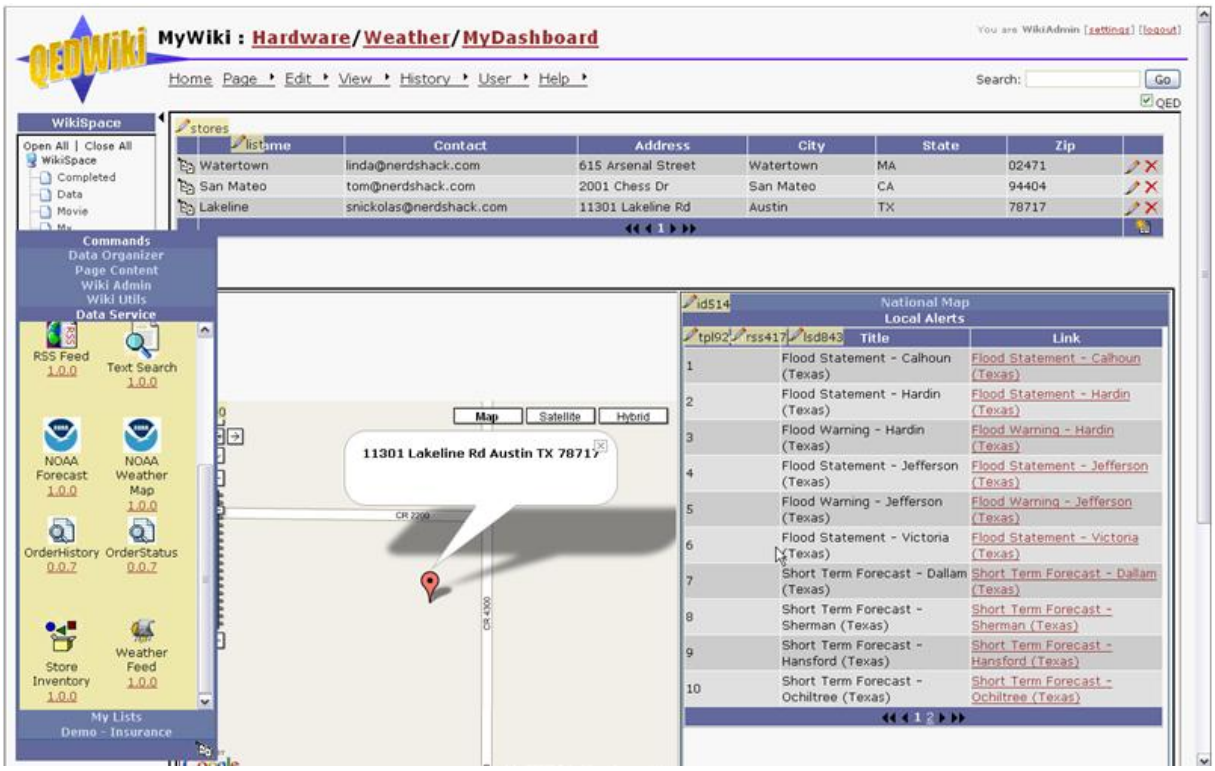

Figura 2.3: Quick and Easily Done Wiki

2.5.4 Damia

Mediante un interfaccia basata sul web, IBM DAMIA [20], rende possibile un agevole utilizzo dei tool che gli sviluppatori e gli utenti IT possono utilizzare per assemblare velocemente i componenti dei dati da Internet e una vasta gamma di

fonti di individuazione dei dati. I benefici di questo servizio includono la capacità di aggregare e di trasformare una consistente varietà di componenti a livello di dati e di contenuto, che possono essere utilizzati nei nuovi mshup. Damia fornisce inoltre la possibilità di svolgere le seguenti attività:

- importare XML, Atom, e feeds RSS;
- assemblare componenti sia da Internet che da fogli di sviluppo Excel (il database costituisce un supporto per questa applicazione);
- importare dati da file locali in formato XML e fogli di sviluppo Excel;
- aggregare e trasformare un'ingente varietà di dati o di contenuti dei componenti in nuovi servizi organizzativi.

Quando si costruisce una applicazione web completa che rende disponibile un'interfaccia utente, sono richiesti ulteriori tool o tecnologie al fine di mostrare il contenuto dei dati messo a disposizone da DAMIA. I costruttori di applicazioni mashup e i lettori dei componenti che si servono di Atom e RSS possono essere utilizzati come il livello di presentazione nella costruzione di applicazioni web. DAMIA presenta la seguente composizione:

- un'applicazione web basata sul browser per l'assemblaggio, la modifica e la presentazione dei mashup;
- servizi per la manipolazione, la custodia e il recupero dei componenti dei dati creati nell'ambito dell'innovazione nello stesso modo che sulla rete Internet. In aggiunta alla creazione dei componenti dei dati a partire da varie sorgenti, Damia può pubblicare informazioni come ad esempio fogli di sviluppo Excel oppure documenti XML nel formato mashup;
- una repository per la condivisione e la raccolta dei componenti o delle informazioni create da DAMIA;
- servizi per l'amministrazione dei componenti e delle informazioni rigurdanti i mashup;
- ricerca delle capacità e tool per la segnalazione e la valutazione dei mashup.

2.5.5 JackBe Presto

JackBe Presto [23] è una piattaforma di Enterprise Mashup che include funzionalità per la creazione e la gestione di mashup aziendali. Esso fornisce un supporto per gli sviluppatori e gli utenti dell'applicazione nella'attività di creazione, dall'inizio fino a personalizzare e condividere mashup Enterprise Apps. Le funzionalità di JackBe Presto sono le seguenti:

- motori di servizi di accesso per i servizi Web, SQL, RSS e Web clipping;
- compositori/creatori di mashup tra cui un grafico, uno strumento drag-and-drop (figura 2.4) e un linguaggio di marcatura per *declarative enterprise mashup*;
- connettori mashup per gli strumenti di enterprise più popolari, fra cui Microsoft Excel, HP Systinet e Oracle WebCenter;
- API mashup per JavaScript, Java, REST, C#, e NET;
- Enterprise Mashup Markup Language (EMML).

Nel marzo 2010 JackBe ha lanciato una versione cloud-based del suo prodotto Presto ospitato su Amazon EC2.

2.5.6 Marmite

Marmite [19] è uno strumento creato da Jeffrey Wong e Jason I Hong. Si presenta come una soluzione che consente agli utenti di creare i propri mashup, senza avere prerequisiti in termini di competenze di programmazione. Permette di:

- accedere a Web Service API;
- combinare Web Service API con screen-scrape orientati alla programmazione;
- presentare uno strumento composto da un menù di operatori, una visualizzazione del flusso di dati e una vista dei dati (figura 2.5).

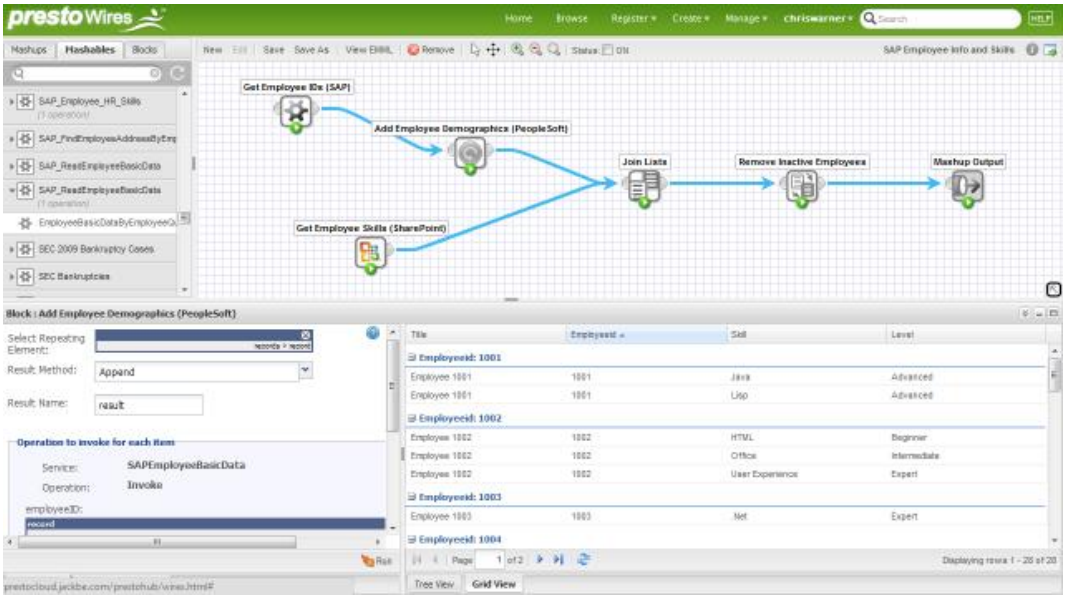

Figura 2.4: JackBe Presto

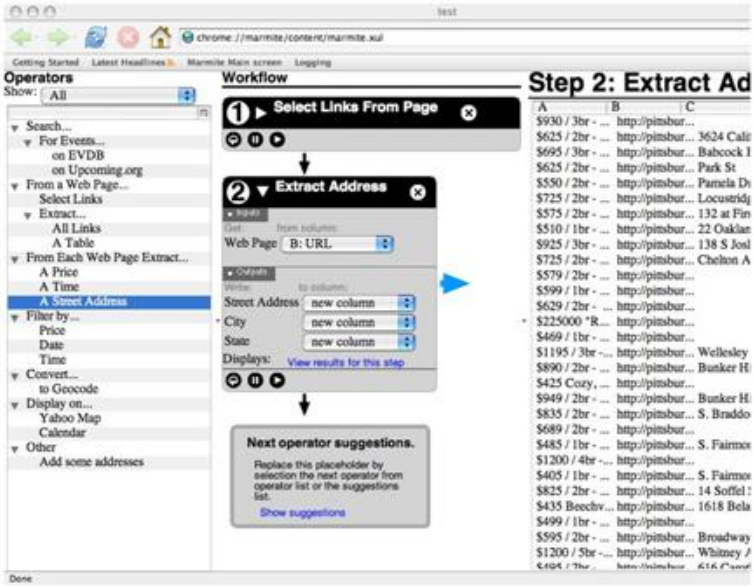

Figura 2.5: Marmite

2.5.7 MashArt

Di seguito è riportata una descrizione generale di MashArt [26, 48]. MashArt (figura 2.6) consiste in uno strumento avanzato che da la possibilità all'utente di componentizzare le proprie applicazioni in maniera relativamente semplice ed efficiente mediante specifici metodi. Innanzitutto vi è un metodo di run-operation invocato dal framework che permette di effettuare un'operazione che è descritta all'interno del mio componente. Il parametro *operation* descrive l'operazione che dovrà essere eseguita. Successivamente ci si servirà di un metodo chiamato *load* il quale andrà a posizionare il componente all'interno del *div* corrispondente nella pagina html. Infine verrà sollevato un evento. Gli eventi generano dei parametri come output, mentre le operazioni successive prendono dei parametri come input. I parametri generati dagli eventi vengono riconosciuti e servono per identificare univocamente il componente specifico che si vuole andare a caricare ed il *div* in cui lo si vuole caricare. Tale è dunque in breve il comportamento di base di mashart, una piattaforma per i mashup e lo sviluppo tramite host di applicazioni web di alto livello e che permette anche l'analisi approfondita di tali applicazioni.

MDL

Per istanziare il modello di componente descritto, utilizziamo MDL, un linguaggio di descrizione astratto e non legato alla tecnologia per i componenti UI (simili ai servizi WSDL per il web). Data un'applicazione che vogliamo componentizzare, MDL consente di definire un nuovo componente per descrivere quali sono gli eventi e le operazioni che caratterizzano il componente e per definire i tipi dei dati e il costruttore. Non c'è un esplicito costrutto di linguaggio per lo stato del componente, come quello che è posto a mano internamente al componente, per esempio il suo UI. D'altra parte, MDL ci consente di descrivere i cambiamenti di stato nella forma di eventi e operazioni.

2.5.8 ServFace

Il progetto ServFace [18] mira a creare una metodologia di reingegnerizzazione dei processi basata sul modello per uno sviluppo integrato di processi per applicazioni basate sui servizi. L'insieme di Service Annotations identificato nel progetto

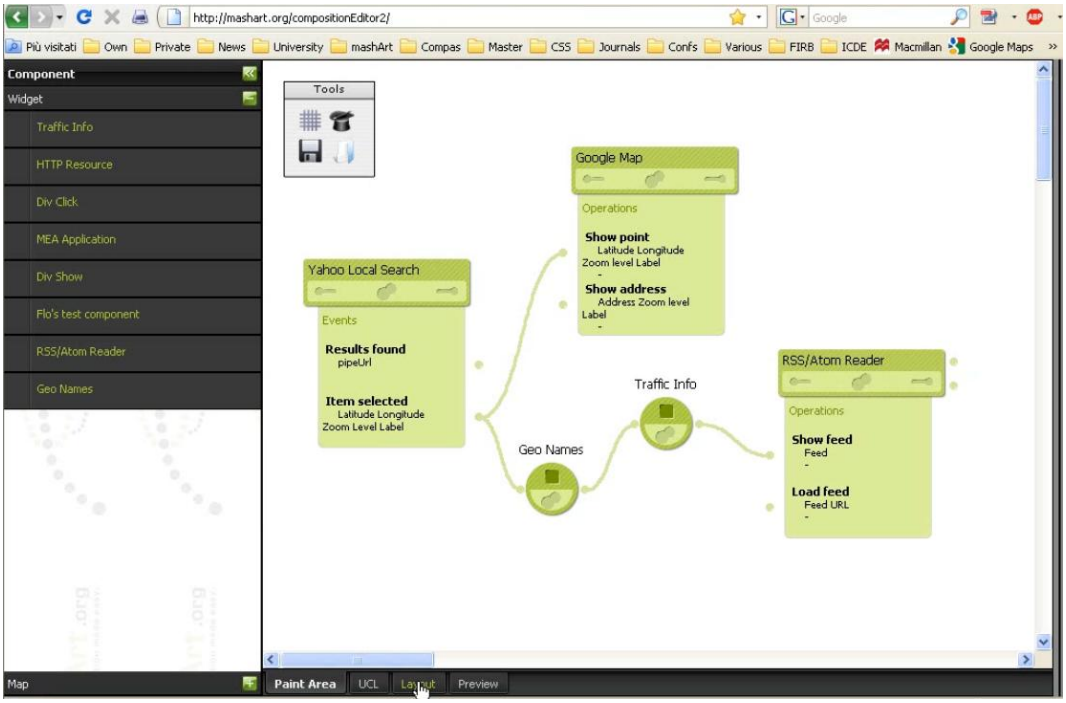

Figura 2.6: MashArt

ServFace sono raccolte nel ServFace Annotation Model. Insieme a servizi di descrizione tecnica quali WSDL, rende disponibili gli input necessari per un meccanismo di inferenza di una interfaccia utente automatizzata, che generi interfacce utente di alta qualità per l'interazione tra utenti umani e Web services annotati. Per la composizione di servizi annotati nella direzione di applicazioni complesse, sono investigati in ServFace due alternativi approcci di modellazione:

- Presentation-oriented service composition, nel quale l'applicazione viene modellata visualmente tramite la composizione delle applicazioni dell'interfaccia utente per parti che sono generate utilizzando le annotazioni dei servizi. Inoltre, un tool ServFace Builder è attualmente in costruzione: dovrà integrare un motore per l'inferenza per generare interfacce utente a partire da servizi annotati. Questo approccio è il solo ad essere più orientato allo sviluppo da parte dell'utente finale;

- Task-oriented service composition: è supportato da un tool chiamato MARIAE [53]. Rende disponibile una nuova soluzione per sfruttare i task di

modellazione (rappresentati in notazione ConcurTaskTrees) e i modelli di interfaccia utente (in linguaggio MARIA) per il design e lo sviluppo di applicazioni interattive basate su servizi web per svariati tipi di piattaforme (desktop, smartphone, vocali, ecc.). Il tool è in grado di importare automaticamente descrizioni di servizi e annotazioni e di supportare l'associazione interattiva di compiti di servizi base con le operazioni di sevizi web. Inoltre un discreto numero di trasformazioni semi-automatiche sfrutta l'informazione in descrizioni di servizi e annotazioni di questo tipo per derivare dei *front-end* utilizzabili su più apparecchi.

2.5.9 SOA4ALL

Il Progetto di Ricerca Europeo SOA4All [30], propone un'innovativa piattaforma di realizzazione dei servizi fruibile dal maggior numero possibile di utenti grazie alla combinazione delle tecnologie Web 2.0, del context management e della semantica in un'unica soluzione completa e user-friendly. SOA4All mira a realizzare un mondo in cui miliardi di parti espongono e consumano servizi attraverso la tecnologia avanzata del web. L'obiettivo principale del progetto è quello di fornire un framework globale e le infrastrutture che integrano i progressi tecnici complementari ed evolutivi in una piattaforma di servizio coerente e indipendente dal dominio di consegna. Al fine di massimizzare l'impatto del progetto sul mondo reale, l'utilità della piattaforma SOA4All sarà dimostrata attraverso la collaborazione con le principali imprese europee. In un contesto più ampio, SOA4All contribuisce in modo significativo a rafforzare la competitività del Software e del settore IT a livello europeo. Il framework SOA4All supporta un mondo in cui vi sono un numero massiccio di componenti e servizi da esporre e utilizzare, obiettivo da realizzare tramite una piattaforma coerente e indipendente dal dominio. L'architettura complessiva di SOA4All può essere strutturata in quattro parti principali:

- SOA4All Studio: Una ricca piattaforma Web che fornisce agli utenti una vista unificata che copre l'intero ciclo di vita dei servizi, tra cui le fasi di progettazione, di esecuzione e analisi post-mortem;

- SOA4All Distributed Service Bus: La spina dorsale infrastrutturale attorno alla quale tutti i componenti SOA4All possono comunicare e collaborare unendo spazi semantici e funzionalità di Enterprise Service Bus;
- SOA4All Services Platform: Il gruppo di servizi che forniscono le funzionalità di base e attività, come la Service ranking e la selezione, il servizio di *discovery*, il Servizio di adeguamento, il Servizio di Composizione, il servizio di esecuzione e la reasoning Engine;
- Business Services: I servizi effettivamente forniti dagli utenti finali.

2.5.10 Caratteristiche dei tool a confronto

In questo breve paragrafo riportiamo una tabella comparativa(composta da tre parti (a), (b), (c)) [34] che raccoglie e paragona tutte le caratteristiche di schematizzazione più generali per ciascuna delle piattaforme descritte.

Le dimensioni per il confronto riguardano:

- *Component Model*: indica su che modello si basa il componente, se sia orientato ad un modello rivolto a logiche di business, oppure se si ponga come scopo finale una modellazione volta all'interfaccia utente;
- *Composition Model*: valuta se il modello di composizione sia interamente lato server o se invece sia determinato da specifici eventi;
- *Development Environment*: discrimina i tool in di sviluppo in base al loro ambiente di sviluppo. Le principali distinzioni sono ambiente di sviluppo basato sul browser oppure basato sul server;
- *Programming Efforts*: permette di categorizzare lo sforzo fatto nella programmazione su tre livelli: basso, medio o alto;
- *Composition Runtime*: valuta se il runtime della composizione sia interamente lato server o se invece sia determinato da specifici eventi;
- *Browser Compatibility*: questo parametro valuta la compatibilità del tool con i browser attualmente più utilizzati;

- *Component Lifecycle Management*: esprime la presenza o meno di una gestione del ciclo di vita di ciascun componente della piattaforma;
- *User-defined Component Support*: indica se nella piattaforma sia presente un qualche tipo di supporto per i componenti definiti dall'utente;
- *Event Bus*: valuta la presenza o l'assenza di un bus sul quale viaggino gli eventi;
- *Service Adaptor*: valuta se vi sia o meno una funzionalità che permetta l'adeguamento dei servizi;
- *Cross-domain Handler*: categorizza alternativamente i tool come server-side-proxy o plug-in a seconda della tipologia di handler cross-domain che essi posseggono;
- *Authentication*: infine questa caratteristica indica la presenza o meno di un qualche tipo di autenticazione nella piattaforma.

Caratteristiche	YahooPipes	IntelMashMaker	QEDWiki
Component Model	business logics	-	business logics
Composition Model	server-side	event-driven	server-side
Development Environment	server-based	browser-based	browser-based
Developer Assistances	low	low	medium
Programming Efforts	low	medium	high
Composition Runtime	server-side	browser-side	browser-side
Browser Compatibility	all	Firefox	All
Component Lifecycle Management	yes	no	-
User-defined Component Support	no	no	yes
Event Bus	no	no	yes
Service Adaptor	yes	no	yes
Cross-domain Handler	server-side-proxy	plug-in	server-side-proxy
Authentication	no	no	no

Tabella 2.2: Tabella comparativa dei tool di sviluppo (a)

Caratteristiche	Damia	JBe Presto	Marmite
Component Model	UI	business-logic	UI
Composition Model	server-side	server-side	event-driven
Development Environment	browser-based	browser-based	browser-based
Developer Assistances	medium	high	medium
Programming Efforts	medium	high	low
Composition Runtime	browser-side	browser-side	-
Browser Compatibility	all	all	-
Component Lifecycle Management	yes	no	yes
User-defined Component Support	yes	no	yes
Event Bus	-	no	-
Service Adaptor	yes	no	no
Cross-domain Handler	server-side-proxy	-	server-side-proxy
Authentication	no	no	no

Tabella 2.3: Tabella comparativa dei tool di sviluppo (b)

Caratteristiche	MashArt	ServFace	SOA4All
Component Model	UI	UI	UI
Composition Model	event-driven	event-driven	-
Development Environment	browser-based	browser-based	browser-based
Developer Assistances	medium	medium	high
Programming Efforts	medium	medium	medium
Composition Runtime	browser-side	browser-side	-
Browser Compatibility	Firefox	all	all
Component Lifecycle Management	yes	no	yes
User-defined Component Support	yes	no	yes
Event Bus	yes	no	yes
Service Adaptor	yes	no	-
Cross-domain Handler	server-side-proxy	plug-in	-
Authentication	no	no	no

Tabella 2.4: Tabella comparativa dei tool di sviluppo (c)

2.6 Annotazioni semantiche

Le annotazioni semantiche [33] sono informazioni semantiche associate a risorse web. Si utilizzano per arricchire il contenuto informativo dei documenti e per esprimere in maniera formale, il significato di una porzione di testo in un documento, di un Web service o di strutture dati coinvolte in un processo di collaborazione (interoperabilità tra applicazioni software). Vi sono dei criteri specifici per classificare le annotazioni; ne elenchiamo di seguito i principali:

- livello di formalità del linguaggio usato: formale, espressa in un linguaggio formale di rappresentazione della conoscenza e che permette di processare i dati senza l'intervento umano (machine understandable) oppure informale, espressa in linguaggio naturale o in un linguaggio controllato e che permette di aggiungere informazioni su documenti o risorse, fruibili da un utente umano;

- posizionamento dell'annotazione: embedded, cioè inserita all'interno dello stesso documento annotato oppure attached , cioè memorizzata separatamente, collegata al documento da un link;

- destinatario dell'annotazione: nel caso di human user ha lo scopo di fornire informazioni aggiuntive su una risorsa e in questo caso l'annotazione in genere sarà poco formale probabilmente realizzata in linguaggio naturale, mentre nel caso di computer ha lo scopo di esplicitare il significato di una risorsa e in tal caso l'annotazione deve essere di tipo formale per poi essere processabile da una macchina;

- tipo di risorsa annotata:

  - annotazione di documenti: frammento di testo, pagina HTM, immagini, ecc.;
  - annotazione di Web Services [29]: esprimere cosa fa un servizio, descrivere i parametri di input e l'output, solitamente è di tipo formale;
  - annotazione di strutture dati o processi, coinvolti nello scambio di informazioni tra applicazioni software che cooperano. Solitamente è di tipo formale, ontology-based;

- livello di restrizione imposto sul linguaggio:
  - absence: i termini del linguaggio possono essere usati senza vincoli;
  - advised (consigliato): si è liberi di scegliere se utilizzare o meno un glossario, un ontologia, (o anche parole chiavi) in alternativa al linguaggio naturale;
  - mandatory (vincolante): c'è l'obbligo di usare un glossario di riferimento o un ontologia (annotazione ontology-based).
- modalità di annotazione [1]:
  - annotazione attraverso istanze di concetti: l'annotazione consiste in associazione di una istanza all'elemento annotato, valorizzazione delle proprietà che descrivono l'istanza;
  - annotazione attraverso concetti: l'annotazione consiste in associazione di un concetto dell'ontologia all'elemento annotato e associazione di una composizione di concetti dell'ontologia (attraverso opportuni operatori) all'elemento annotato.

Esistono anche svariati tool di annotazione, molti dei quali presentano i più disparati problemi, come ad esempio Cohse e Melita.

2.6.1 Annotazioni di compatibilità e similarità

Il servizio mashup coinvolge diversi aspetti, estendendosi da servizio di composizione a regole per l'annotazione semantica del servizio e del mashup dei componenti e alla mediazione di dati permettendo facili combinazioni di servizi nella applicazione finale del mashup.

Esistono difficoltà nel campo semantico dei mashup, che sottolineano l'importanza delle piattaforme e degli accostamenti, che rendono meno oneroso il carico della composizione di mashup, che richiede capacità di programmazione, conoscenze approfondite dei formati di input e output per l'implementazione manuale della mediazione dei dati nella composizione finale. La composizione manuale di pesanti

[1] questa classificazione riguarda solo le SA Ontology-based.

standard si contrappone ai linguaggi più semplici nelle soluzioni di tipo RESTful. Ciononostante, costruendo applicazioni mashup, incominciando da un'abbondanza di diversi servizi disponibili nel Web, c'è bisogno anche di un supporto per trovare il giusto servizio prima di comporli. La migliore delle nostre conoscenze è un'estensiva spiegazione di come le tecniche avanzate per la scoperta del servizio semantico, che non è ancora stato scoperto, possano aiutare il disegnatore di mashup.

In una piattaforma online di mashup che rende disponibili la composizione, il riuso, la divisione e la annotazione di Web APIs, è presentato anche il servizio dei mashup [42]. Le diverse caratteristiche della piattaforma risiedono nel linguaggio dello specifico dominio, che è introdotto per rappresentare esplicitamente le attività che un disegnatore di mashup deve compiere, così come la mediazione dei dati e la mediazione del protocollo di servizio.

I principali lavori descritti in [42], si concentrano sugli algoritmi e sulle soluzioni di ottimizzazione per un efficiente servizio di composizione mashup. Viene proposto un servizio visivo del linguaggio mashup, per comporre ed eseguire le queries nell'ambito dei servizi di ricerca. I sevizi di ricerca sono definiti come sevizi che rendono disponibili dei dati classificati. La composizione del servizio è costruita come un grafo aciclico direzionato, i cui nodi sono invocazioni del servizio e i cui archi sono connessioni tra i servizi. Il lavoro si focalizza sulla definizione di un piano di accesso fisico al servizio per l'esecuzione della composizione del servizio Web. Viene proposto anche un algoritmo di evoluzione, che automaticamente compone differenti servizi di informazione Web basata su descrizioni del servizio semantico. L'informazione che può essere recuperata dal servizio di invocazione Web è automaticamente trasformata in una rappresentazione semantica e presentata come un mashup per gli utenti del sistema. La retrocessione di quella soluzione è l'involuzione del disegnatore di mashup nel processo di composizione, che è essenzialmente automatico e non supporta la riparazione del servizio e la fase di selezione, non assicurando così il migliore adempimento delle intenzioni del disegnatore. Vi sono inoltre un agile modello sia per le applicazioni Web guidate dai dati, che è studiato per la composizione Web del workflow, e un linguaggio per questa specifica, chiamata BITE. BITE [42] combina le principali composizioni del processo SOA con le richieste architetturali di tipo REST e le funzionalità

del workflow. Un approccio guidato dall'ontologia per la composizione del servizio, che è relativo al nostro studio, è stato proposto in un diverso scenario dell'applicazione, dove il disegnatore dei processi del Web è supportato nella composizione semiautomatica dei Servizi Web attraverso il controllo delle similarità semantiche tra interfacce e sevizi individuali.

In particolare, si distingue tra la composizione automatica della corrispondenza delle interfacce, dove le diverse composizioni possibili sono classificate e ne è selezionata una ottimale, e la composizione seguita dall'uomo, dove il disegnatore seleziona un servizio da una lista ordinata e costruisce incrementalmente il processo del Web. Con rispetto per il nostro lavoro, una strategia di composizione a livelli (ad esempio top-down), così come lo sfruttamento di un'ontologia specifica per la selezione veloce di servizi utilizzabili in termini di componenti, è una soluzione ottimale, dal momento che una simile soluzione potrebbe aiutare il disegnatore in una composizione interattiva. Inoltre, in generale, non sono dati dettagli riguardo le tecniche di similarità e non sono proposte metriche basate sulla semantica a supporto del disegnatore all'interno del processo di composizione. È consigliabile optare per un approccio guidato dal modello al fine di conseguire un modello collaborativo del servizio mashup. Sfruttando unitamente le tecniche di annotazione sintattica e semantica si può realizzare una metodologia di composizione che sia rivolta all'utente finale e quindi altamente fruibile da esso.

2.7 Generazione di reccomendations per la composizione dei mashup

In questa sezione verranno presentati due lavori comparabili con quello svolto in questa tesi. Essi si focalizzano sulla generazione di raccomandazioni a supporto dello sviluppo di mashup, ma non sono orientati alla qualità, uno degli argomenti chiave di questo lavoro di tesi. Come già accennato nell'introduzione, il nostro lavoro è infatti tra i primi ad affrontare il problema della qualità per la composizione di mashup. Tuttavia, i due lavori qui descritti meritano attenzione poiché, anche se da una diversa angolatura, affrontano il problema di facilitare la scelta di componenti e di pattern di composizione per la creazione di mashup. In questo

senso, essi possono perciò essere confrontati con il lavoro di questa tesi.

2.7.1 Mashup Advisor

Un tentativo nel creare un ambiente di sviluppo di mashup che faccia delle recommendation è Mashup Advisor [14]. Esso si rivolge ad un pubblico che non deve necessariamente fare parte della categoria dei programmatori e cerca di rendere più rapida ed efficace la composizione di servizi in un mashup. Tutto ciò viene reso possibile da un sistema di raccomandazioni generate automaticamente sulla base di considerazioni semantiche e statistiche sull'uso di componenti in composizioni già prodotte in passato dagli stessi o da altri utenti. Mashup Advisor fornisce delle raccomandazioni a fronte di un cambio di stato durante la composizione del mashup o a fronte di una richiesta esplicita da parte dell'utente. Il processo di raccomandazione è formato da due fasi: nella prima, il tool genera una lista ordinata di componenti in base ad una valutazione, mentre nella seconda viene calcolato il piano migliore per la scelta del prossimo componente in base al mashup parziale già presente. L'architettura di questo tool si compone di quattro componenti: un repository manager, un semantic-matcher, un output ranker e un planner. Il primo analizza la repository e calcola informazioni statistiche sull'uso di componenti in altre composizioni. Il gestore della statistica calcola le statistiche d'uso dei concetti all'interno di Mashup Advisor. Per il calcolo probabilistico, il semantic-matcher può essere configurato in modo da tenere in considerazione quanto due concetti siano semanticamente simili. Il semantic matcher è usato sia dal planner che dall'ouput ranker e serve a trovare il migliore matching per un dato concetto, calcolando un *punteggio di similarità*. L'obiettivo dell'output ranker è triplice: in primo luogo, questo modulo identifica un insieme di candidati che hanno degli output coerenti con il mashup parziale che l'utente sta componendo e che quindi possono essere aggiunti; in secondando luogo, assegna un punteggio di rilevanza ad ogni candidato; infine, classifica i candidati in base al loro punteggio. L'output ranker seleziona i componenti dal repository manager, escludendo i concetti che appaiono nel mashup parziale dell'utente, sia in forma di output che in forma di input. Per ogni candidato di output, il punteggio è stimato con la probabilità condizionata che quell'output possa entrare a fare parte del mashup, sapendo co-

me è strutturato il mashup parziale del momento. Una volta che l'output ranker raccomanda un numero di concetti di output rilevanti, l'utente può selezionare qualsiasi concetto; Mashup Advisor calcolerà quindi la miglior combinazione per il concetto selezionato. Il tool usa un pianificatore per calcolare il più alto valore di utilità per raggiungere uno scopo (il componente selezionato), a partire dallo stato del mondo (i concetti del mashup parziale).

2.7.2 MatchUp: Autocompletion for Mashups

Il sistema MatchUp [5] si propone di supportare uno sviluppo dei mashup che sia rapido, a richiesta e intuitivo; ciò grazie ad un innovativo meccanismo di autocompletamento. L'intuizione che ha guidato lo sviluppo di MatchUp è che i mashup, anche se sviluppati da diversi utenti, hanno delle caratteristiche in comune: usano delle classi di componenti simili, come è simile il modo in cui vengono legati gli stessi componenti. MatchUp analizza questa similarità per predire, dato un mashup parzialmente composto, quale saranno i più probabili, e potenziali completamenti (includendo componenti e connessioni non ancora inseriti) per quel particolare mashup parziale. Questo sistema è realizzato grazie ad un nuovo algoritmo di ranking, il cui risultato finale è una lista dei *top-k* completamenti da cui gli utenti possono scegliere in base alle proprie necessità.

Capitolo 3

DashMash: verso la composizione dei mashup orientata agli utenti finali

L'approccio descritto per la piattaforma DashMash cerca di rispondere alle necessità spiegate nell'introduzione, incoraggiando la composizione di mashup aziendali in varie direzioni. La più importante è il paradigma di composizione intuitiva che nasconde all'utente la complessità della composizione dei servizi. Questo è possibile grazie al meccanismo per la definizione automatica e basata sul sistema di coppie di servizi per flusso di dati e il controllo di sincronizzazione dei servizi, mentre permette all'utente di customizzare le coppie per aggiungere maggior interattività al mashup composto. Queste funzionalità sono state implementate fornendo modelli e astrazioni che verranno descritti nei paragrafi seguenti. Questi aspetti e le corrispondenti soluzioni saranno descritte nel resto del capitolo.

3.1 La piattaforma DashMash

La piattaforma DashMash [13, 7, 6], costituisce un esempio di piattaforma mashup end-user-oriented, che si pone l'obiettivo di colmare le lacune che in genere impediscono agli utenti finali di sfruttare appieno il potenziale del mashup a livello di strumenti di innovazione. DashMash offre un paradigma di composizione intelligibile e facile da usare, che permette anche agli utenti inesperti di comporre un

mashup proprio. Il suo paradigma è efficace e aumenta la soddisfazione degli utenti finali. Su questa piattaforma si basano gli algoritmi che stanno alla base della presente trattazione, in quanto detti algoritmi cooperano nel perseguire l'obiettivo di fornire possibilità sempre maggiori per gli utenti. DashMash è una piattaforma di mashup Web che consente la creazione di cruscotti per l'accesso e l'analisi di informazioni. In contrasto con l'approccio tradizionale alla costruzione di cruscotti, basato su motori analitici rigidi e monolitici, DashMash si propone di offrire un'insieme di servizi (component services) che l'utente può opportunamente comporre in base a specifiche necessità. DashMash permette all'utente di comporre servizi senza che egli debba necessariamente padroneggiare aspetti tecnici, come ad esempio il passaggio di messaggi e dati che molti utenti non conoscono.

In DashMash la composizione è applicata a servizi adeguatamente registrati e descritti che chiamiamo componenti. I componenti di DashMash possono essere specifici per un dominio o generici. Il primo tipo consiste in servizi creati adhoc per specifici domini per i quali deve esser costruito il cruscotto, per esempio per fornire accesso a sorgenti di dati interne all'organizzazione. Generalmente i componenti di dominio specifici sono costruiti internamente (ad esempio dal dipartimento IT aziendale), mentre i componenti generici corrispondono a servizi pubblici (ad esempio mappe e RSS Feeds). Data la natura *aperta* di DashMash (figura 3.1) (e dei mashup in genere) la natura interna o esterna del componente non costituisce un fattore discriminante.

Dato un repository di servizi pronti all'uso (adattati e descritti), DashMash fornisce un ambiente di tipo *sandbox* dove i servizi possono essere combinati in base a determinate logiche di integrazione. Se da un lato può limitare ciò che l'utente può fare, d'altro canto permette un processo leggero di composizione, rende possibile la generazione automatica dei modelli di composizione e incrementa il grado di controllo sulla qualità del mashup finale. Gli utenti sono liberi i comporre vari tipi di applicazioni soddisfacendo diverse necessità.

La caratteristica principale che distingue la piattaforma DashMash dagli altri ambienti per i mashup è il meccanismo di composizione intuitiva. Come mostreremo in dettaglio nei paragrafi a venire, DashMash è equipaggiato da un ambiente visuale che permette all'utente di combinare i servizi trascinando le icone dei componenti in un riquadro di composizione. La piattaforma è in grado di genera-

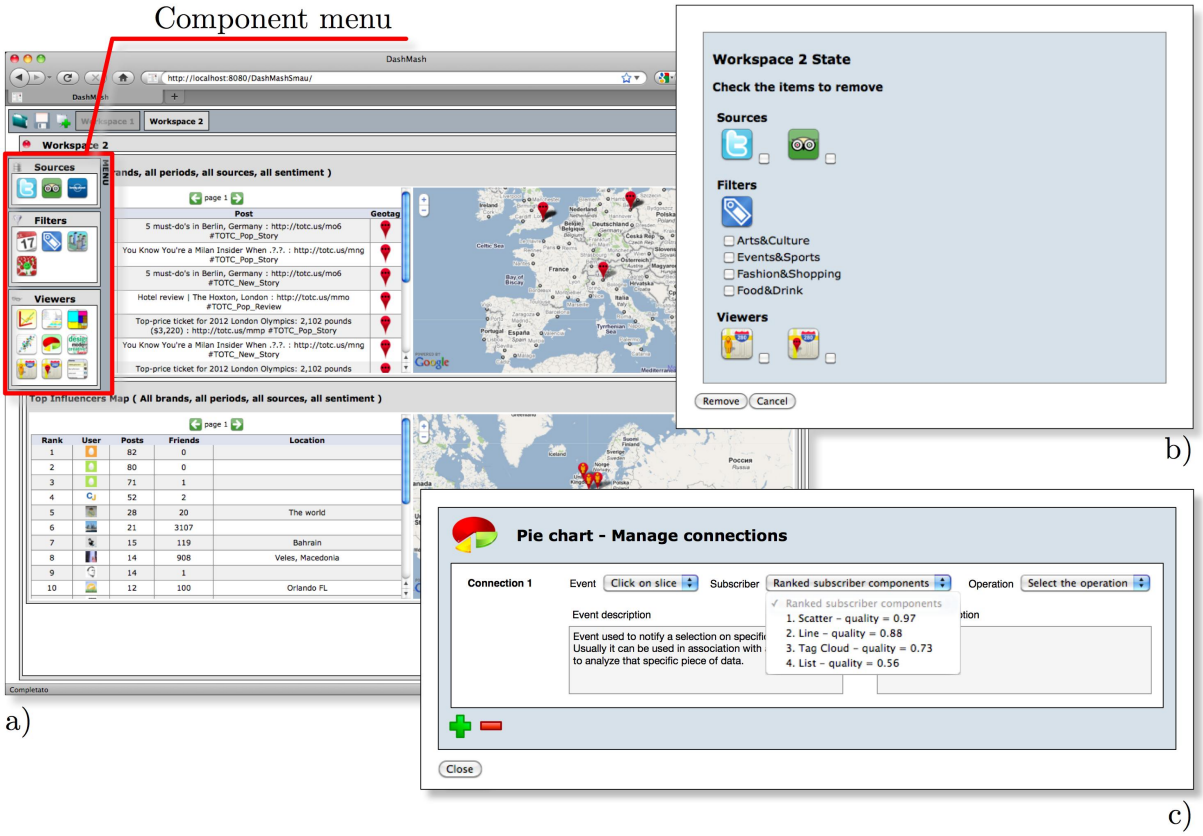

Figura 3.1: La piattaforma DashMash: a)menù dei componenti, b) stato dello workspace, c) gestore delle connessioni

re automaticamente coppie di servizi basati sulla classificazione dei componenti e delle possibili coppie parametri-operazioni che possono essere definite tra essi. Oltre a queste coppie di default la piattaforma supporta la definizione di altre coppie per soddisfare necessità di composizione inaspettate. La loro definizione comunque è anche basata su rappresentazioni intuitive degli elementi coinvolti (chiamati parametri di eventi e operazioni di servizio). L'organizzazione della piattaforma è centrata su un paradigma per l'orchestrazione di componenti che è gestita da un framework intermedio incaricato di gestire sia la definizione della composizione di mashup che l'esecuzione della composizione stessa. A differenza di altre piattaforme dove il progetto del mashup è tenuto separato dall'esecuzione del mashup, in DashMash le due fasi e il rispettivo ambiente sono strettamente interconnessi. L'effetto risultante è che le azioni di composizione dell'utente sono intercettate e automaticamente tradotte in un modello di composizione; questo modello e ogni aggiornamento su di esso vengono eseguiti immediatamente. Gli utenti sono perciò in grado di definire e provare la loro composizione in maniera interattiva e ripetitiva.

3.2 Il modello dei componenti e della composizione

Per raggiungere gli obbiettivi descritti, in primo luogo DashMash trae vantaggio dai modelli utilizzati per la descrizione sia dei componenti che della composizione. Sfruttiamo l'astrazione definita nel progetto Mixup [3], un motore mashup web la cui composizione logica è basata su un modello a eventi: eventi generati dall'interazione dell'utente con un componente (ad esempio la selezione di una parola su un grafico che rappresenta una nuvola piena di parole) possono essere mappati ad operazioni di uno o più componenti sottoscritti a tali eventi (ad esempio il filtraggio di informazioni sulla base della parola selezionata): l'occorrenza dell'evento perciò causa un cambiamento di stato nei componenti sottoscritti. Il mappaggio tra i componenti viene espresso attraverso l'utilizzo di listener di eventi, ingredienti base per la logica di composizione. I listener di eventi specificano l'evento generato da un particolare componente (componenti *Pubblicatori*) e le

operazioni che il componente ricevente (componente *Sottoscrittore*) deve eseguire.

La composizione logica descritta sopra (figura 3.2) richiede che ogni componente sia descritto attraverso un preciso modello. Attraverso l'indicazione del *binding* con gli attuali servizi/API, il modello del componente specifica gli eventi che un componente può generare e quindi comunica al mondo esterno i cambiamenti dello stato del componente. Data la descrizione astratta di un componente, un modello di composizione specifica quali componenti sono correlati in una composizione di mashup e il modo in cui vengono accoppiati attraverso i listener che mappano gli eventi alle operazioni. I modelli di componenti e di composizione vengono memorizzati in un registro di componenti della piattaforma come descrittori XML. In seguito mostreremo come i due livelli di descrizione forniscono un modello uniforme per la combinazione di servizi e anche permettono la programmazione ad alto livello e la generazione di codice. Un'applicazione composita perciò consiste in uno o più componenti, adeguatamente descritti e un modello di composizione che rappresenta i *binding* definiti. Il framework di esecuzione inoltre offre supporto per:

- la traduzione delle azioni di composizione dell'utente in un modello di composizione;
- la sincronizzazione dei componenti del mashup risultante, come specificato nel modello di composizione creato;
- la gestione del layout dell'applicazione composita attraverso la generazione di template HTML che includono indicatori per agganciare ed eseguire i componenti.

3.2.1 UISDL

In questo paragrafo viene descritta la nuova sintassi con cui si presentano i componenti che popolano la piattaforma *DashMash*, che discende concettualmente dalla sintassi MDL, ma che risulta innovativa per caratteristiche che permettono maggiori performance e danno la possibilità di avere modularità. Le parti salienti del file UISDL sono la descrizione del componente e le operazioni che permettono al

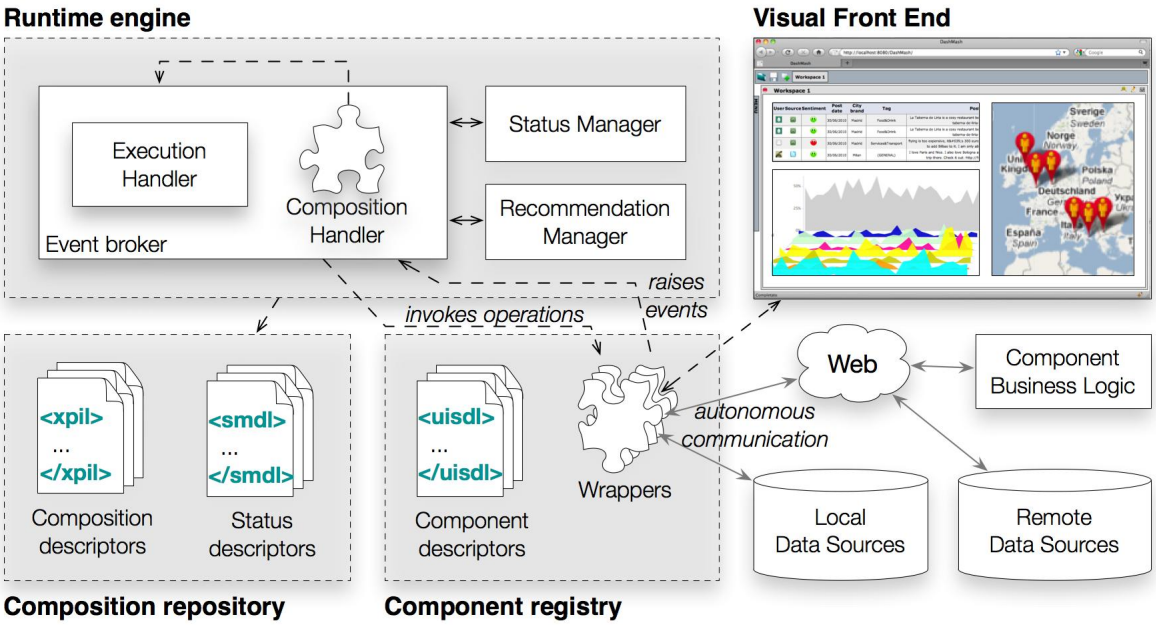

Figura 3.2: Il framework di composizione

framework di interagire con il componente. In breve non si deve fare altro che avere un componente *taggato* in base alle operazioni e agli eventi che esso comprende. A loro volta sia le operazioni che gli eventi possono avere degli output, mentre solo le operazioni possiedono degli input; per ovviare a questa discrepanza si è scelto di assimilare input e output alla più semplice definizione di parametri. Alla luce di questa breve delucidazione non si può fornire migliore chiarimento che un esempio visivo di come sia strutturato un componente, come mostrato di seguito.

```
<?xml version="1.0" encoding="UTF-8"?>

<uisdl version="0.1">
    <component id="componente_di_prova" name="componente_di_prova" description="
        Composition Handler is very useful component that..." adapter="" address=
        "">

        <operation name="getData" description="this operation get data from.."
            dialog-descripion="get data from..." address="getData">

            <param name="param1" description="parametro 1" direction="input" type
                ="xsd:string" validation-regex="*" validation-minvalue="0"
                validation-maxvalue="5"/>
```

```
            <param name="param2" description="parametro 2" direction="output"
                type="xsd:string"/>
        </operation>

        <operation name="getData2" description="this operation get data from.."
            address="getData">

            <param name="param1" description="parametro 1" direction="input" type
                ="xsd:string" validation-regex="*" validation-minvalue="0"
                validation-maxvalue="5" />
            <param name="param2" description="parametro 2" direction="output"
                type="xsd:string"/>
        </operation>

        <event name="dateUpdated" address="dateUpdated" dialogdescripion="on date
             updated...">

            <param name="response" direction="output" type="xsd:string"/>
        </event>

        <event name="dateRemoved" address="dateUpdated">

            <param name="response" direction="output" type="xsd:string"/>
        </event>
    </component>
</uisdl>
```

3.2.2 XPIL, eXtensible Presentation Integration Language

Gli elementi di composizione descritti possono essere specificati in un determinato linguaggio di composizione, cioè eXtensible Presentation Integration Language (XPIL). Così il linguaggio contiene due insiemi di elementi XML: il primo espone i componenti in uso, e il secondo descrive i modelli di composizione. Il container Xpil è l'elemento originario del documento XPIL. Questo ha diversi attributi:

- Language namespace: http://www.openxup.org/2006/08/xpil/integration
- Xmins:tns: questo attributo contiene il riferimento namespace del componente. Il valore è un URL che può inequivocabilmente identificarlo.

Components: <component>
Questo elemento punta a un componente di rappresentazione del livello ed è definito da questi attributi:

- Id: questo elemento specifica un unico id per il componente nel documento XPIL;

- Ref: questo elemento specifica la posizione di un documento XPIL o UISDL. Il valore può essere un path, una URL, e questo può essere un riferimento relativo o assoluto. Se il valore è riferito per un documento XPIL, il componente è un componente made-up, ma questo sarà trattato come un singolo componente nella composizione. L'elemento componente può contenere una lista di proprietà.

Di seguito mostriamo un esempio di componenti:

```
...
<component ref="../components/compositionHandler/compositionHandler.uisdl"
id="compositionHandler" address="compositionHandler"/>
<component ref="../components/dataService/dataService.uisdl"
id="dataService" address="dataService"/>
<component ref="../components/saveService/saveService.uisdl"
id="saveService" address="saveService"/>
...
```

Event listeners: <listener>
Questo elemento specifica un event listener che collega un evento del componente ad un'operazione o ad un altro componente ed è caratterizzato da questi attributi:

- Id: questo elemento specifica un unico id per l'ascoltatore in un documento XPIL;

- Publisher: questo elemento contiene l'id del componente della risorsa che vara gli eventi;

- Event: questo attributo specifica il nome dell'evento. Il valore si riferisce al nome o attributo del tag di evento nel descrittore del componente. La combinazione tra questo valore e l'attributo publisher identifica l'evento.

- Subscriber: questo attributo contiene l'id del componente che deve far funzionare un'operazione perché l'evento sia lanciato;

- Operation: questo attributo specifica il nome dell'operazione che dovrà essere effettuata dopo la ricezione della notifica di un evento. L'elemento può

contenere XSLT/XQuery per la trasformazione del dati e per la loro conversione. Inoltre, questo può contenere script o riferimenti a codici esterni per aggiungere una logica di integrazione addizionale.

Di seguito mostriamo un breve esempio di event listener:

```
...
<listener id="7" publisher="dataService"
event="dataReady"
subscriber="viewerBox1_viewerPieChartHC1"
operation="getData"/>
<listener id="8" publisher="viewerBox1_viewerPieChartHC1"
event="changeParameters"
subscriber="compositionHandler"
operation="changeViewerParameters"/>
...
```

XPIL rende dunque disponibili metodologie di supporto e rappresentazioni dello schema di composizione mostrando quali elementi sono coinvolti nella composizione e come essi interagiscono (attraverso coppie <event- publisher, operation-subscriber>). Inoltre, XPIL rende possibile la descrizione formale e parametrica degli eventi.

3.3 Composizione ed esecuzione del Mashup

Il framework DahMash è un modulo client scritto in Javascript. L'event broker intercetta gli eventi e li distribuisce ai moduli incaricati di gestirli. Gli eventi possono essere relativi alla definizione dinamica della composizione (ad esempio il trascinamento dell'icona di un componente in un'area di composizione) o alle azioni degli utenti e del sistema verificatesi durante l'esecuzione del mashup (questi eventi possono causare modifiche dello stato di alcuni componenti). L'utente interagisce con l'ambiente visuale del DashMash (figura 3.3), in quanto può selezionare i componenti presenti nella barra degli strumenti e aggiungerli alla composizione di mashup. L'aggiunta di un componente si ottiene spostando l'icona in un pannello di composizione chiamato viewer box, che è un contenitore implementato con un HTML che aiuta a contestualizzare l'effetto delle azioni di composizione su alcune sorgenti dati.

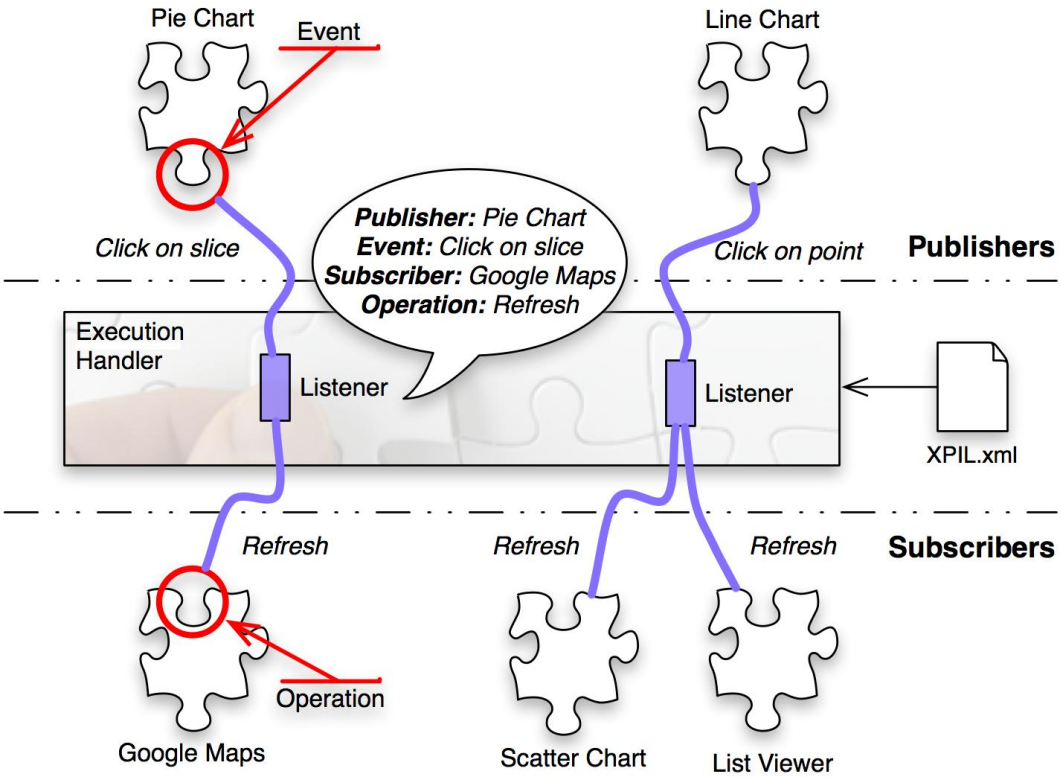

Figura 3.3: Event Bus

Il composition handler gestisce gli eventi di composizione. Trasforma gli eventi in listener e crea o aggiorna (se già esistente) il modello di composizione corrente, aggiungendo puntatori ai nuovi componenti e i listener necessari per collegarli ai componenti già esistenti. Nei paragrafi successivi verrà meglio approfondito il meccanismo spiegato sopra, che comunque risulta del tutto automatico. Il composition handler inoltre invia gli eventi di composizione allo status manager, che ha il compito di mantenere alcune proprietà che possono essere utili per recuperare lo stato di un mashup definito in precedenza, ed eseguirlo più tardi. Queste proprietà dipendono dallo specifico componente, ad esempio i parametri di configurazione che possono essere applicati. Si possono riferire ad esempio a valori di default oppure a valori specifici (il valore dei parametri per un'interrogazione sulle sorgenti dati) a proprietà di layout (ad esempio il colore usato per mostrare i valori di un grafico) o a qualsiasi altra proprietà che l'utente può o vuole settare per controllare lo stato del componente, il contenuto o l'aspetto. Quando l'aggiornamento della composizione e dello stato è terminata, l'execution handler ricarica la composizione e visualizza il mashup. Durante l'esecuzione del mashup i listener vengono attivati dagli eventi corrispondenti. Ogni componente pubblicatore (*Publisher*) notifica all'execution handler il verificarsi di un dato evento. In base ai listener specificati nel modello di composizione l'execution handler a turno informa i componenti sottoscritti, e questo scatena l'esecuzione delle operazioni corrispondenti. La rappresentazione del mashup è ottenuta dall'integrazione di ogni componente dell'interfaccia utente in base a dei template di layout predefiniti, dove ogni componente è eseguito all'interno di un HTML. Gli utenti possono immediatamente vedere il risultato della composizione poiché il cruscotto cambia il proprio stato non appena un nuovo componente viene aggiunto oppure appena viene applicata una nuova configurazione.

Vale la pena notare che il composition handler è anch'esso un componente del mashup: ad ogni azione di composizione dell'utente si genera un evento del composition handler che viene notificato e poi gestito in fase di esecuzione. Un effetto interessante di questa scelta architetturale è che, la gestione degli eventi, qualsiasi sia la natura degli eventi è incapsulata all'interno di un singolo modulo (l'execution handler). Un aspetto ancora più importante e totalmente in linea con la filosofia dei mashup è che la logica della composizione automatica è programma-

bile e perciò flessibile: certamente dipende dall'insieme dei listener predefiniti, la cui conoscenza è all'interno del composition handler, ma essendo un componente del mashup, può essere facilmente tolta e sostituita.

3.4 Generazione del modello di composizione

Per consentire la definizione automatica del *binding* dei componenti, così che all'utente non venga richiesto di programmare esplicitamente i listener, iniziamo con una classificazione dei componenti. E' possibile identificare le seguenti tre classi si componenti:

- i data services sono in grado di recuperare dati dalle sorgenti dati disponibili. Vengono utilizzati principalmente per accedere a database/datawarehouse interni che possono memorizzare dati eterogenei, per esempio contenuti estratti da siti Web che pubblicano informazioni pertinenti, persino commenti generati dagli utenti (ad esempio blog e forum). Un singolo data service è necessario per accedere ai data source interni; quando è richiesta l'integrazione di sorgenti multiple, si può sfruttare un layer di integrazione lato server;

- i filtri aggiungono condizioni di selezione sulle sorgenti dati e consentono un raffinamento interattivo sull'insieme di dati che devono essere analizzati. Eseguono operazioni di filtraggio per ripulire i contenuti delle sorgenti Web sulla base di criteri selezionati (ad esempio le parole che interessano oppure contenuti più recenti).

- i viewers supportano la visualizzazione di dati che possono essere estratti attraverso i data services da sorgenti interne e da generiche sorgenti esterne accessibili attraverso componenti generici. Poiché la visualizzazione dei dati implica alcune forme di aggregazione, i viewers possono anche fornire logiche di trasformazione. Perciò i viewers possono essere semplici visualizzatori di tabelle o di grafici di ogni tipo (ad esempio quelli forniti attraverso le Google Chart API) o un qualsiasi servizio di visualizzazione/aggregazione che possa essere utile per un dominio specifico (ad esempio nuvole di parole per l'analisi del sentiment o mappe per la localizzazione di punti di interesse);

- componenti generici che possono essere integrati per rendere più efficienti i componenti di analisi. Data la natura *aperta* dei mashup, possono riferirsi a differenti funzionalità, ad esempio le mappe, i calendari, il recupero di contenuti multimediali, perfino altri data sources (ad esempio Rss feeds) che possono completare le informazioni estratte attraverso i data services, e possibilmente essere scandagliati attraverso i servizi di analisi, con aspetti addizionali utili. Per questi aspetti, la capacità dell'utente di creare innovazioni è largamente favorita.

Questa classificazione di servizi ci permette di identificare alcuni comportamenti di default che ogni mashup basato su questi servizi dovrebbe avere e che perciò possono essere gestiti automaticamente dalla piattaforma attraverso l'aggiunta di listener nel modello di composizione. Per esempio, i filtri sono produttori di parametri usati per selezionare dati attraverso il data service. I viewers sono consumatori di dati, cioè elaborano i contenuti (possibilmente filtrati) estratti attraverso il data service e producono dati aggregati. Alcuni componenti come ad esempio i data services possono essere sia consumatori che produttori. Oltre al comportamento del flusso dati è necessario modellare la sincronizzazione tra il composition handler e i differenti componenti per permetterci di gestire gli eventi di composizione. La composizione difatti è trattata come l'esecuzione di un mashup. La principale sincronizzazione tra differenti classi di componenti è basata su alcuni eventi sollevati che vengono gestiti dall'execution handler, da cui deriva un semplice modello di composizione. Per esempio, per gestire l'aggiunta di una sorgente dati in un viewer box, il composition handler pubblica l'evento `updateDataService` che scatena l'operzione `setDataSource`, attraverso la quale il data service è configurato per recuperare i dati dalla sorgente dati aggiunta e visualizzarli nel viewr box dove la composizione ha luogo, attraverso un viewer. Analogamente viene definito un listener tra il composition handler e il componente filtro per gestire l'aggiunta o la modifica di un filtro. L'aggiunta di un viewer corrisponde a due differenti listener: un listener che accoppia il composition handler e il data service, per gestire la costruzione delle query aggregate, e un listener che accoppia il data service allo specifico componente viewer per informare il viewer stesso quando il data service ha completato l'estrazione dei dati aggregati necessari e li rende disponibili all'ambiente di esecuzione lato client. I listener relativi al composition handler vengono

aggiunti di default ad ogni composizione. In questo modo il composition handler è in grado di ascoltare qualsiasi evento di composizione. Un altro tipo di listener che deve essere aggiunto automaticamente si basa sullo specifico componente che l'utente inserisce nel mashup. Ad esempio dei listener relativi ai viewer possono dipendere dai parametri o dalle operazioni esposte dal servizio specifico. Infine alcuni listener possono essere aggiunti dall'utente finale per avere sincronizzazioni aggiuntive. Questo significa che l'utente deve dare indicazioni sull'accoppiamento dei componenti. Comunque questo ambiente visuale è ancora in grado di supportare utenti inesperti attraverso dialog-windows intuitive. La conoscenza del possibile mappaggio è codificata all'interno del composition handler e può essere facilmente configurata con il vantaggio risultante che DashMash può essere facilmente adattato a domini con eventualmente diversi servizi e classificazioni. In estremo il DashMsh può anche lavorare in assenza di classificazioni dando all'utente la libertà di accoppiare i componenti in qualsiasi maniera.

Capitolo 4

La generazione di raccomandazioni per la composizione del mashup

Come già asserito nel titolo, questo capitolo si pone l'obiettivo di illustrare i vari passi che compongono l'algoritmo implementato, che supporta il calcolo di compatibilità, similarità e qualità dei componenti e dei mashup. L'implementazione dell'algoritmo viene descritta passo passo, con elevato livello di dettaglio in ogni sua caratteristica e peculiarità, facendosi scrupolo di rimandare al capitolo successivo e all'appendice A per la descrizione dell'architettura di riferimento e delle tecnologie utilizzate.

4.1 Il processo a supporto della composizione

Come primo passo per supportare la composizione, si va ad analizzare la compatibilità tra due componenti generici: in questa prima fase si opera a livello sintattico, valutando se due componenti dal punto di vista sintattico hanno gli stessi operazioni, eventi, input ed output. L'analisi sintattica di compatibilità viene poi compattata e sintetizzata all'interno di una matrice di compatibilità, che si forma di elementi dal valore booleano che determinano in maniera esclusiva se ci sia o meno compatibilità.

A questo punto nel sottoinsieme di componenti che sono risultati tra loro compatibili si vanno ad individuare i valori percentuali di similarità, che vanno a determinare e ad individuare l'interscambiabilità tra questi componenti e le

possibili alternative di scelta tra componenti simili, servendosi delle annotazioni di similarità.

Altra componente non meno rilevante ai fini di condurre le scelte degli utenti è fornita dai tag relativi alla qualità, che individuano a loro volta i valori qualitativi di ogni componente o gruppo già aggregato di componenti a seconda di come gli utenti hanno stimato e valutato il suddetto.

4.2 Annotare un componente

Per poter capire come annotare unb componente secondo la similarità, si forniscono delle nozioni di come muoversi nell'utilizzo del tesauro WordNet e delle ontologie specifiche. Come punto di partenza è necessario annotare i componenti grezzi secondo dettagliati criteri, che permettono di conferire al componente delle caratterizzazioni univoche utili per valutarlo in relazione agli altri componenti e per fare gli opportuni confronti. Vi sono pertanto le annotazioni relative alla similarità e quelle relative alla qualità. Questo tipo di descrizione o *tagging* del componente permette di riconoscere le annotazioni relative alla similarità e alla qualità.

Le annotazioni relative alla similarità si distinguono in due categorie, cioè da una parte quelle relative a WordNet e che quindi si appoggiano a questo tesauro come supporto per il calcolo della similarità, dall'altra quelle attinenti alle ontologie, che sono visibilmente più onerose e complesse. Ricordiamo comunque che la componente delle ontologie è indispensabile per avere un calcolo della similarità efficiente, dal momento che WordNet da solo presenta qualche svantaggio, come si evidenzierà anche in seguito. Si sono dunque introdotti i tag aggiuntivi nell'operazione che sono volti a facilitare l'individuazione della specifica similarità del componente:

- similarity:meaning contiene il significato unico già utilizzato nel calcolo della compatibilità;

- similarity:verb è il tag che individua l'azione svolta dal componente sotto forma di verbo all'infinito;

- similarity:object individua un eventuale complemento oggetto a compendio del predicato verbale atto a specificare con più dettaglio, ove necessario, l'operazione svolta dal componente.

Per quanto riguarda i tag aggiuntivi di input e output si hanno invece le seguenti sfumature di significato:

- similarity:meaning costituisce una sorta di denominazione dell'input/output stesso;
- similarity:semantic area di nuovo viene ad individuare l'area semantica cui appartiene il nome assegnato all'input/output del componente.

Dunque input e output si comportano in modo analogo per quanto riguarda il tagging, mentre le operazioni hanno un comportamento simile ma con delle caratteristiche leggermente diverse, come enunciato sopra.

Si procede dunque esponendo come modificare nello specifico il componente. Inizialmente parleremo delle annotazioni relative a Wordnet; il componente presenta i tag relativi alle operazioni e agli eventi, che andranno modificati secondo le istruzioni di cui segue. Il tag iniziale si presenta nella seguente forma: <operation name=¨Get address¨ ref=¨getAddress¨ methodMeaning=¨getAddress¨>. L'annotazione consiste nell'aggiunta di questo contenuto: similarity:meaning=getAddress similarity:verb=get similarity:object=address. Come è facile notare si è scelto di dare una caratterizzazione dell'operazione (in questo caso, ma analogamente ci si comporta con gli eventi), in termini di verbo e oggetto. Sarà compito dell'algoritmo agganciarsi a WordNet e leggere questi due specifici parametri, ma come avvenga questa azione verrà trattato nei successivi capitoli.

A questo punto si descrive il componente con le annotazioni relative alle ontologie. Questa operazione risulta un po' più complicata, perché prevede inizialmente di fare una analisi delle aree semantiche in cui vanno a trovarsi i termini relativi agli input e agli output delle operazioni o degli eventi. Quando si sono isolate queste aree si procede ad individuare le ontologie che possono racchiudere questi campi semantici. Per attuare questa ricerca si consiglia di servirsi di un motore di ricerca specifico per ontologie, chiamato Swoogle [11], che permette di reperire una vasta gamma di ontologie, dalle più specifiche alle più generali. In questo modo si hanno gli elementi necessari per proseguire nell'annotazione.

Le operazioni e gli eventi racchiudono input, o alternativamente output, che si presentano secondo questa convenzione: <input name=¨Longitude¨ type=¨xsd:string¨ parMeaning=¨longitude¨> </input>. Si cerca una ontologia che possa racchiudere in sé il termine 'longitude'; si individua in particolare l'ontologia Geography.owl, molto completa e dettagliata. Trovata l'ontologia si modifica l'annotazione del componente nel seguente modo: <input name=Longitude type=xsd:float similarity:meaning = Geography.owl#Longitude similarity:semantic-area=Geography.owl/>. A questo punto, si ottiene un componente pronto per essere analizzato secondo i dati di similarità.

Diamo di seguito un esempio di annotazione di similarità:

```
<component id="componente_di_prova" name="componente_di_prova" description="
    Composition Handler is very useful component that..." adapter="" address="">

        <operation name="getData" description="this operation get data from.."
            dialog-descripion="get data from..." address="getData" similarity-
            meaning="UpdateDate" similarity-verb="update" similarity-object="date
            ">

            <param name="param1" description="parametro 1" direction="input" type
                ="xsd:string" validation-regex="*" validation-minvalue="0"
                validation-maxvalue="5"
                similarity-meaning="Geography.owl#Latitude" />

            <param name="param2" description="parametro 2" direction="output"
                type="xsd:string"
                similarity-meaning="Geography.owl#Longitude" />
        </operation>

        <operation name="getData2" description="this operation get data from.."
            address="getData" similarity-meaning="UpdateDate" similarity-verb="
            update" similarity-object="date">

            <param name="param1" description="parametro 1" direction="input" type
                ="xsd:string" validation-regex="*" validation-minvalue="0"
                validation-maxvalue="5"
                similarity-meaning="Geography.owl#Longitude" />

            <param name="param2" description="parametro 2" direction="output"
                type="xsd:string"
                similarity-meaning="Geography.owl#Latitude" />
        </operation>

        <event name="dateUpdated" address="dateUpdated" dialogdescripion="on date
             updated..."
```

```
        similarity-meaning="UpdateDate" similarity-verb="update" similarity-
            object="date">

        <param name="response" direction="output" type="xsd:string"
            similarity-meaning="Geography.owl#Longitude" />
    </event>

    <event name="dateRemoved" address="dateUpdated"
        similarity-meaning="UpdateDate" similarity-verb="update"
        similarity-object="date">

        <param name="response" direction="output" type="xsd:string"
            similarity-meaning="Geography.owl#Latitude" />
    </event>
</component>
```

Resta da valutare il componente secondo la qualità [15]. Il modo di annotare in relazione alla qualità è certamente molto più variegato e sfaccettato: sono molte infatti le caratteristiche che determinano la qualità di un componente. Solo per dare un'indicazione in merito possiamo menzionare la performance (throughput e response time), il costo, la disponibilità, l'affidabilità, la sicurezza, l'autenticazione, la precisione e innumerevoli altre se ne potrebbero individuare. Inoltre, non bisogna assolutamente dimenticare che anche il grado di soddisfazione degli utenti costituisce una forte informazione per la determinazione della qualità di un componente o eventualmente di un intero mashup.

Alla luce di tutte queste valutazioni e della schematizzazione di figura 4.1, non resta che annotare il componente secondo le specifiche che sono di seguito descritte:

- il tag di reputazione, *reputation*, che racchiude un valore compreso tra 0 e 1 che valuta appunto la reputazione del componente stesso;
- il tag di linguaggio, che racchiude il linguaggio o i linguaggi di cui si serve il componente;
- il tag relativo ai DataFormat, che racchiude il formato o i formati con cui vengono trasferiti e comunicati i dati;
- il tag di sicurezza, che indica che tipi di protezione caratterizzano il componente o in alternativa se il componente non ha alcun tipo di garanzia di sicurezza;

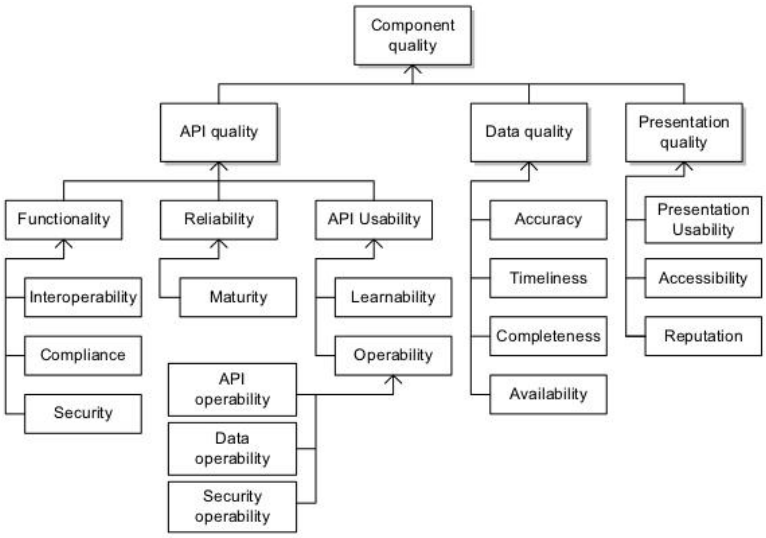

Figura 4.1: Sintetizzazione della struttura di qualità di un mashup

- il tag relativo alla puntualità del componente, *timeliness*, che indica il livello di reattività in termini di tempo del componente stesso ed è indicata con dei valori che sono compresi tra 0 e 1;

- il tag relativo all'accuratezza, che indica quanto sono precisi i dati che il componente fornisce, o meglio, quanto è accurato il componente. Di nuovo abbiamo dei valori compresi tra 0 e 1;

- il tag relativo alla completezza, che indica quanto sono completi i dati che il componente fornisce, o meglio, quanto è completo il componente. Ancora una volta i valori che esprimono tale grandezza sono compresi tra 0 e 1;

- il tag relativo alla disponibilità o accessibilità, che indica quanto un componente è disponibile o accessibile e viene indicata con una gamma di possibili valori che va da 0 a 1.

Per quanto riguarda i dati di qualità del componente elenchiamo i seguenti (che rispecchiano esattamente i tag che si trovano all'interno delle annotazioni del componente stesso):

- funzionalità a livello esterno; il criterio di funzionalità può essere scomposto negli attributi di interoperabilità e sicurezza:
  - l'interoperabilità è una delle più importanti dimensioni che contribuiscono alla qualità di un componente di un mashup;
  - la sicurezza di un componente è legata al meccanismo di protezione che è usato per accedere alle funzionalità offerte.
- affidabilità: per la valutazione dell'affidabilità a livello esterno è possibile considerare una misura della maturità del componente, che si basa principalmente su statistiche d'uso e sulla frequenza dei suoi cambiamenti o aggiornamenti;
- l'usabilità dell'API si riferisce alla facilità d'uso dell'API e può essere misurata in base alla comprensibilità, alla facilità di apprendimento e all'operabilità;
- dati relativi agli attributi di qualità percepita. Gli attributi di qualità percepita misurano il grado di positività dell'esperienza di utilizzo dell'applicazione, di conseguenza tali attributi sono in genere valutati tramite tecniche di ispezione che coinvolgono valutatori esperti, ma anche utenti comuni, e ad essi vengono associati dei valori che riassumono i risultati di queste analisi. Gli attributi principali di questo tipo sono usabilità, accessibilità e reputazione, esposti già nell'ambito delle annotazioni di qualità.

Per una migliore comprensione si riporta un esempio di annotazione di qualità:

```
<qualityAttributes>
            <reputation>0.9</reputation>
            <languages>
                <language>javascript</language>
            </languages>
            <dataFormats>
                <dataFormat>pvp</dataFormat>
            </dataFormats>
            <security>no authentication</security>
            <timeliness>0.9</timeliness>
            <accuracy>0.9</accuracy>
            <completeness>0.8</completeness>
            <availability>1</availability>
    </qualityAttributes>
```

4.3 Il calcolo della compatibilità

Introduciamo ora l'algoritmo per la valutazione della compatibilità tra componenti, il cui risultato è la matrice di compatibilità. La valutazione della compatibilità viene fatta in base ai tipi di ciascun input, output o operazione, evento di ciascun componente, cioè solamente a livello sintattico. Si ottiene compatibilità solo nel caso in cui ci sia una corrispondenza perfetta tra tutti i tipi presenti. A rigore di precisione è bene fare una distinzione tra parametri obbligatori e parametri opzionali; questi ultimi infatti non costituiscono un ostacolo per la compatibilità, in quanto essendo solo opzionali assumono un ruolo meno importante rispetto a quelli obbligatori, i quali invece costituiscono il nucleo fondante per la valutazione della compatibilità. Per rendere le cose più semplici si può supporre di avere una matrice che sarà riempita di 0 e 1; si avrà uno 0 nel caso di assenza di compatibilità, mentre si avrà 1 in caso contrario. Si ottiene così una matrice sparsa, che accentuerà tanto più questa caratteristica, quanto più numerosi risulteranno i componenti da analizzare. L'analisi di similarità sarà pertanto condotta semplicemente per quelle coppie di elementi, in corrispondenza dei quali si trova un 1 nella matrice di compatibilità.

La matrice di compatibilità risulta strutturata su confronti tra coppie di operazioni ed eventi e varia a seconda che si stia analizzando la compatibilità seriale o quella parallela. Per avere un riferimento chiaro proponiamo degli esempi generici nelle tabelle 4.1 e 4.2. L'utente dunque non solo potrà visionare i dati disaggregati, ma anche il valore aggregato di compatibilità, come mostrato nella sezione relativa alle prove.

4.3.1 L'algoritmo

Alla luce delle spiegazioni fatte, presentiamo di seguito lo pseudocodice relativo all'algoritmo per il calcolo della compatibilità. Innanzitutto facciamo le seguenti premesse per favorire una migliore comprensione. Dato un insieme C di componenti, definiamo CompatibilitàSeriale[c_i,c_j] la matrice di compatibilità seriale dei componenti e CompatibilitàParallela[c_i,c_j] la matrice di compatibilità parallela tra componenti. Diciamo inoltre che $(E|O)_{k_{c_i}}$ indicano il k-esimo evento o la k-esima

	$(O\|E)_{1_{c1}}$	$(O\|E)_{2_{c1}}$	...	$(O\|E)_{n_{c1}}$	...	$(O\|E)_{1_{cj}}$	$(O\|E)_{2_{cj}}$	...	$(O\|E)_{n_{cj}}$
$(O\|E)_{1_{c1}}$	1	1	1	0	0	1	0	0	1
$(O\|E)_{2_{c1}}$	0	0	0	1	0	0	0	1	0
...	1	1	1	0	0	1	0	0	1
$(O\|E)_{n_{c1}}$	0	0	0	1	0	0	0	1	0
...	0	0	0	0	1	0	0	1	0
$(O\|E)_{1_{cj}}$	1	1	1	0	0	1	0	1	0
$(O\|E)_{2_{cj}}$	0	0	0	1	0	0	0	1	0
...	1	1	1	0	0	1	0	0	1
$(O\|E)_{n_{cj}}$	1	1	1	0	0	1	0	1	0

Tabella 4.1: Matrice generica della compatibilità parallela

	$O_{1_{c1}}$	$O_{2_{c1}}$	...	$O_{n_{c1}}$	...	$O_{1_{cj}}$	$O_{2_{cj}}$	...	$O_{n_{cj}}$
$E_{1_{c1}}$	1	1	1	0	0	1	0	0	1
$E_{2_{c1}}$	1	1	1	0	0	1	0	0	1
...	1	1	1	0	0	1	0	0	1
$E_{n_{c1}}$	0	0	0	1	0	0	0	1	0
...	0	0	0	0	1	0	0	1	0
$E_{1_{cj}}$	1	1	1	0	0	1	0	1	0
$E_{2_{cj}}$	1	1	1	0	0	1	0	0	1
...	1	1	1	0	0	1	0	0	1
$E_{n_{cj}}$	0	0	0	1	0	0	0	1	0

Tabella 4.2: Matrice generica della compatibilità seriale

operazione dell'i-esimo componente. Le funzioni che regolano il nostro algoritmo sono:

- funAnalisiSeriale(c_i,c_j): funzione che valuta la compatibilità seriale tra due componenti (restituisce un valore booleano 0 o 1);
- funAnalisiParallela(c_i,c_j): funzione che valuta la compatibilità parallela tra due componenti (restituisce un valore booleano 0 o 1).

funAnalisiParallela(c_i,c_j)
begin
if $\exists((O_{k_{c_i}} = O_{k_{c_j}}) \bigwedge (E_{k_{c_i}} = E_{k_{c_j}}))$ **then**
  CompatibilitàParallela$[c_i, c_j] = 1$;
else
  if $\neg\exists((O_{k_{c_i}} = O_{k_{c_j}}) \bigwedge (E_{k_{c_i}} = E_{k_{c_j}}))$ **then**
    CompatibilitàParallela$[c_i, c_j] = 0$;
  end if
end if
end
funAnalisiSeriale(c_i,c_j)
begin
if $\exists(O_{k_{c_i}} = E_{k_{c_j}})$ **then**
  CompatibilitàSeriale$[c_i, c_j] = 1$;
else
  if $\neg\exists(O_{k_{c_i}} = E_{k_{c_j}})$ **then**
    CompatibilitàSeriale$[c_i, c_j] = 0$;
  end if
end if
end
compatibilità(C)
begin
for all $c_i, c_j \in C$ **do**
  $funAnalisiParallela(c_i, c_j)$;
  $funAnalisiSeriale(c_i, c_j)$;
end for

return CompatibilitàSeriale[c_i, c_j], CompatibilitàParallela[c_i, c_j];
end

4.4 Il calcolo della similarità

Dopo aver ottenuto la matrice di compatibilità, si va a stabilire la matrice di similarità relativamente a quel sottoinsieme di componenti che sono risultati compatibili. La similarità che si ottiene con l'algoritmo formulato prevede due parti distinte che vanno poi a comporre la matrice di similarità: una prima parte prevede appunto l'utilizzo di WordNet mentre la seconda è composta tramite l'uso delle ontologie e sarà trattata in dettaglio nel seguito. Si è innanzitutto analizzato il funzionamento di WordNet e come fosse composto al suo interno: si è dunque notato che tramite WordNet non si poteva ottenere il concetto di parola presa singolarmente (come menzionato più approfonditamente nell'appendice A), ma solo il concetto di Synset, cioè un insieme dei sinonimi della parola in analisi. Come sottoinsiemi dei synset si sono individuati verbsynset e nounsynset che rispettivamente permettono di concentrarsi specificamente sui verbi e sui nomi. La parte di similarità ottenuta con WordNet è atta soprattutto a fornirci un livello di similarità dei tag "operazione" ed "evento" che sono posti nei componenti e che individuano l'azione che il componente supporta. Si è notato che WordNet si comporta molto positivamente nel confronto tra verbi, ma risulta più fallimentare nel caso dei nomi. Nel caso dei nostri tag di operazione i nomi comparivano solo come complementi oggetti e quindi ai nostri scopi WordNet forniva comunque delle buone rispondenze in termini di similarità attesa a buon senso e similarità effettiva stimata tramite il nostro algoritmo. Tale algoritmo prevede di partire da due synsets in principio e di prendere le wordform di ciascuno di questi due e successivamente di effettuare un confronto e di valutare quante corrispondenze si individuavano. Si passava dunque ad individuare gli hyperonimi di ciascuna delle componenti facenti parte di entrambe le wordform. In questo modo si procede pensando a WordNet come ad un grande e articolato albero semantico, che assume forme particolari a seconda dei vocaboli che si stanno analizzando. Individuati gli hyperonimi si è quindi ripetuta l'analisi di individuazione delle corrispondenze dei due insiemi di hyperonimi in modo da valutare la similarità ad un ulteriore livello

di dettaglio. All'interno degli hyperonimi si sono ovviamente incluse le wordform in modo da tenere traccia dei livelli precedenti e non perdere l'eventuale similarità tra due concetti o meglio tra due termini che sono uno sottoclasse dell'altro. Per quanto riguarda i nomi si è scelto di risalire gli hyperonimi fino al terzo livello, cioè quattro livelli totali di dettaglio, uno fornito dalle wordform e tre ulteriori livelli di dettaglio forniti dagli hyperonimi. Questo ci è stato utile per ottenere dei risultati fruibili e sensati da WordNet dal momento che anche tra sinonimi in alcuni casi forniva scarsi coefficienti di similarità. Sempre per questo motivo si è scelto di dare dei pesi a ciascun livello indagato in modo da trovare dei risultati ottimali e da ridurre al minimo il livello di errore tra la similarità attesa e la similarità stimata con Wordnet. A tale proposito si sono condotte numerose prove che ci hanno permesso di valutare i coefficienti dei pesi da assegnare a ciascun livello. Tali coefficienti si possono valutare e ricalcolare utilizzando il nostro algoritmo nella parte relativa a Wordnet. Ricordiamo che ci è utile avere dei coefficienti sia per i verbi, che per i nomi, dal momento che le operazioni sono generalmente espresse o come semplice verbo, o come verbo e oggetto (ad esempio *find* o *find restaurant*). Non ci sono problemi relativi alla scarsa affidabilità di WordNet coi nomi, dal momento che nel caso delle azioni con verbo e oggetto il fattore dominante è comunque l'azione espressa dal verbo e il nome è solo una sorta di corollario.

Per quanto riguarda la similarità relativa ai nomi, si è convenuto di adottare i seguenti coefficienti pesati e valutati per ciascun livello(come mostrato graficamente in figura 4.2):

- peso relativo alle wordform: 0
- peso del primo livello di hyperonimi: 0.3
- peso del secondo livello di hyperonimi: 0.4
- peso del terzo livello di hyperonimi: 0.3

Per i verbi si è seguita una procedura analoga. In questo caso però si e scelto di risalire solo fino al secondo livello degli hyperonimi dal momento che se si fosse scelto di risalire troppo, come ovvio, si sarebbe trovata una esplosione degli insiemi di similarità e di conseguenza un dato di similarità molto elevato anche tra verbi

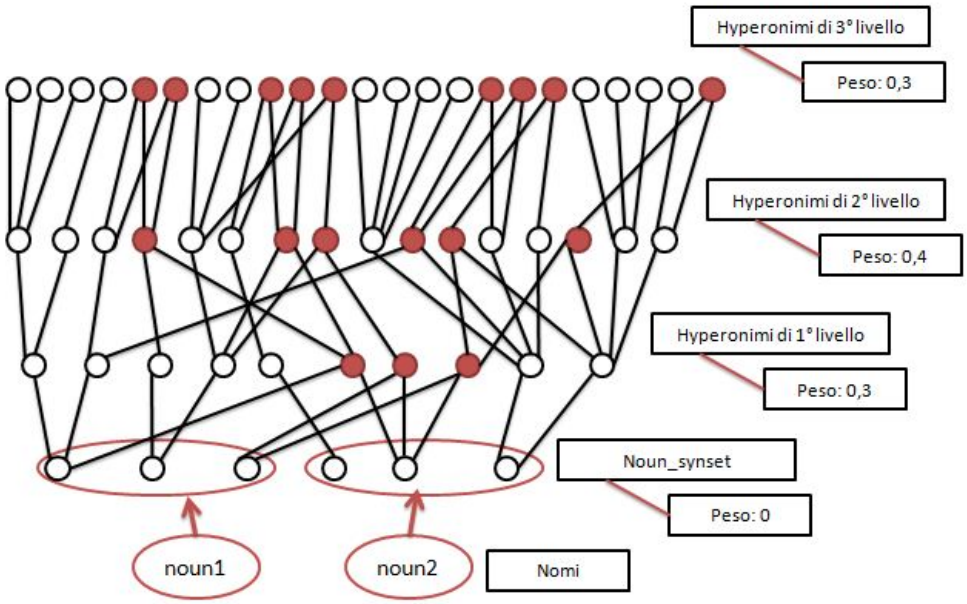

Figura 4.2: La similarità dei nomi con WordNet

appartenenti ad aree semantiche completamente diverse e addirittura antitetiche in certe situazioni. Sempre nello stesso modo si è condotta una prova di verifica che riassume i livelli di similarità tra verbi per i quali si sono utilizzati i seguenti coefficienti (come mostrato anche graficamente in figura 4.3):

- peso relativo alle wordform: 0.1
- peso del primo livello di hyperonimi: 0.8
- peso del secondo livello di hyperonimi: 0.1

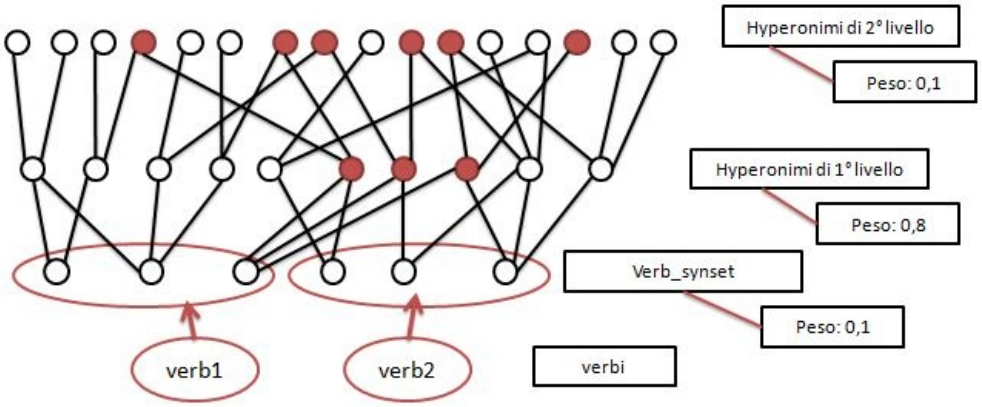

Figura 4.3: La similarità dei verbi con WordNet

A fronte delle prove effettuate anche con componenti taggati e proposti come esempio si è verificato che questi coefficienti assegnati ai livelli portano a risultati effettivamente soddisfacenti e inerenti a quello che realmente ci si aspetta di individuare. Per avere un'idea migliore riportiamo un sottoinsieme di vocaboli tra quelli utilizzati nelle sperimentazioni per avere una prova concreta di come l'algoritmo funzioni correttamente con i valori e i pesi scelti per i nomi e per i verbi, come si vede nelle tabelle riportate nelle pagine seguenti (tabelle 4.3, 4.4, 4.5 e 4.6). Per risultati complessivi su componenti e non su semplici vocaboli rimandiamo

al capitolo successivo, dove abbiamo una sezione interamente dedicata alle prove dell'intero applicativo composto dai tre algoritmi implementati in questa tesi.

Bisogna ora spiegare come avviene il calcolo della similarità dei parametri. Come già accennato in precedenza, la similarità dei parametri dei componenti, cioè gli input e gli output viene determinata tramite l'utilizzo delle ontologie. Di nuovo si deve immaginare di avere un albero, solo che in questo caso l'albero viene costituito tramite le ontologie, invece che con Wordnet. È noto che tutti i concetti di una ontologia discendono più o meno direttamente dal concetto "thing", a partire da questo si diramano tutte le componenti lessicali e si specializzano o nella stessa ontologia o in ontologie più specifiche. Risalendo l'albero generato dai due termini che si stanno esaminando, si incontrerà prima o poi un nodo in comune; una volta raggiunto il nodo si stimerà la distanza dei due termini dal nodo individuato (come mostrato in figura 4.4). Tramite questo calcolo si giunge al livello di similarità. In particolare per stimare questa componente della similarità avremo le seguenti operazioni:

- creare un grafo a partire dalla gerarchia di ontologie atto a mappare tutti i concetti presenti in esse;

- in questo caso tutti i concetti vengono visti come come delle *entity class* e posseggono delle sovraclassi e delle sottoclassi.

Date le precedenti premesse si procede nel calcolo vero e proprio della similarità tramite una funzione valutata in modo da ottenere i risultati desiderati. Tale funzione prevede di risalire il grafo fino al nodo padre di tutto per vedere quanto il grafo stesso sia profondo. Successivamente si individua la distanza dei due concetti presi in esame dal parente più vicino che hanno in comune in modo da restare il più possibile lontano dalla sommità del grafo. Infine si valuta la similarità tramite la funzione di cui poco sopra. Schematizziamo i passaggi fatti nel seguente modo (rimandiamo allo pseudocodice proposto alla fine della sezione per una schematizzazione globale):

- si prendono i concetti in comune tra i due concetti che si stanno esaminando: get_common_class(OntClass C1, OntClass C2);

nome1	nome2	similarità ottenuta
orb	ball	0,99
vocabulary	wording	0,05
illness	trouble	0,54
food	sky	0,15
sea	sky	0,04
address	url	0,81
trip	expedition	0,32
vegetable	potato	0,26
vegetable	fruit	0,05
church	hospital	0,42
sofa	water	0,12
cup	glass	0,19
glass	box	0,2
trumpet	french-horn	0,98
banjo	harmonica	0,24
pan	plate	0,39
barbecue	casserole	0,82
way	road	0,95
latitude	longitude	0,68
button	label	0,19
skirt	gun	0,11
car	van	0,2
bacon	bathroom	0,03
skirt	car	0,06
zoom	interface	0,22
day	date	0,4

Tabella 4.3: Tabella dei nomi (a)

nome1	nome2	similarità ottenuta
longitude	address	0,06
page	webpage	0,5
feed	page	0,1
url	page	0,06
feed	url	0,21
button	slider	0,15
slider	zoom	0,04
label	text	0,05
text	bar	0,3

Tabella 4.4: Tabella dei nomi (b)

verbo1	verbo2	similarità ottenuta
fly	write	0,06
drink	think	0,07
search	find	0,4
look	watch	0,39
spend	pay	0,27
speak	talk	0,65
say	tell	0,6
make	do	0,6
send	walk	0,13
look for	observe	0,27
eat	drink	0,18
sleep	wake	0,02
move	go	0,73
wait	eat	0,09
hear	take	0,45
take	use	0,71

Tabella 4.5: Tabella dei verbi (a)

verbo1	verbo2	similarità ottenuta
want	think	0,05
can	stay	0,15
give	take	0,57
bring	take	0,76
come	go	0,54
come	become	0,61
visit	come	0,17
be	become	0,18
travel	transfer	0,23

Tabella 4.6: Tabella dei verbi (b)

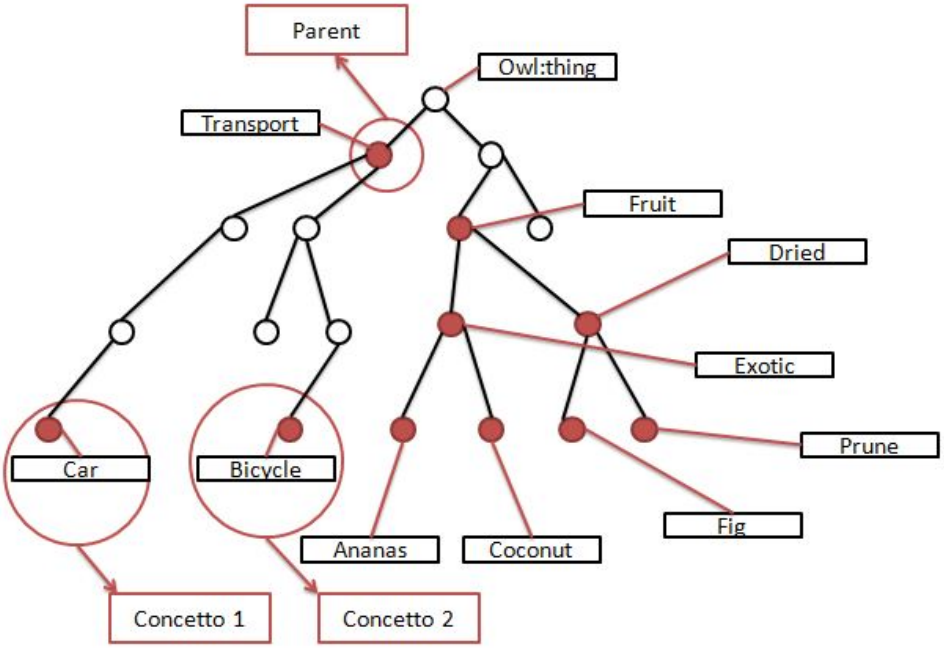

Figura 4.4: L'albero semantico ottenuto con le ontologie

- si sceglie il nodo più lontano da *Thing* e più vicino a C1 e C2 quindi più specifico;

- si effettua la misura della similarità con la seguente funzione:

$$S = \frac{c}{c + max\{a, b\}}$$

(funzione che viene esemplificata anche in figura 4.5). Bisogna però fare attenzione ad avere un valore di S che sia compreso tra 0 e 1, cioè:

$$0 < S < 1$$

Infine per completezza specifichiamo i significati dei parametri a, b e c:

  - c rappresenta la distanza tra il padre minimo, cioè il parente più vicino e OWL:Thing, cioè il padre di tutto;
  - a e b rappresentano le distanze tra i nodi dei due concetti in esame e il padre minimo.

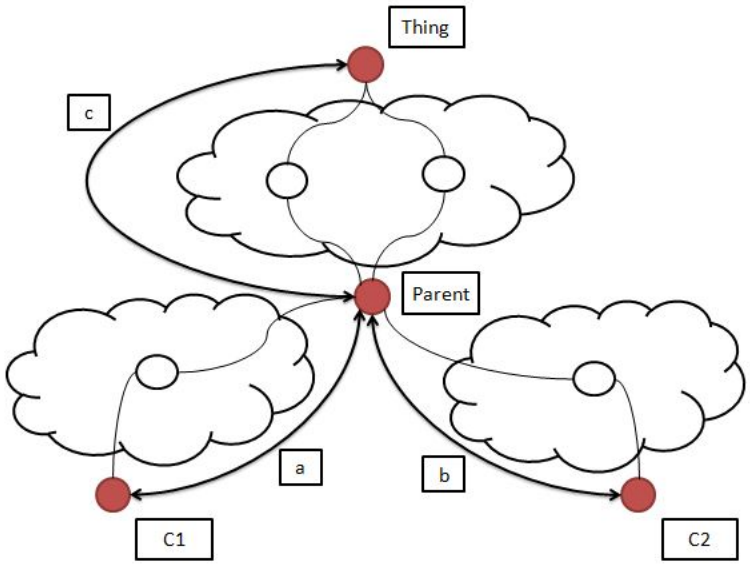

Figura 4.5: La funzione di similarità per le ontologie

A questo punto si riesce a calcolare la similarità finale, cioè quella composta congiuntamente dalla similarità dei parametri (input e output) e quella delle operazioni, o equivalentemente quella componente di similarità che si appoggia alle ontologie e quella che si appoggia a WordNet, tramite una media pesata tra queste due. In questa media pesata entra in gioco l'innovatività del nostro metodo soprattutto nei confronti dell'utente finale. Quest'ultimo infatti, potrà dare dei pesi diversi ai vari dati disaggregati, in modo da personalizzarsi la sua media pesata a seconda delle specifiche necessità che ha nella creazione del suo mashup. In particolare l'utente finale potrà variare i pesi assegnati alle due componenti della similarità a seconda che stia valutando la similarità verticale o orizzontale. Questa metodologia infatti ci permette di fare *reccomendation* nei confronti dell'utente che sta componentizzando il suo mashup, eventualmente anche se si tratta di un utente con scarsa competenza (come mostrato in figura 4.6).

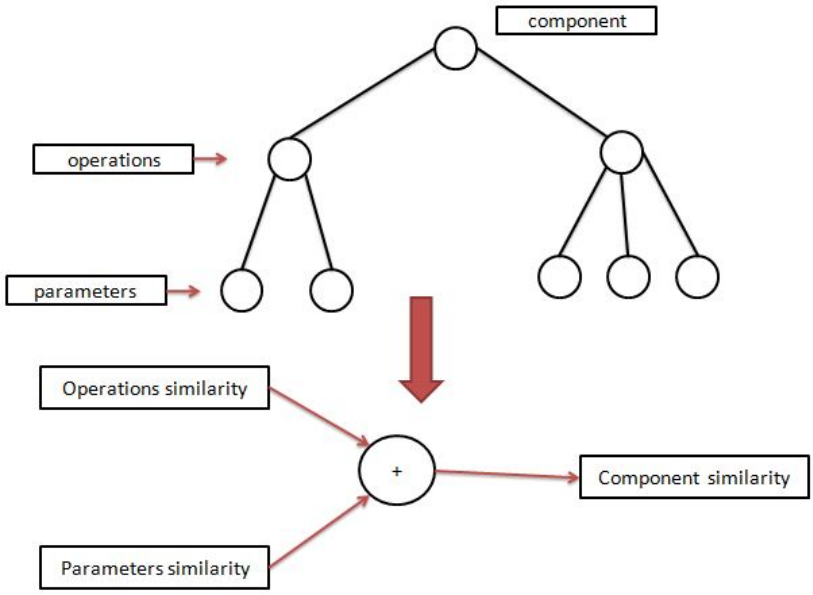

Figura 4.6: La media dei pesi per le personalizzazioni degli utenti finali

Tutto ciò rende il nostro algoritmo più preciso a seconda dei bisogni dell'utente, ma soprattutto è molto più fruibile ed intuitivo anche ai fini di una distribuzione di tipo commerciale, che sia volta in particolare alla soddisfazione e alla facilitazione dell'utente finale. L'idea che ha ispirato la generazione dell'algoritmo e

che ci ha guidato nell'implementazione delle specifiche funzionalità è costituita essenzialmente dalle metodologie descritte in [45].

4.4.1 Le criticità evidenziate

Risulta chiaro ora che le ontologie sono fondamentali ai fini della disambiguazione semantica nel caso dei nomi, soprattutto per lo specifico compito svolto dal nostro algoritmo. Bisogna però evidenziarne alcuni problemi, nell'ottica di avere dei miglioramenti futuri in questa direzione. Innanzitutto, come già diffusamente spiegato si ribadisce l'assenza di una repository globale e universale di tutte le ontologie, che costituisce un non indifferente disagio, soprattutto nella prima fase di annotazione. Come secondando problema legato alle ontologie si devono menzionare tutti quei problemi relativi al differente livello di dettaglio tra le varie ontologie e a volte nella stessa ontologia: in particolare, per meglio comprendere, si può pensare al caso in cui si confrontino delle terminologie legate ad ambiti più o meno vasti; ontologie più o meno ramificate avranno termini con distanze differenti dalla radice thing. Questo comporta gravi problemi nel dare una valutazione effettiva delle distanze e della loro incidenza sul grado di similarità. Questi differenti aspetti costituiscono delle problematiche non trascurabili e che vanno tenute in considerazione nell'ambito di successivi sviluppi. Rimandiamo al capitolo conclusivo per avere una argomentazione forte della necessità delle ontologie per gli scopi di questa tesi e per avere anche delle proposte di miglioramento per l'avvenire.

Per quanto riguarda la similarità è bene specificare anche un concetto che rappresenta un fattore fondamentale della nostra trattazione e cioè il fatto che la valutazione della similarità tra componenti è operata sia verticalmente, che orizzontalmente. Per comprendere meglio questa distinzione è opportuno pensare che l'utente finale non solo avrà bisogno di una lista di possibili componenti tra loro simili, che possano essergli utili nella sua creazione di un mashup, ma avrà anche bisogno di un matching tra gli output di un componente e gli input del componente che vorrebbe mettere in cascata al precedente. Si tratta quindi di fornire da una parte i dati di similarità verticale, cioè quelli che riguardano la similarità tra tutti gli input di tutti i componenti e tutti gli output di tutti i componenti, dall'altra

quei dati che riguardano invece la similarità orizzontale, cioè la valutazione della similarità tra gli output di tutti i componenti e gli input di tutti i componenti al fine di verificare quali componenti possano essere efficientemente messi in cascata ad altri.

4.4.2 L'algoritmo

Per avere una migliore idea di come sia strutturato l'algoritmo per il calcolo della similarità ne proponiamo un activity diagram che possa chiarirne il funzionamento (figura 4.7). Di seguito proponiamo invece lo pseudocodice, premurandoci di fare alcune doverose premesse. Dato C insieme di componenti, $(E|O)_{k_{c_i}}$ indicano il k-esimo evento o la k-esima operazione dell'i-esimo componente. definiamo le seguenti funzioni su questo insieme:

- funSimilaritàOrizzontale(c_i,c_j): funzione che calcola la similarità orizzontale tra due componenti;
- funSimilaritàVerticale(c_i,c_j): funzione che calcola la similarità verticale tra due componenti;
- getOntology: funzione che a partire da un input o da un output ne ritorna l'ontologia di appartenenza in forma di albero;
- findCommonNode: funzione che dati due nodi di un'ontologia risale l'albero delle ontologie fino a trovare un nodo in comune;
- weightedAverage: funzione che calcola la media pesata di due valori secondo pesi inseriti manualmente dagli utenti;
- WordNetGlobalSimilarity(Operation|Event): funzione di similarità che si basa sull'utilizzo di WordNet (valida sia per gli eventi che per le operazioni), dove i pesi utilizzati sono quelli discussi nella sezione dedicata al calcolo della similarità;
- getWordform: funzione che a partire da un nome o un vero restituisce le sue wordform in WordNet;

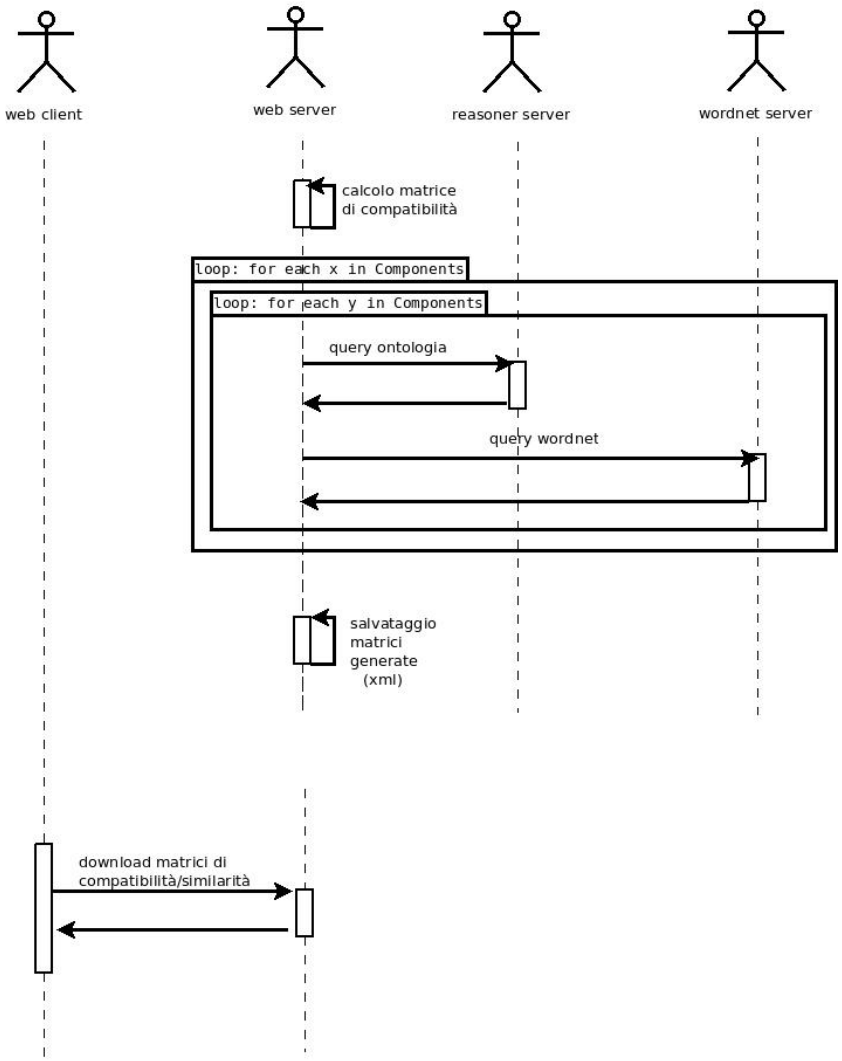

Figura 4.7: Activity diagram del funzionamento

- WordNetSimilarity(Name|Verb): funzione che calcola la similarità tra nomi o verbi di due operazioni o eventi,dove i pesi utilizzati sono quelli individuati nella sezione dedicata al calcolo della similarità;
- evaluateDistance(A,B): funzione che calcola la distanza tra due nodi nel grafo dell'ontologie;
- getCommonElements(A,B): funzione che trova gli elementi in comune tra due insiemi di vocaboli;
- getHyperonyms: funzione che a partire da un nome o un vero restituisce i suoi hyperonimi in WordNet.

funSimilaritàVerticale(c_i,c_j)
begin
for all $O_{k_{c_i}}$ **do**
  $getOntology(O_{k_{c_i}})$;
  for all $O_{k_{c_j}}$ **do**
    $getOntology(O_{k_{c_j}})$;
    $commonNode = findCommonNode(O_{k_{c_i}}, O_{k_{c_j}})$;
    $Aop = evaluateDistance(commonNode, O_{k_{c_i}})$;
    $Bop = evaluateDistance(commonNode, O_{k_{c_j}})$;
    $Cop = evaluateDistance(commonNode, owl : Thing)$;
    $Soperation = Cop \div (Cop + max\{Aop, Bop\})$
  end for
end for
for all $E_{k_{c_i}}$ **do**
  $getOntology(E_{k_{c_i}})$;
  for all $E_{k_{c_j}}$ **do**
    $getOntology(E_{k_{c_j}})$;
    $commonNode = findCommonNode(E_{k_{c_i}}, E_{k_{c_j}})$;
    $Aev = evaluateDistance(commonNode, E_{k_{c_i}})$;
    $Bev = evaluateDistance(commonNode, E_{k_{c_j}})$;
    $Cev = evaluateDistance(commonNode, owl : Thing)$;
    $Sevent = Cev \div (Cev + max\{Aev, Bev\})$

end for
end for
$E - Osimilarity = (Soperation + Sevent) \div 2$;
return similaritàVerticale= $weightedAverage(E - Osimilarity,$
$WordNetGlobalSimilarity(c_i, c_j))$;
end
funSimilaritàOrizzontale(c_i,c_j)
begin
for all $O_{k_{c_i}}$ **do**
$getOntology(O_{k_{c_i}})$;
for all $E_{k_{c_j}}$ **do**
$getOntology(E_{k_{c_j}})$;
$commonNode = findCommonNode(O_{k_{c_i}}, E_{k_{c_j}})$;
$A = evaluateDistance(commonNode, O_{k_{c_i}})$;
$B = evaluateDistance(commonNode, E_{k_{c_j}})$;
$C = evaluateDistance(commonNode, owl : Thing)$;
return similaritàOrizzotale= $C \div (C + max\{A, B\})$
end for
end for
end
WordNetGlobalSimilarity(Operation|Event)(c_i, c_j)
begin
for all $(O|E)_{k_{c_i},name}$ **do**
for all $(O|E)_{k_{c_j},name}$ **do**
$nameSim = WordNetSimilarityName((O|E)_{k_{c_i},name}, (O|E)_{k_{c_j},name})$;
end for
end for
for all $(O|E)_{k_{c_i},verb}$ **do**
for all $(O|E)_{k_{c_j},verb}$ **do**
$verbSim = WordNetSimilarityVerb((O|E)_{k_{c_i},verb}, (O|E)_{k_{c_j},verb})$;
end for
end for

return WordNetGlobalSimilarity(Operation|Event)= $(verbSim \times 0,7 + nameSim \times 0,3) \div 2$;
end
WordNetSimilarityName(i,j)
begin
$WordformI = getWordform(i)$;
$WordformJ = getWordform(j)$;
$getCommonElements(WordformI, WordformJ) = X$;
$WordformI + I.getHyperonyms = FirstHypI$;
$WordformJ + J.getHyperonyms = FirstHypJ$;
$getCommonElements(FirstHypI, FirstHypJ) = Y$;
$FirstHypI + FirtsHypI.getHyperonyms = SecHypI$;
$FirstHypJ + FirtsHypJ.getHyperonyms = SecHypJ$;
$getCommonElements(SecHypI, SecHypJ) = W$;
$SecHypI.getHyperonyms + SecHypI = ThirdHypI$;
$SecHypJ.getHyperonyms + SecHypJ = ThirdHypJ$;
$getCommonElements(ThirdHypI, ThirdHypJ) = Z$;
return nameSimilarity $= (0 \times X + 0,3 \times Y + 0,4 \times W + 0,3 \times Z) \div 4$;
end
WordNetSimilarityVerb(i,j)
begin
$WordformI = getWordform(i)$;
$WordformJ = getWordform(j)$;
$getCommonElements(WordformI, WordformJ) = X$;
$WordformI + I.getHyperonyms = FirstHypI$;
$WordformJ + J.getHyperonyms = FirstHypJ$;
$getCommonElements(FirstHypI, FirstHypJ) = Y$;
$FirstHypI + FirtsHypI.getHyperonyms = SecHypI$;
$FirstHypJ + FirtsHypJ.getHyperonyms = SecHypJ$;
$getCommonElements(SecHypI, SecHypJ) = Z$;
return verbSimilarity $= (0,1 \times X + 0,8 \times Y + 0,1 \times Z) \div 3$;
end

4.4.3 La complessità

Una parte imprescindibile della trattazione è valutare quale sia la complessità dell'algoritmo. Le situazioni che ci troviamo a dover gestire sono fondamentalmente tre:

- aggiunta di un componente nella repository;
- modifica di un componente nella repository;
- cancellazione di un componente dalla repository.

Se abbiamo n componenti dei quali valutare la similarità, è facile intuire, alla luce dell'algoritmo descritto, che il numero di confronti da fare è dell'ordine di n^2, in quanto si va a costruire una matrice che racchiude in sé i confronti a coppie di tutti i componenti caricati nell'ambiente. Un altro problema legato alla complessità è legato all'azione di aggiunta di un componente; in questo caso dovremo fare un numero di confronti aggiuntivi pari a n, per cui avremo una complessità aggiuntiva che è dell'ordine di n, come si vede in figura 4.8. Queste considerazioni danno un'idea del carico elaborativo che avviene in background all'interno del nostro algoritmo, che è indubbiamente molto articolato e soprattutto molto preciso e accurato.

Riportiamo per completezza i passaggi matematici che ci hanno condotto ad ottenere le stime di complessità poco sopra descritte. Per quanto riguarda il calcolo della similarità per i componenti presenti nella repository avremo i seguenti confronti:

$$\frac{n(n-1)}{2} \sim \Theta(n^2)$$

Nel caso in cui venga aggiunto un componente, come già spiegato avremo n ulteriori confronti, cioè pari alla seguente complessità: $\sim \Theta(n)$

Per ovviare, anche solo in parte a questi problemi sono stati fatti svariati interventi di ottimizzazione sull'algoritmo originario, per renderlo più performante. In primo luogo si opera il filtraggio dei componenti compatibili: già questa azione comporta una scrematura iniziale che va ad eliminare tutti i componenti che non sono compatibili e mi permette di procedere all'analisi della similarità su di un sottoinsieme di componenti. Come secondo intervento di ottimizzazione possiamo

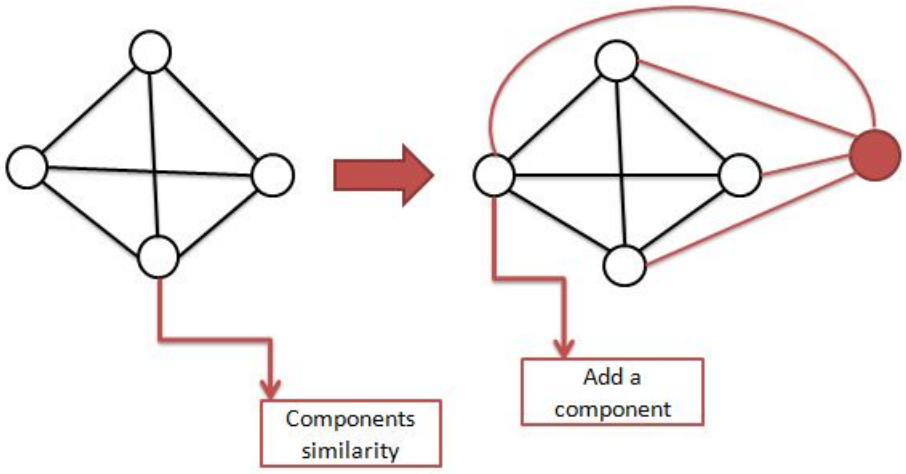

Figura 4.8: La complessità

utilizzare il caching multi-livello; in particolare ad un primo livello si salvano in ambito di sistema i dati di compatibilità e similarità sui componenti già presenti, in modo che di volta in volta si debbano solo calcolare i dati di compatibilità e similarità solo per i componenti che sono modificati e per quelli che sono aggiunti, premurandosi ovviamente di eliminare i dati relativi ai componenti che vengono cancellati dall'utente. Successivamente si vanno a salvare i dati relativi al dominio di uno specifico utente o di una categoria di utenti. Questo come già accennato nei precedenti paragrafi non solo costituisce una ottimizzazione, perché evita di calcolare dati inutili per un utente che si concentri su di una specifica area di interesse, ma crea anche un notevole grado di personalizzazione all'interno della piattaforma. L'utente singolo avrà quindi la possibilità di vedere solo i componenti che gli interessano appositamente filtrati dal *mare magnum* di possibili componenti disponibili.

I tre interventi di ottimizzazione descritti rendono indubbiamente più performante il nostro algoritmo e lo rendono aperto a possibili ulteriori miglioramenti in ambito di performance e di personalizzazione rivolta all'utente finale, ma questi aspetti saranno ulteriormente approfonditi nella sezione della presente trattazione

dedicata agli eventuali sviluppi futuri.

Bisogna comunque notare che le valutazioni di compatibilità e qualità vengono fatte lato server un'unica volta e si presentano in formato stabile all'utente finale. Gli aggiornamenti vengono fatti solo in caso di modifiche alla repository dei componenti. Questi aspetti della similarità la rendono paragonabile ad un servizio di manutenzione straordinaria della piattaforma DashMash.

4.5 La misura della qualità

Infine una parte fondamentale dell'algoritmo è data dal calcolo della qualità dei componenti [46, 47]. Come già accennato nella parte introduttiva è utile fornire una vasta gamma di dati di qualità, che in seguito daranno all'utente non solo informazioni sul dato di qualità complessivo, ma anche sui singoli pesi delle specifiche caratteristiche qualitative. Per spiegare meglio questo concetto è opportuno fare un piccolo esempio. Supponiamo che un utente debba fare la ricerca di un gruppo di ristoranti in un raggio di 3 km attorno alla sua abitazione: si troverebbe a scegliere tra un ricco gruppo di componenti possibili (o addirittura di mashup possibili), proposti ad esempio da Yahoo, da Google o da Bing alternativamente. L'utente medio farà pertanto una analisi della qualità anche in relazione ai suoi bisogni: ad esempio potrebbe servirgli una ricerca molto dettagliata e precisa, ma non troppo costosa, oppure una ricerca molto veloce, anche se costosa. In questo caso è facile vedere che importanza cruciale assume il dato di qualità non solo aggregato, ma anche scisso in tutte le possibili componenti in modo che l'utente possa fare tutte le sue considerazioni, a seconda delle proprie necessità. Nella ampia rosa di possibili valutazioni di qualità, si è scelto di analizzarne un sottoinsieme abbastanza corposo che isola quelle caratteristiche qualitative che sono di maggiore importanza per poter dare dei riscontri concreti all'utente. La qualità si compone di due parti fondamentali: la qualità intrinseca del componente o qualità locale e la qualità dell'intero mashup già componentizzato o qualità globale. Date queste specifiche caratterizzazioni è facile pensare che nello specifico vi sia interesse a spiegare dettagliatamente i ruoli dei parametri e delle metriche secondo i quali verrà calcolata la qualità del mashup.

Le caratteristiche di qualità esposte sopra sono certamente un consistente gruppo di possibili scelte; ai nostri scopi se ne sceglie un sottoinsieme di specifico interesse, lasciando eventualmente delle possibili estensioni nella sezione relativa agli sviluppi futuri.

Dopo queste premesse si comincia ad introdurre il funzionamento vero e proprio dell'algoritmo per i dati di qualità. In breve l'algoritmo di qualità si basa sull'individuare all'interno del mashup una struttura a grafo in cui i nodi sono i componenti e gli archi sono i collegamenti tra i componenti (i *binding*); in particolare questi archi sono orientati e seguono il flusso di output-input dei componenti in cascata. Per comprendere meglio la struttura e i criteri di importanza di un mashup, è bene descrivere i ruoli principali che un componente può assumere; questi ruoli sono essenzialmente tre e li elenchiamo qui di seguito [2]:

- slave: rappresenta il caso in cui il comportamento di un componente è condizionato dal comportamento di un altro componente; tipicamente il suo stato cambia in relazione al comportamento di un componente master;

- filter: questa categoria ha un ruolo marginale rispetto agli altri due. Il suo scopo è offrire meccanismi di accesso selettivo ai contenuti che un componente può mostrare, cioè l'utente può scegliere quale contenuto visualizzare all'interno dell'applicazione. I componenti filter, cioè, aiutano a ridurre il dataset di altri componenti;

- master: Identifica il componente con cui l'utente interagisce maggiormente, per questo la sua importanza è considerata più alta rispetto gli altri due ruoli.

Apriamo una breve parentesi sulle possibili configurazioni (figura 4.9) nelle quali possiamo trovare i componenti:

- slave-slave: è la configurazione più semplice. Un mashup che integra questo tipo di componenti permette all'utente di interagire con i componenti presenti, ma in maniera isolata: non c'è flusso di informazioni tra un componente e l'altro. Possono essere usati dei filtri per selezionare il dataset mostrato all'utente;

- master-master: è il percorso più completo, dove oltre ai componenti filtro, gli altri componenti svolgono il ruolo di master. Ciò significa che gli effetti dell'azione eseguita su un componente si ripercuotono su tutti gli altri, che si sincronizzano in accordo ai dati selezionati. Ciò significa che un componente master svolge il ruolo di slave quando l'azione viene eseguita sull'altro componente;

- master-slave: questo è il pattern più diffuso. E' basato su tutti e tre i ruoli visti sopra. Un componente filtro permette all'utente di restringere il set di dati mostrati simultaneamente dagli altri componenti. Il componente master permette all'utente di eseguire l'azione principale, per esempio selezionare il dato di interesse. Il componente slave si sincronizza automaticamente con la selezione fatta sul componente master, per esempio visualizzando i dettagli dell'elemento selezionato.

Alla luce di ciò, esponiamo il funzionamento dell'algoritmo relativo al calcolo della qualità. Innanzitutto si calcola tramite una funzione apposita l'importanza di ciascun nodo, in grado di riflettere il ruolo di mastre o slave dei vari componenti. Tale importanza dipende direttamente dal numero di cammini possibili che attraversano il nodo e che collegano le entrate e le uscite dal grafo o mashup. Successivamente si calcola la qualità globale del mashup come media pesata secondo l'importanza delle qualità di ciascun componente.

4.5.1 La qualità del componente

Ci dedichiamo nello specifico alla descrizione del calcolo della qualità del singolo componente, anche svincolato dal mashup. La qualità del componente viene calcolata in base alle annotazioni descritte nei precedenti paragrafi. Tale qualità non è altro che una media pesata, secondo pesi accuratamente determinati tramite accurate sperimentazioni, delle singole qualità del componente. Il fatto di tenere disaggregate tutte queste caratteristiche di qualità fornisce anche l'eventuale possibilità di valutare i dati finali del mashup in maniera disaggregata sempre facendo una media pesata a seconda dell'importanza del nodo.

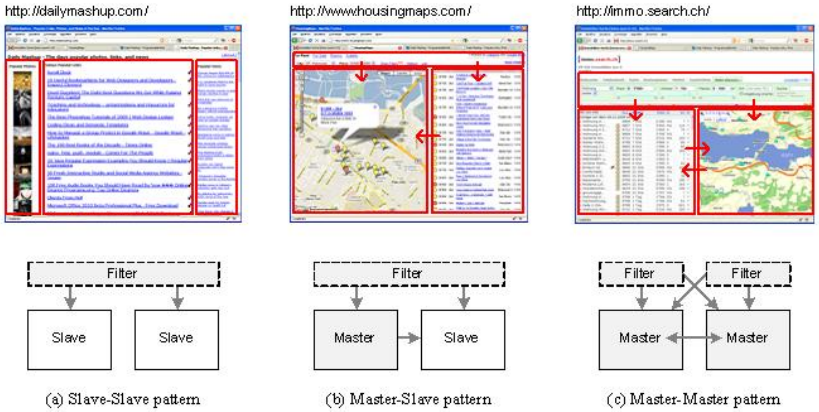

Figura 4.9: Possibili configurazioni dei componenti

4.5.2 La qualità aggregata

Presentiamo in questo breve paragrafo l'analisi topologica a supporto della qualità e le metriche utilizzate per attuarla. Tale analisi rende ancora migliore il nostro algoritmo nell'ottica dell'utilizzo da parte di un utente finale. Si crea infatti, la possibilità per l'utente di richiedere una analisi topologica nel momento in cui abbia ultimato la preparazione del suo mashup; l'analisi topologica gli permetterà di ottenere informazioni preziose circa la presenza di eventuali componenti critici e la qualità complessiva del mashup e la sua affidabilità. Per comprendere meglio la grande carica innovativa dell'analisi topologica ai fini del nostro algoritmo, basta pensare ad un mashup che concentri la sua attività su un componente centrale: se questo componente non è affidabile o è di scarsa qualità tutto il mashup ne risentirà. Risulta pertanto evidente l'essenzialità di valutare la centralità o importanza di un componente all'interno del mashup, in modo da valutare i suoi dati di qualità anche in relazione al suo ruolo all'interno del mashup; se un componente di scarsa qualità occupa un ruolo marginale nel mashup, non vi sono grandi problemi, perché il mashup potrebbe utilizzarlo poco e non subire gravi cali di affidabilità o performance. Tale studio topologico risulta quindi molto utile per l'utente perché valuta il dato di qualità locale in relazione alla topologia, in modo da rendere più accurato il dato di qualità globale, che viene valutata non solo come semplice media delle qualità dei componenti che lo costituiscono, ma come media pesata delle qualità locali di questi in relazione a specifiche metriche che ne valutano i livelli di centralità e connessione. Ricordiamo che tutti questi studi vengono operati agevolmente tramite l'utilizzo della libreria JUNG [40]. Bisogna ora dire quali siano queste metriche. Ci sono in particolare utili tutti i costrutti relativi ai grafi messi a disposizione dalla libreria JUNG in particolare il metodo che restituisce l'elenco dei successori di un nodo, che verrà ampiamente utilizzato all'interno dell'algoritmo di qualità, nell'ambito delle funzioni di importanza e per il calcolo del numero di cammini passanti per un determinato nodo e che collegano gli input agli output. Risulta evidente che queste misure legate ai mashup, intesi come rete, sono fondamentali per fornire ulteriori funzionalità ad un utente, soprattutto qualora, per esempio, egli voglia sostituire un componente malfunzionante, che abbia alta *importanza* con un altro componente simile.

4.5.3 L'algoritmo

Proponiamo a seguire lo pseudocodice che concerne l'algoritmo di qualità e che in particolare illustra le funzioni di basilare utilità che calcolano rispettivamente il numero di cammini che collegano gli input con gli output, passanti per un dato nodo, e l'importanza di un determinato nodo.

Forniamo alcune spiegazioni preliminari: dato C un insieme di componenti, utilizziamo le seguenti funzioni:

- mashupToGraph(m): funzione che dato un mashup, lo converte in grafo, facendo in modo che i componenti diventino nodi e che i binding diventino archi;

- numCammini(nodoA, nodoB): funzione che dati due nodi, calcola il numero di cammini che li collega;

- localQuality(ci): funzione che calcola la qualità di un singolo componente, valutandone i tag stringa e i tag numerici;

- get[](ci): funzioni che estraggono i valori dei tag numerici e di quelli in formato stringa (dataformat, language, security);

- weightedAverage(accuracy, availability, interoperability, security, timeliness, reputation, completeness): funzione che a partire dai dati di qualità, fa una media pesata secondo opportuni pesi;

- EvaluateSecurity(tagSec): funzione che valuta un dato numerico di sicurezza a partire dal tag di sicurezza;

- EvaluateInteroperability(tagLing, tagDf): funzione che valuta un dato numerico di interoperabilità a partire dai tag di linguaggio e dataformat;

- allNodesWeigtedAverage (Importanza(ci), localQuality(ci)): funzione che dati i valori di qualità dei nodi e la loro importanza calcola la qualità globale del mashup come media pesata secondo l'importanza di ciascun nodo delle qualità dei nodi stessi;

- Importanza(ci): funzione che calcola l'importanza del nodo ci come numero dei cammini passanti per quel nodo che collegano il nodo iniziale e il nodo finale (cioè nodo input e nodo output, nodi accessori del grafo-mashup).

localQuality(c_i)
begin
$getAccuracy(ci) = accuracy$;
$getAvailability(c_i) = availability$;
$getCompleteness(c_i) = completeness$;
$getReputation(c_i) = reputation$;
$getTimeliness(c_i) = timeliness$;
$getTagling(c_i) = tagLing$;
$getTagdf(c_i) = tagDf$;
$getTagsec(c_i) = tagSec$;
$security = EvaluateSecurity(tagSec)$;
$interoperability = EvaluateInteroperability(tagLing, tagDf)$;
return localQuality = $weightedAverage(accuracy, availability, interoperability, security, timeliness, reputation, completeness)$;
end
globalQuality(m)
begin
$mashupToGraph(m)$;
return globalQuality = $allNodesWeightedAverage(Importanza(c_i), localQuality(c_i))$;
end
Importanza(c_i)
begin
$Importanza = NumCammini(nodoIniziale, c_i) \times NumCammini(c_i, nodoFinale)$;
return Importanza;
end

4.5.4 La complessità

La complessità dell'algoritmo di qualità si condensa essenzialmente nella funzione *NumCammini*. L'analisi sul grafo si riduce essenzialmente ad una analisi in

profondità del grafo, cioè una analisi di tipo *Depth First*. Viene infatti esplorato il grafo andando, in ogni istante dell'esecuzione dell'algoritmo, il più possibile in profondità: gli archi del grafo vengono esplorati a partire dall'ultimo vertice scoperto v che abbia ancora degli archi non esplorati uscenti da esso. Una volta terminata la visita di tutti gli archi non esplorati del vertice v si ritorna indietro per esplorare tutti gli archi uscenti a partire dal vertice da cui un successore di v era stato precedentemente scoperto. Il processo di esplorazione continua fin quando tutti i vertici del grafo non siano stati esplorati.

La complessità del nostro algoritmo si risolve pertanto in una complessità del tipo $\Theta(|V| + |E|)$, dove con |E| indico il numero degli archi presenti nel grafo, mentre con |V| indico il numero dei nodi presenti nel grafo.

Capitolo 5

L'implementazione

Questo capitolo chiarisce le specifiche caratteristiche dell'architettura e di quegli strumenti fondamentali che hanno guidato l'analisi semantica più nello specifico e in generale tutto il lavoro svolto nell'ambito della generazione di reccomendations. Ci si soffermerà dapprima sul back-end dell'applicativo, cioè sull'architettura lato server e successivamente sul front-end, nel paragrafo dedicato alle prove pratiche, che chiarificherà la struttura e gli aspetti di interfaccia lato client.

5.1 L'architettura

L'architettura scelta è la comune architettura a tre livelli, che viene di buona norma utilizzata in situazioni analoghe alla nostra. Nell'architettura a tre livelli (detta anche thin client) il client non comunica direttamente con il server del database ma con un server dell'applicazione. In questo modo il client svolge solo il compito di interfaccia utente e la logica dell'applicazione viene inserita nel server applicativo. Questa soluzione è sicuramente più modulare: se si modifica la base di dati sottostante, il server dell'applicazione richiede a sua volta delle modifiche, ma l'interfaccia utente può anche restare invariata. Server applicativo e server di database possono risiedere nella stessa macchina o su macchine diverse collegate in rete. Nell'architettura a 3 livelli (detta 3-tier architecture), esiste un livello intermedio, cioè si ha generalmente un'architettura condivisa tra:

- un client, cioè il computer che richiede la risorsa, dotata di un'interfaccia utente (generalmente un navigatore web) incaricato della presentazione;

- il server d'applicazione (detto anche middleware), incaricato di fornire la risorsa ma facendo riferimento ad un altro server;
- il server di dati, che fornisce al server dell'applicazione i dati di cui ha bisogno.

Nell'architettura a 3 livelli, le applicazioni a livello del server sono de localizzate, il che significa che ogni server è specializzato in un compito (server web/server database ad esempio). L'architettura a tre livelli permette :

- un elevato livello di flessibilità;
- un elevato livello di sicurezza, dato che questa può essere definita in maniera indipendente per ogni servizio e ad ogni livello;
- delle performance soddisfacenti, dato che condivide dei compiti tra i differenti server.

Descriveremo dunque le seguenti componenti che vanno a costituire l'architettura a supporto della nostra elaborazione:

- web client;
- web server;
- application server;
- database.

In particolare all'interno del Web client si ha una interfaccia utente molto reattiva, a motivo dell'utilizzo di AJAX.

Per quanto riguarda il web server specifichiamo che in esso sono situati componenti, le ontologie e il web service, questo in modo da rendere più snella e fruibile la nostra applicazione. Inoltre questa modalità favorisce futuri aggiornamenti e manutenzioni di tutto l'apparato, anche perché in questo modo chi aggiunge nuovi componenti ha la possibilità di annotarli anche considerando le ontologie che sono già presenti sul web server. Questo evita anche di avere una esplosione di ontologie utilizzate per la annotazione, dal momento che si è stimato nel momento in cui si

sono annotati i componenti per le prove che via via che si aggiungevano ontologie, individuate col metodo descritto nel capitolo relativo all'annotazione dei componenti, alla repository del nostro dominio applicativo, che il numero di ontologie da aggiungere nel web server diminuiva quasi esponenzialmente con l'aumento dei componenti presenti. Questa diminuzione esponenziale si verifica perché, nel momento in cui l'utente aggiunge un nuovo componente e lo deve annotare, può esaminare le ontologie già presenti sul web server (tramite un apposito strumento che facilita la navigazione dell'utente all'interno delle ontologia già caricate) e con grande probabilità troverà già tra queste ontologie i vocaboli dell'area semantica specifica, che gli servono per la sua particolare annotazione del nuovo componente che sta andando a caricare.

Si parla ora della terza componente elencata, cioè l'application server. All'interno dell'application server si trovano altre parti fondamentali per il nostro algoritmo:

- il WordNet dictionary, nella sua versione più attuale. Questo permette di manutenere in maniera agevole l'applicativo e di aggiornare con facilità la piattaforma con le versioni via via più aggiornate di WordNet;
- il reasoner server (Pellet [35]), che di nuovo sarà agevolmente manutenibile;
- entity bean, (un tipo di Enterprise JavaBean, un componente J2EE lato server, che rappresenta i dati persistenti conservati in un database);
- session bean, (un tipo di Enterprise JavaBean che rende possibili operazioni, come calcoli o accessi al database);
- WorkManager, che è un elemento fondamentale per la gestione di tutto l'apparato e viene supportato dall'utilizzo dell'application server Jboss [37].

Il quarto aspetto da descrivere è il database, che presenta le seguenti caratteristiche, che sono basilari per rendere ottimale il nostro algoritmo:

- analisi di utilizzo. Nel momento in cui un utente rimpiazza un componente possiamo tenere traccia di questa azione: pertanto i dati salvati ci permettono di aiutare gli altri utenti nelle loro scelte, il che rende innovativa e decisamente *user friendly* la nostra piattaforma;

- cache similarity, che non è altro se non il salvataggio dei dati relativi alla similarità in cache in modo tale da avere sempre disponibili questi dati, concetto che già è stato accennato parlando delle ottimizzazioni apportate all'algoritmo fulcro della presente trattazione;

- gestione dei processi;

- gestione degli utenti, rendendo possibile un certo livello di personalizzazione, ad esempio mostrando allo specifico utente solo un gruppo di componenti appartenenti ad una tipologia che lui solitamente utilizza (ad esempio un creatore di mashup, che crea mashup solo relativi all'ambito del turismo).

Ribadiamo, infine, che tutta la nostra architettura ruota sull'utilizzo dell'application server (nel nostro caso Jboss). In particolare si propone la specifica architettura per l'algoritmo di similarità in figura 5.1, dove tutte le componenti tranne il web client sono inseriti all'interno della piattaforma DashMash.

Vine mostrata anche l'architettura relativa alle valutazioni di qualità (figura 5.2), dove un *Quality Manager*, che risiede sul client invia una richiesta al servizio Rest, che appoggiandosi anche a JBoss fornisce il servizio di qualità e restituisce i risultati al client tramite Json (JavaScript Object Notation è un semplice formato per lo scambio di dati).

5.2 Alcune prove pratiche

In questa sezione ci dedichiamo a valutare i principali risultati ottenuti e a commentare il loro impatto. La sezione dedicata alle prove è stata introdotta anche per far comprendere più agevolmente come si presenteranno e risultati all'utente finale e e come egli potrà sfruttarli in modo ottimale. Verranno riportate i risultati di sperimentazione nella forma tabulare che si presenterà nell'interfaccia lato client. In questo modo non solo ci si è dedicati a descrivere il back-end lato server, ma si darà una visione d'insieme globale e più completa.

Per prima cosa diamo una descrizione dell'insieme di componenti che abbiamo utilizzato per le nostre prove:

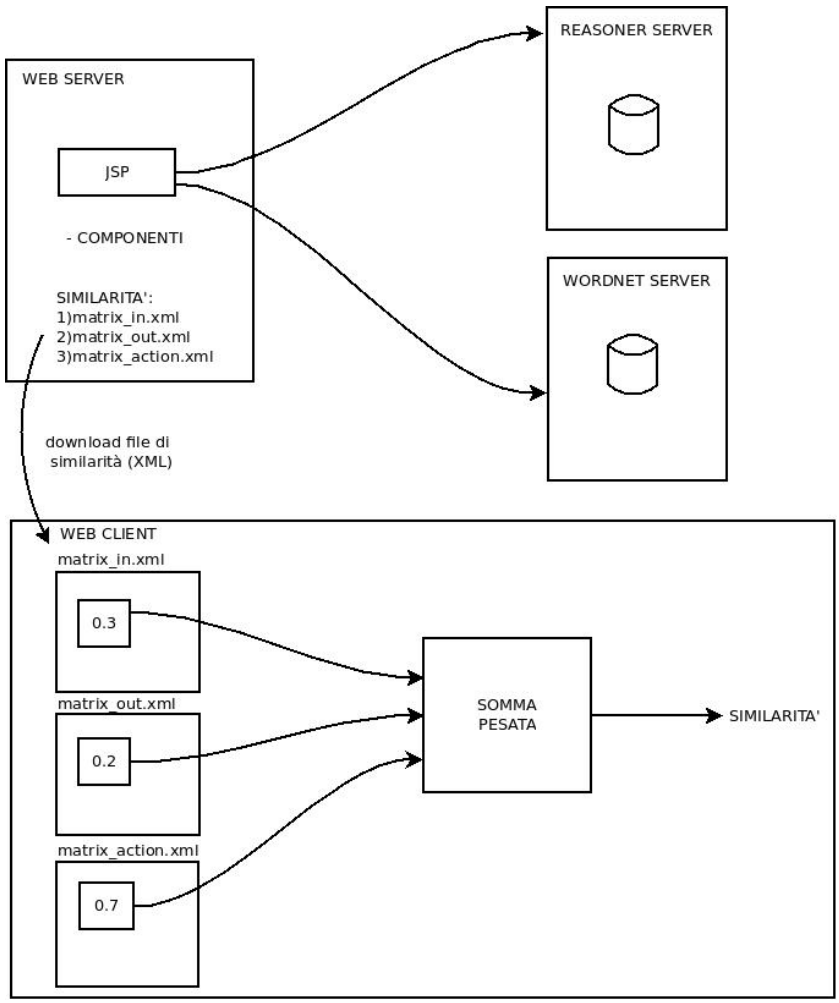

Figura 5.1: La distribuzione del calcolo della similarità nei componenti architetturali

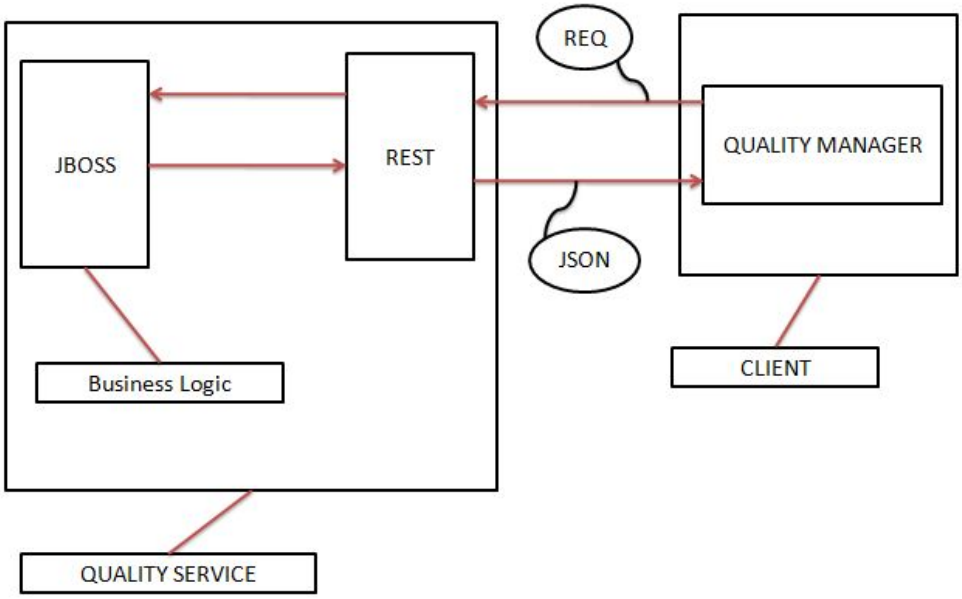

Figura 5.2: La distribuzione del calcolo della qualità nei componenti architetturali

- *FindPos*: è un componente che a fronte di coordinate di latitudine e longitudine in input, restituisce in uscita un luogo geografico;

- *SearchPos*: analogamente a FindPos, è un componente che a fronte di coordinate di latitudine e longitudine in input, restituisce in uscita un luogo geografico;

- *CaloriesShow*: dato un percorso tra due luoghi espressi in termini di longitudine e latitudine e la velocità dello spostamento, restituisce in uscita le calorie bruciate;

- *CaloriesService*: mostra graficamente il livello di calorie bruciate a fronte dell'aggiornamento di un indicatore delle calorie;

- *GeoNames*: è un componente che a fronte di coordinate di latitudine e longitudine in entrata, restituisce come output un indirizzo di un luogo (in termini di città e via);

- *Gmap*: è un componente che a fronte di coordinate di latitudine e longitudine, unitamente ad un livello di zoom in ingresso, mostra in output una mappa geografica.

Proponiamo quindi delle tabelle che raffigurino delle matrici di compatibilità seriale e parallela ottenute simulando il nostro algoritmo su un gruppo di componenti (tabelle 5.1 e 5.2).

	FindPos	SearchPos	GeoNames	CaloriesService	CaloriesShow	Gmap
FindPos	1	1	1	0	0	1
SearchPos	1	1	1	0	0	1
GeoNames	1	1	1	0	0	1
CaloriesService	0	0	0	1	0	0
CaloriesShow	0	0	0	0	1	0
Gmap	1	1	1	0	0	1

Tabella 5.1: Matrice di compatibilità parallela simulata

	FindPos	SearchPos	GeoNames	CaloriesService	CaloriesShow	Gmap
FindPos	0	0	0	1	1	0
SearchPos	0	0	0	1	1	0
GeoNames	0	0	0	1	1	0
CaloriesService	1	1	1	0	1	1
CaloriesShow	1	1	1	1	0	1
Gmap	0	0	0	1	1	0

Tabella 5.2: Matrice di compatibilità seriale simulata

Si nota subito che le matrici sono sparse (e il grado di sparsità aumenta con il numero di componenti presenti) e simmetriche. Per quanto riguarda questa prova sulla compatibilità ci riteniamo molto soddisfatti, dal momento che si trova una effettiva rispondenza dell'algoritmo implementato alle aspettative che ci si era posti. Il confronto sintattico sui tipi è quindi riuscito. Riteniamo opportuno precisare che questo è solo un estratto minimale di tutte le prove che sono state fatte e che in generale tutte le altre prove hanno fornito risultati altrettanto validi, anche su gruppi più numerosi di componenti.

Proseguendo nella valutazione pratica del nostro algoritmo passiamo ora ad analizzare la matrice di similarità ottenuta sia col metodo di similarità verticale,

sia col metodo di similarità orizzontale, sempre sul medesimo range di componenti (tabelle 5.3 e 5.4).

	FindPos	SearchPos	GeoNames	CaloriesService	CaloriesShow	Gmap
FindPos	1	0.923	0.457	0	0	0.516
SearchPos	0.923	1	0.521	0	0	0.608
GeoNames	0.457	0.521	1	0	0	0.712
CaloriesService	0	0	0	1	0	0
CaloriesShow	0	0	0	0	1	0
Gmap	0.516	0.608	0.712	0	0	1

Tabella 5.3: Matrice di similarità verticale simulata

	FindPos	SearchPos	GeoNames	CaloriesService	CaloriesShow	Gmap
FindPos	0	0	0	0.432	0.678	0
SearchPos	0	0	0	0.498	0.636	0
GeoNames	0	0	0	0.516	0.534	0
CaloriesService	0.432	0.498	0.516	0	0.879	0.328
CaloriesShow	0.678	0.636	0.534	0.879	0	0.327
Gmap	0	0	0	0.328	0.327	0

Tabella 5.4: Matrice di similarità orizzontale simulata

I valori ottenuti nelle matrici di similarità sono soddisfacenti in merito a quello che si era previsto che accadesse. I valori registrati sono stati ottenuti aggregando i singoli valori come descritto nello pseudocodice dell'algoritmo di similarità. Di nuovo troviamo matrici simmetriche, che ricalcano esattamente quanto ottenuto con le matrici di compatibilità, con la differenza che in questo caso al posto di un *generico* 1, abbiamo dei coefficienti accuratamente calibrati, come ampiamente descritto nela sezione dedicata all'algoritmo di similarità. Chiaramente i componenti confrontati con sé stessi avranno coefficiente di similarità unitario nelle matrice di similarità. Le sperimentazioni sono state molteplici e tutte hanno condotto a risultati gratificanti, parimenti a quelli mostrati nelle tabelle 5.3 e 5.4. L'utente tramite l'interfaccia grafica potrà visionare anche i dati disaggregati e potrà variare i pesi descritti nell'algoritmo (chiaramente solo quelli variabili dall'utente tramite menù a tendina) a suo piacimento e a seconda delle sue specifiche necessità.

A questo punto non resta che proporre i risultati che si sono ottenuti tramite l'algoritmo per il calcolo della qualità. Prendiamo, ad esempio, i componenti già

utilizzati per le altre sperimentazioni riportate. A fronte delle annotazioni interne dei componenti (ovviamente relative alla qualità), si sono ottenuti i valori di qualità riportati in tabella 5.5.

Componenti	FindPos	SearchPos	GeoNames	CaloriesService	CaloriesShow	Gmap
Qualità	0.5562	0.6437	0.7687	0.6125	0.6001	0.989

Tabella 5.5: Valori simulati di qualità dei singoli componenti

Notiamo subito che questo dato di qualità ci permette di avere ulteriori informazioni in merito ai componenti e ci fornisce un ulteriore spunto di scelta ai fini della componentizzazione. Si osserva infatti che due componenti molto simili come FindPos e SearchPos, hanno differenti valori di qualità: in particolare SearchPos ha qualità maggiore rispetto a FindPos. Ovviamente oltre al dato di qualità l'utente può osservare i dati disaggregati, che permettono di valutare le caratteristiche qualitative singole. Nuovamente possiamo asserire che l'algoritmo si comporta nel modo sperato, fornendo valori ragionevoli. Ancora diciamo i dati riportati sono un sunto delle sperimentazioni fatte. Per concludere la sezione proponiamo un dato sulla qualità di un mashup. Si sono utilizzati cinque dei sei componenti descritti sopra, collegandoli tra loro in modo da formare un grafo con cinque nodi pari al numero dei componenti e gli archi orientati corrispondenti ai binding, come mostrato in figura 5.3. Indubbiamente ci sono molteplici combinazioni possibili con questi cinque e/o con altri componenti, ma qui riportiamo un grafo o mashup di prova, a titolo esemplificativo.

Procedendo secondo il metodo dell'importanza si è ottenuta una qualità globale del grafo, o del mashup (analogamente), pari a 0.6380. Ricordiamo che l'algoritmo mette a disposizione dell'utente anche una funzionalità che permette di calcolare il dato disaggregato di qualità per il mashup (ad esempio solo la reputazione o solo la accuratezza e così via).

Concludiamo questa sezione dichiarandoci soddisfatti dei risultati ottenuti. Non si esclude, comunque, che vi possano essere ulteriori ottimizzazioni, ma per questo rimandiamo alla sezione relativa agli eventuali sviluppi futuri.

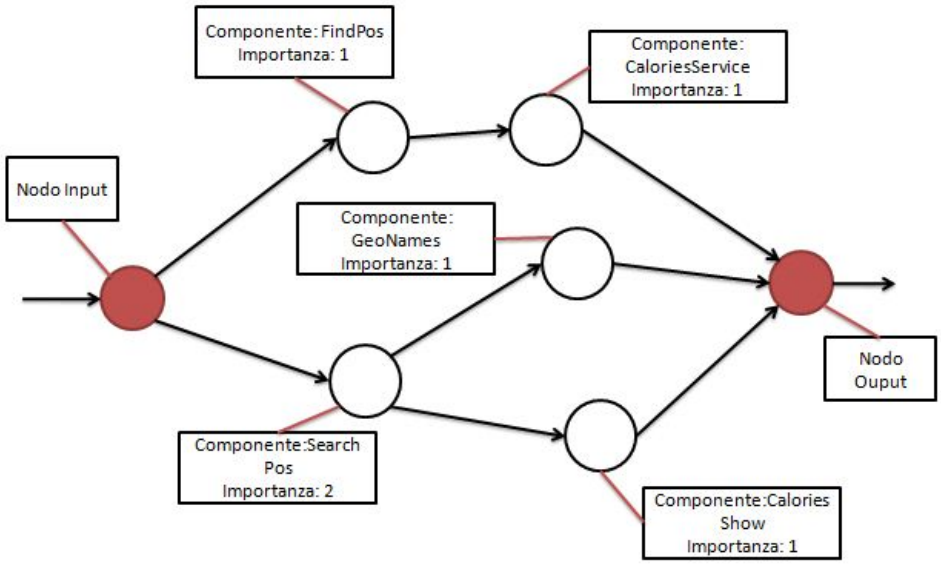

Figura 5.3: Il grafo per il calcolo della qualità globale del mashup

5.3 Integrazione degli algoritmi nel front-end

Questa sezione spiega come gli algoritmi descritti si integrino all'interno del front-end, cioè come si presenterà l'interfaccia utente. In questo modo risulterà definitivamente chiaro come il sistema di raccomandazione generato dai nostri algoritmi possa aiutare l'utente finale nell'operazione di composizione di un mashup. Innanzitutto come si può vedere in figura 5.4 l'interfaccia si può dividere in due zone: sulla sinistra abbiamo una zona più estesa, che contiene un browser che contiene l'elenco degli workspace e per ogni workspace l'elenco dei componenti contenuti in esso, mentre sulla destra abbiamo un'area denominata *See Also*, nella quale vengono visualizzate di volta in volta le informazioni di dettaglio, che l'utente sceglie di vedere navigando negli workspace.

Ogni workspace viene contrasegnato dal suo valore in termini di qualità tramite un numero di stelline che va da 1 a 5. Selezionando lo workspace a cui si è interessati, nella parte del *See Also* verranno visualizzate tutte le composizioni simili allo workspace di partenza e i relativi dati di qualità.

Analogamente per i componenti che sono contenuti in ciascuno workspace sono contrassegnati da un numero di stelline che va da 1 a 5 e che indica i livelli di qualità del componente. Di nuovo selezionando lo specifico componente l'utente finale potrà visualizzare nella sezione di schermo *See Also* l'elenco di tutti i componenti simili a quello selezionato, ognuno corredato dai corrispondenti dati di qualità.

Come si può constatare osservando figura 5.4, l'interfaccia utente proposta è abbastanza intuitiva e si presta all'utilizzo da parte di utenti finali anche inesperti.

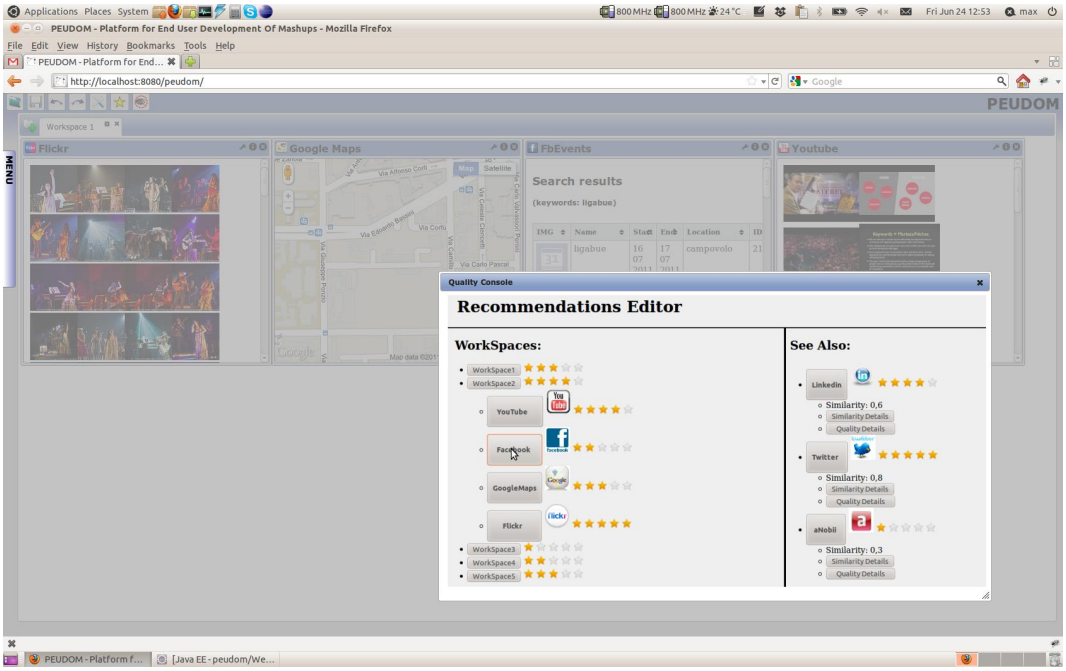

Figura 5.4: L'interfaccia a supporto delle recommendations

Nell'esempio mostrato si sta navigando all'interno del Workspace2. Come si nota l'utente ha quattro componenti, di cui uno, il componente Facebook ha qualità inferiore rispetto alle sue aspettative. Selezionando il componente nella sezione *See Also* compaiono tre componenti simili, con il dato aggregato di similarità e l'indicatore di qualità. A sua discrezione l'utente potrà visualizzare i dettagli (cioè i dati disaggregati) relativi a similarità e qualità, selezionando con il mouse rispettivamente il pulsante *Similarity details* o il pulsante *Quality details*.

Risulta chiaro come l'interfaccia e gli algoritmi ad essa sottesi possano fattivamente coadiuvare l'utente finale nella creazione del suo mashup tramite l'utilizzo

di recommendations.

Capitolo 6

Portabilità del metodo

Per avvalorare ulteriormente il presente lavoro, non si può prescindere dal menzionare la sua portabilità. Gli argomenti di questa tesi infatti si prestano ad essere ripresi e ampiamente utilizzati anche in altri ambiti di interesse sempre nel campo del Web semantico. Si può infatti proporre un metodo di utilizzo dell'algoritmo enunciato per supportare il Search Computing [44], soprattutto nell'ambito specifico delle liquid queries. Occorre dunque definire in breve cosa sia una liquid query per poter meglio comprendere come il metodo sia portabile. Nella figura 6.1 mostriamo un primo esempio d'impatto visivo.

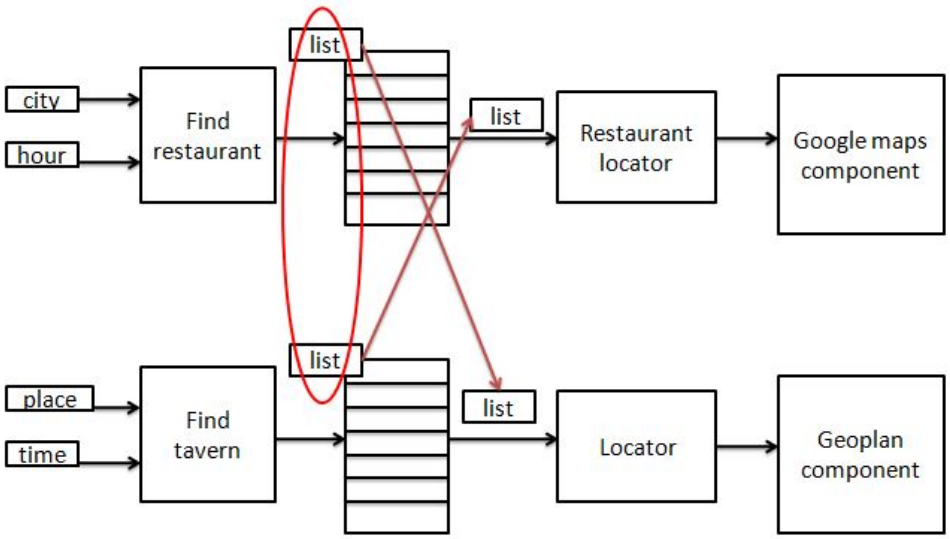

Figura 6.1: Un esempio intuitivo di un possibile scenario

Le liquid queries si concentrano sugli aspetti di front-end del Search Computing: si hanno un ciclo di vita delle query, query di alto livello, un framework lato client per la configurazione e per il rendering automatico dell'interfaccia delle query e dei risultati e infine un set di primitive d'interazione utilizzabili dall'utente, che gli permettano di attuare la ricerca esplorativa. Il paradigma delle liquid queries prevede l'utilizzo di un tool per la manipolazione flessibile dei dati per condurre la ricerca delle informazioni, basato sull'esplorazione progressiva dello spazio di ricerca. Definiamo dunque una liquid query come un insieme di interfacce di servizio, un insieme di schemi di connessione per raggiungere tutte le interfacce di servizio coinvolte, un insieme di predicati aggiuntivi di selezione o di join, una funzione di ranking automatica di default definita sui punteggi delle interfacce di servizio, infine un insieme di possibili raggruppamenti, ordinamenti e clustering di attributi, che possono applicati sugli insiemi dei risultati estratti. Gli attori di questo sistema sono l'Application designer e l'utente finale. Il primo si occupa di definire un template per le liquid queries, mentre il secondo sottomette le liquid queries e poi manipola i liquid results.

È a questo punto che si inserisce l'algoritmo esposto nei precedenti paragrafi: mediante questo algoritmo infatti diviene possibile fornire metodologie aggiuntive all'Application designer per arricchire e rendere sempre più preciso e variegato il template per le liquid queries. Tramite un insieme di componenti mashup sarà infatti possibile valutare le query complesse dei più svariati generi, proponendo anche una vasta gamma di componenti simili tra loro e compatibili con determinate specifiche di interesse per l'utente.

Il compito dell'Application designer si renderà dunque più agevole, avendo quest'ultimo a disposizione un nutrito gruppo di componenti da scegliere ad hoc anche in merito ai dati di qualità, che sono cruciali per la scelta di contenuti per le peculiarità, descritte nei precedenti paragrafi, relative a questi dati. I vantaggi non saranno però solo per l'Application designer, ma anche per gli utenti finali come poc'anzi accennato. L'utente avrà a disposizione una ampia scelta di servizi analoghi e potrà valutare i dati di ogni singolo componente che andrà a far parte della sua ricerca complessa. Si potrà quindi fornire all'utente finale non solo un valore della qualità globale, ma anche un valore locale della qualità di ciascun componente coinvolto nel mashup. Come già accennato nei precedenti paragrafi

l'utente potrà scegliere il componente o l'intero mashup che più conviene alle sue esigenze anche valutando il parere espresso da altri utenti non solo sul singolo componente, ma anche su interi mashup già componentizzati. In parole povere l'utente si troverà di fronte a un ricco range di componenti, tra i quali potrà scegliere il componente per lui migliore per rispondere alla sua query.

Per fare un esempio potremmo prendere una query complessa di questo tipo: "dove posso trovare un ristorante di cucina messicana a Milano nei pressi di un cinema che trasmetta solo film di arti marziali fino alle 3 di notte?". A fronte di questa richiesta ci possono essere molteplici mashup che filtrino questo tipo di informazioni, che ricevano cioè questi input e a fronte di determinate operazioni o eventi, restituiscano output compatibili con la ricerca che l'utente sta effettuando. Dovranno dunque essere selezionati tutti quei mashup che abbiano componenti relativi alla ricerca di ristoranti o trattorie e che filtrino quelli che propongono cucina messicana, alla ricerca di cinema che evidenziano il determinato genere dei film trasmessi e che valutino se tra questi generi sia presente anche arti marziali, fornendo inoltre gli orari di apertura di questi cinema (figure 6.2 e 6.3). Questa ricerca di componenti e di mashup si appoggerà a WordNet ed alle ontologie e opererà secondo l'algoritmo di cui si è già diffusamente parlato.

Si possono avere mashup più o meno ricchi, più o meno precisi, più o meno veloci e più o meno costosi, ma soprattutto ciascun mashup che è nel gruppo di quelli compatibili, con la ricerca sarà classificato e ordinato rispetto al suo livello di similarità e al suo livello di qualità. Infine la qualità sarà corredata come già accennato, dai fondamentali dati relativi alla soddisfazione espressa dagli utenti che hanno già usufruito di quel mashup. Tramite tutti questi dati l'utente potrà scegliere come condurre la sua ricerca, avendo a disposizione molte più possibilità e molte più direzioni verso cui condurre la propria ricerca (figura 6.4).

Risulta quindi chiara l'elevata portabilità e fruibilità del metodo oggetto della presente tesi. Come già accennato si auspica non solo che si possa adottare fattivamente questo metodo nell'ambito del Search Computing, ma che lo si possa *portare* anche in altri ambiti, sempre relativi all'analisi sintattica, semantica e qualitativa.

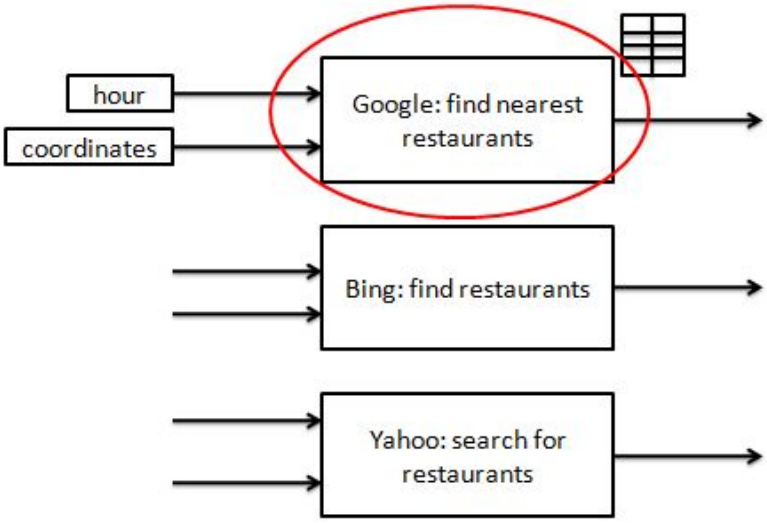

Figura 6.2: Raccolta di tutti i mashup esistenti specifici per una certa ricerca

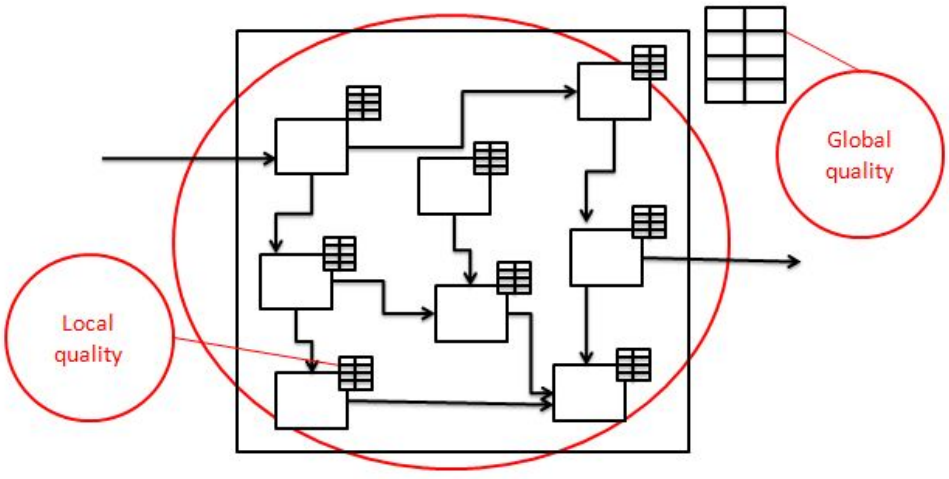

Figura 6.3: Struttura interna di un mashup

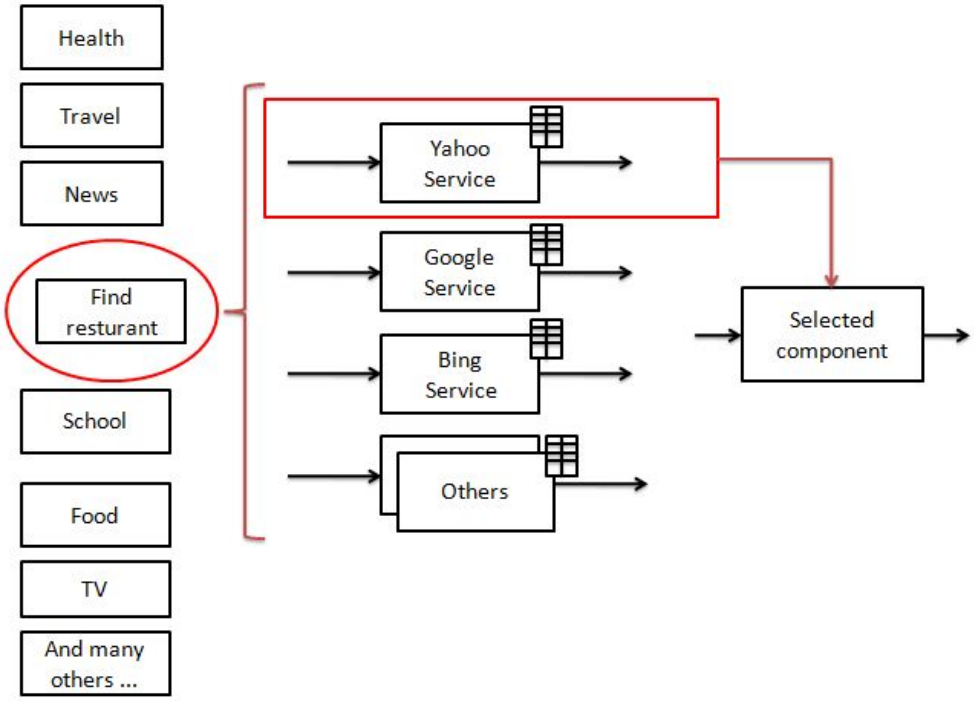

Figura 6.4: La scelta del componente

Capitolo 7

Conclusioni e sviluppi futuri

In questa parte della tesi ci proponiamo di trarre alcune conclusioni relative al lavoro svolto e di concentrarci su quelle prospettive innovative che si aprono a fronte dello studio fatto.

7.1 Argomentazioni a supporto dell'utilizzo delle ontologie

Considerando l'ambito degli sviluppi futuri, si deve sicuramente menzionare il fatto che in un futuro più o meno recente potrà verificarsi l'eventualità, già annunciata nei precedenti paragrafi, che si venga a creare una repository globale di ontologie, che ci renda possibile avere un unico luogo virtuale dove trovare ogni tipo di ontologia ci sia di volta in volta necessaria per annotare e valutare i vari componenti. Indubbiamente il problema è già stato individuato e molti stanno cercando di risolverlo facendo in modo di aggirare l'enorme disguido dell'assenza di un organo di raccolta globale e manutenzione per le ontologie.

In un passato non troppo distante il progetto *WONDERWEB* [39] ufficialmente terminato nel luglio 2004, si proponeva di raccogliere tutte le ontologie esistenti e di dare loro una strutturazione univoca e fondata su determinati criteri. Purtroppo, a causa di mancanza di fondi il progetto terminò senza concludere l'enorme lavoro che aveva cominciato. Questo impegno potrà in un futuro non troppo lontano essere ripreso e portato finalmente a termine. Viene da chiedersi perché ci sia un

tale sforzo nel pontificare le qualità delle ontologie. La verità è che l'approccio ontologico è veramente imprescindibile ai nostri fini. Se si vuole comprendere a fondo l'importanza di quanto affermato poco sopra, per prima cosa è bene valutare il punto di arrivo del nostro studio, che si serve dell'approccio ontologico in maniera massiva. Gli obiettivi che si propone di raggiungere questo paragrafo sono essenzialmente due.

Un primo obiettivo è quello far comprendere come le ontologie fondazionali abbiano la capacità di rendere sempre più fruibile da parte dell'utente l'applicativo che si va a realizzare; si procede dunque ad ottimizzare sempre di più il servizio reso all'utente finale, incrementando il livello di interazione e rendendo sempre più compenetrati i ruoli dei tre fondamentali attori che riconosciamo nel sistema, cioè quelle figure che hanno accesso e che operano sulla piattaforma *Dash Mash*, cioè amministratore, utente esperto e utente finale.

L'idea che si persegue è quella di arrivare a colmare sempre di più le distanze tra i tre ruoli individuati, facendo in modo che l'utente inesperto abbia la possibilità di adempiere il maggior numero di compiti possibili, senza avere un bagaglio di conoscenze troppo corposo. In pratica si tenderà a rendere sempre più ridotto il dislivello in termini di conoscenze tra questi tre attori, in particolare nell'ottica principale di coadiuvare maggiormente l'utente finale. L'idea di fondo è quella di guidare passo passo l'utente nello sviluppo e soprattutto nell'annotazione del componente, facendo in modo che si attui un processo di *learning* nel momento in cui l'utente esplora la piattaforma e la arricchisce con i suoi componenti. Si vuole in questo modo ottenere il coinvolgimento dell'utente finale all'interno del processo di creazione di contenuti propri e di sviluppo di nuove applicazioni mashup.

Un secondo obiettivo si basa sulla convinzione che la *macchina* deve adattarsi all'utente e non il viceversa, dove per macchina si intende l'applicazione con la quale l'utente (finale o esperto che sia) viene ad interagire, nello specifico di questa trattazione la piattaforma *Dash Mash*. Per questo motivo viene privilegiato un approccio che ponga il suo fondamento nel linguaggio naturale in termini non solo di lessico, ma anche di semantica. Considerando unitamente questi due concetti viene a delinearsi chiaramente l'importanza delle ontologie per condurre l'analisi semantica del linguaggio naturale e per creare un ponte tra la macchina e l'utente, con il proponimento di favorire quest'ultimo. Risulta quindi evidente a questo

punto il motivo per cui le ontologie sono così importanti e insostituibili nell'ambito della presente tesi.

I principali aspetti negativi che riemergono ogniqualvolta si parla di ontologie sono essenzialmente due: il primo riguarda la scarsa reperibilità di ontologie, mentre il secondo riguarda la loro difformità, il differente livello di dettagli a seconda delle diverse ontologie, il loro scarso grado di interrelazione con le più importanti ontologie *top level*, il loro essere spesso create *ad hoc*, per determinati ambiti e per specifici scopi applicativi. Tuttavia alcuni progetti correnti, ad esempio YAGO [1], tendono a creare basi di conoscenza sempre più ampie, in grado di integrare in modo omogeneo sorgenti ontologiche diverse.

Ontologie fondazionali ben strutturate e rigorose possono ovviare ai problemi evidenziati poco prima, in primo luogo perché un insieme di ontologie di tipo *top level*, sviluppate in modo univoco potrebbero essere raccolte in una repository universale e ogni utente potrebbe caricarne di nuove a patto che seguano certi parametri realizzativi e che non collidano con ontologie preesistenti (a meno che non siano evidentemente migliori), in secondo luogo perché ontologie fondazionali con elevato livello di correlazione tra concetti, porterebbero ad un facile aggancio da parte di ontologie *middle level* (ontologie di collegamento tra quelle fondazionali e quelle specifiche) o direttamente da parte di ontologie riguardanti specifici domini applicativi. Questo secondo punto evidenzia come si potrebbe agevolmente arginare il problema legato al differente livello di dettaglio e soprattutto eliminare gli sgradevoli inconvenienti derivanti dallo scarso grado di correlazione e interrelazione tra ontologie poc'anzi descritto.

Notoriamente, le ontologie fondazionali permettono *interoperabilità semantica*, che a sua volta da la possibilità di mantenere intatti i contenuti ed i significati nel passaggio delle informazioni. Tutto ciò si ottiene tramite la costruzione di un modello ontologico e la elaborazione di un modello integrato. La strutturazione di un solido modello ontologico permette di avere una metodologia standard ed univocamente definite, ma per fare tutto ciò si rendono necessari alcuni passaggi

[1] YAGO è un'ampia base di conoscenza semantica derivata da Wikipedia Wordnet e GeoNames, che al momento rappresenta più di 10 milioni di entità (ad esempio persone, organizzazioni, città, ecc.) e contiene più di 80 milioni di fatti riguardo tali entità. Per maggiori informazioni si visiti il sito *www.mpi-inf.mpg.de/yago-naga/yago/*.

fondamentali:

- comprendere il genere di legami intercorrenti tra i servizi;
- indagare le relazioni preesistenti tra servizi interconnessi.

In particolare nella presente tesi, si adotta un approccio di tipo fondazionale [9, 10], cioè l'utilizzo di ontologie fondazionali o ontologie *top level*. Queste sono strutturate in maniera ottimale, dal momento che prima partono da una solida analisi ontologica e solo successivamente traducono l'analisi in un linguaggio di markup come *XML* o *OWL*, dal momento che questi non sono altro che linguaggi e non concettualizzazioni logiche. Chiaramente una ontologia in generale risulta essere utile quando presenta delle particolari caratteristiche, che costituiscono fattori di categorizzazione molto importanti:

- si devono riconoscere con accuratezza anche le distinzioni di significato più sottili anche in termini generali, come verrà esplicato nella sezione relativa agli esempi, in modo da rendere chiara la necessità di questo vincolo;
- la comprensione reciproca risulta importante anche a scapito dell'interoperabilità: i componenti devo comprendersi tra loro, così come gli utenti, che devono certamente comprendersi tra di loro, ma anche comprendere esattamente i contenuti che la piattaforma e l'algoritmo espongono loro;
- anche situazioni di disaccordo vengono notificate, in modo tale da migliorare eventuali comportamenti del sistema in risposta ad alcune specifiche richieste.

L'utilizzo di dette ontologie deve inoltre essere specificatamente giustificato e argomentato, nel particolare compito che si sta affrontando, in primo luogo per evitare ogni tipo di obiezione e di atteggiamento scettico e in secondo luogo per comprovare in modo più generale la necessità di un approccio di tipo ontologico e in particolare fondazionale.

In maniera propositiva si auspica quindi, che questa strada possa essere seguita e coltivata anche al fine di rendere più fruibili agli utenti la piattaforma *Dash Mash* e gli algoritmi che sono il fulcro della presente tesi. Inoltre, questa metodologia

potrebbe proficuamente essere applicata ad altre piattaforme e ad altri servizi, che si appoggiano all'utilizzo delle ontologie, con la prospettiva di esportare un'idea positiva come la facilitazione dei compiti degli utenti e in particolare di quelli che in precedenza abbiamo definito utenti finali o inesperti.

7.2 Sviluppi futuri

Nell'ambito della qualità vi sono una moltitudine di possibili sviluppi in differenti direzioni, sia in termini di ottimizzazione, sia in termini di nuove implementazioni e studi relativi a metriche aggiuntive. Nell'odierna era informatica, la qualità è un tema di importanza consolidata; nel contempo i mashup sono oggetto di un interesse sempre crescente. È quindi evidente la necessità di effettuare studi di qualità su questo tipo di applicazioni, in modo da identificare quanto prima gli aspetti significativi, caratteristici di buone composizioni. Questo lavoro di tesi ha mostrato come la qualità possa guidare la composizione e come la percezione del mashup finale da parte dell'utente sia condizionata dalla qualità stessa. Innanzitutto è stato sviluppato un modello che identifica le caratteristiche di qualità rilevanti per i mashup. I concetti fondamentali e innovativi introdotti in questo modello sono la qualità del componente e la qualità finale. Grazie ad essi è stato possibile da un lato massimizzare la qualità del mashup individuando i componenti migliori per il particolare mashup che si sta componendo, dall'altro modificare la qualità in base al ruolo del componente all'interno del mashup stesso. Gli aspetti più teorici di definizione del modello di qualità hanno avuto naturale conseguenza nell'implementazione di un modulo dell'editor di DashMash che è flessibile, in quanto pensato per poter processare qualsiasi tipo di componente, e completo, dato che le sue funzionalità spaziano dal calcolo della qualità, alla raccomandazione dei componenti. Tramite l'editor esteso è stato possibile mostrare come la qualità cambi sia il processo di sviluppo di un mashup che il risultato finale, aumentando la qualità del processo di sviluppo e del mashup finale, dal punto di vista quindi sia del compositore che dell'utente. Il processo di valutazione di compatibilità e similarità, alla base delle raccomandazioni e dei suggerimenti, copre sia la sfera sintattica sia quella semantica. Inoltre il processo di valutazione di questi due concetti è definito per un livello di dettaglio molto alto: per ogni operazione

dei componenti disponibili nella repository dell'editor di composizione. Possiamo quindi dire che questo lavoro di tesi ha mostrato che la qualità è un parametro discriminante per la composizione di un buon mashup e che un ambiente di sviluppo basato su di essa può dare maggiore rapidità al processo di composizione. Questo lavoro di tesi, nonostante sia completo nel trattare i concetti di qualità, lascia ancora spazio a lavori di sviluppo che possano ancor di più dimostrare la validità della composizione di mashup guidata dalla qualità. Verranno analizzati i punti aperti a ulteriori studi qui di seguito:

- Il modello di componente utilizzato considera buona parte degli attributi come delle black box che si assume vengano correttamente valorizzate. Quindi si potrebbero sviluppare dei procedimenti automatici, magari supportati da un ambiente in cui è possibile testare il componente in modo automatizzato, per la valutazione degli stessi. Inoltre una carenza importante è per gli attributi classificati come qualità percepite, che valutano la sfera estetica del mashup; essi, nonostante siano valorizzati all'interno del componente, non sono studiati tanto a fondo da poter prescindere da tecniche di ispezione da parte di valutatori, introducendo una forte componente soggettiva. Una sviluppo futuro può sicuramente riguardare la ricerca di particolari aspetti appartenenti a questa classe che possano essere valutati senza l'intervento umano. Oltre a queste carenze, dato che i mashup sono una tecnologia emergente, l'utilizzo sempre più frequente di queste applicazioni può portare a considerare nuovi aspetti non colti da questo modello;

- tutte le metriche di qualità vengono associate al componente singolo e pesate in relazione al mashup già realizzato; in un futuro si potrà pensare di dare a questi componenti un valore dinamico di qualità che evolva a livello di *composition time*, a seconda di come procede la componentizzazione;

- sempre sulla scia del punto precedente si potrebbe pensare di valutare la qualità a seconda di tutte le possibili sostituzioni di componenti simili, in modo che l'utente possa vedere in tempo reale gli sviluppi possibili per il suo mashup, cioè che abbia la possibilità di valutare non solo tra generici componenti simili e le relative qualità, ma che mentre compone il suo mashup l'editor gli proponga scelte simili e migliori per la qualità globale del mashup.

Tutte queste possibilità comportano indubbiamente elevati carichi di lavoro, ma se realizzate permetterebbero di ottenere dei tool con una carica innovativa considerevole soprattutto per quanto riguarda l'impegno posto a favorire l'utente finale. Il lavoro svolto in questa tesi rappresenta un passo verso la possibilità di offrire ad utenti non esperti degli strumenti informatici di un certo livello, un ambiente per la costruzione di applicazioni di alta qualità. Compatibilità, similarità e qualità sono tre parti inscindibili di un *unicum*, che fornisce una metodologia ottimale per l'utente per la strutturazione di mashup. Questa strada pertanto merita di essere studiata e coltivata in modo da giungere ad ulteriori miglioramenti, dei quali quelli proposti poco sopra costituiscono una rampa di lancio per trovarne moltissimi altri.

Appendice A

Tecnologie adottate

In questa appendice ci dedichiamo a descrivere più dettagliatamente alcune tecnologie che hanno supportato lo sviluppo del nostro algoritmo e che sono funzionali alla generazione di raccomandazioni.

A.1 WordNet

WordNet è un database semantico-lessicale per la lingua inglese elaborato dal linguista George Armitage Miller presso l'Università di Princeton, che si propone di organizzare, definire e descrivere i concetti espressi dai vocaboli. L'organizzazione del lessico si avvale di raggruppamenti di termini con significato affine, chiamati synset (dalla contrazione di synonym set), e del collegamento dei loro significati attraverso diversi tipi di relazioni chiaramente definite. All'interno dei synset le differenze di significato sono numerate e definite. Le relazioni semantiche variano in funzione del tipo di parola e includono le seguenti categorizzazioni. Per i sostantivi si hanno: iperonimia (hyperonyms): Y è un iperonimo di X se ogni X è una specie di Y; iponimia (hyponyms): Y è un iponimo di X se ogni Y è (una specie di) X; coordinazione: Y è un termine coordinato di X se X e Y hanno un iperonimo in comune; olonimia (holonym): Y è un olonimo di X se X è parte Y; meronimia (meronym): Y è un meronimo di X se Y è parte X; per i verbi si hanno: iperonimia (hypernyms): il verbo Y è un iperonimo del verbo X se l'attività X è (una specie di) Y (come viaggio rispetto a movimento); troponimia (troponyms): il verbo Y è un troponimo del verbo X se nel fare l'attività Y si

fa anche la X (come mormorare rispetto a parlare); implicazione (entailment): il verbo Y è un'implicazione del verbo X se nel fare X uno deve per forza fare Y (come russare rispetto a dormire); coordinazione: Y è un termine coordinato di X se X e Y hanno un iperonimo in comune; gli aggettivi sono classificati come: nomi relativi, simile a, participi dei verbi; gli avverbi seguono la classificazione dell'aggettivo da cui derivano. Per dare ulteriori dettagli forniamo la specifica versione di WordNet utilizzata nelle sperimentazioni, cioè WordNet 3.0, facilmente reperibile e scaricabile. Unitamente a WordNet ci si è dovuti appoggiare anche alla libreria *jaws*, che è la libreria che permette di accedere al vocabolario WordNet dal codice Java.

A.2 Le ontologie

In informatica, un'ontologia è una rappresentazione formale, condivisa ed esplicita di una concettualizzazione di un dominio di interesse. Più nel dettaglio, si tratta di una teoria assiomatica del primo ordine esprimibile in una logica descrittiva. Il termine ontologia (formale) è entrato in uso nel campo dell'intelligenza artificiale e della rappresentazione della conoscenza, per descrivere il modo in cui diversi schemi vengono combinati in una struttura dati contenente tutte le entità rilevanti e le loro relazioni in un dominio. I programmi informatici possono poi usare l'ontologia per una varietà di scopi, tra cui il ragionamento induttivo, la classificazione, e svariate tecniche per la risoluzione di problemi. Le ontologie sono caratterizzate dai linguaggi OWL e RDF e presentano i seguenti operatori principali: subClassOf, equivalentClass, inverseOf, disjointWith e, eventualmente, se ne possono trovare altri in altri contesti. Un'ontologia può essere rappresentata anche come una struttura ad albero dove ogni nodo è sottoclasse diretta o indiretta dei nodi che lo precedono. Inoltre bisogna ricordare che le ontologie seguono una ben precisa gerarchia che si può descrivere dicendo che vi è un radice costituita dall'onotlogia SUMO.owl (Suggested Upper Merged Ontology), dalla quale discendono tutte le altre. Le ontologie che discendono direttamente dall'ontologia SUMO, vengono dette middle-level e sono quelle ontologie che appunto costituiscono da ponte tra l'onotlogia SUMO e i livelli successivi. Mano a mano che si scende nella gerarchia si trovano tutte quelle ontologie che sono correlate ad una

specifica area semantica: più si scende verso il basso in questo ipotetico albero più si incontrano aree semantiche più dettagliate. Vorremmo però sottolineare un aspetto degradante per le ontologie, cioè che esse non sono ancora state raccolte o catalogate da nessun ente preposto alla loro conservazione e manutenzione. Questo fatto le rende poco gestibili ma soprattutto non permette di utilizzare a pieno le loro potenzialità. Si auspica che in futuro ci sia modo di utilizzare una tecnologia analoga, ma che non utilizzi le ontologie, per gli scopi qui esposti o che alternativamente vi sia un ente che le raccolga in modo efficiente e in modo da renderle sempre più reperibili e sempre aggiornate.

A.2.1 Il reasoner semantico

Un reasoner semantico, motore di ragionamento, motore di regole, o semplicemente un reasoner, è un pezzo di software in grado di dedurre conseguenze logiche da un insieme di fatti, affermazioni o assiomi. La nozione di reasoner semantico generalizza quella di motore di inferenza, fornendo un insieme più ricco di meccanismi con cui lavorare. Le regole di inferenza vengono comunemente indicate per mezzo di un linguaggio ontologico, e spesso un linguaggio di descrizione. Molti reasoner usano logica dei predicati del primo ordine per eseguire un ragionamento; l'inferenza procede normalmente tramite concatenamento in avanti e all' indietro. Ci sono anche esempi di reasoner probabilistici, che risultano però meno sviluppati, meno conosciuti, ma soprattutto meno efficienti. In particolare in questa tesi si è utilizzato il reasoner semantico Pellet. A rigore di chiarezza, si indica come specifica versione utilizzata Pellet 2.1.0; inoltre si sono utilizzate la libreria *jena 2.6.2*, per accedere al reasoner dal codice Java e la libreria jaxp, che serve per leggere e scrivere file XML.

A.3 Jquery

Si procede ora con la descrizione di Jquery [38], altra componente fondamentale da annoverare nell'elenco delle tecnologie fruite per l'architettura descritta in precedenza: Jquery è una libreria di funzioni (un cosiddetto software framework) per le pagine web, codificata in javascript, che si propone come obiettivo quello

di astrarre ad un livello più alto la programmazione lato client del comportamento di ogni singola pagina HTML. Tramite l'uso della libreria Jquery è possibile, con poche righe di codice, effettuare svariate operazioni, come ad esempio ottenere l'altezza di un elemento, o farlo scomparire con effetto dissolvenza. Anche la gestione degli eventi è completamente standardizzata e gestita automaticamente, assieme alla loro propagazione; stessa cosa per quanto riguarda l'utilizzo di AJAX, in quanto sono presenti alcune funzioni molto utili e veloci che si occupano di istanziare i giusti oggetti ed effettuare la connessione e l'invio dei dati. Il framework fornisce metodi e funzioni per gestire al meglio aspetti grafici e strutturali come posizione di elementi, effetto di click su immagini, manipolazione del Document Object Model e quant'altro ancora, mantenendo la compatibilità tra browser diversi e standardizzando gli oggetti messi a disposizione dall'interprete javascript del browser. Per controllare lo stile degli elementi, in maniera semplificata e standardizzata, sono forniti i metodi per l'utilizzo dei CSS, che eventualmente potranno essere introdotti e modificati agevolmente anche nell'ottica di un futuro intervento di manutenzione o ai fini di una eventuale commercializzazione.

Bibliografia

[1] Hui Guo, Anca Ivan, Rama Akkiraju, Richard Goodwin. *Learning ontologies to improve the quality of automatic web service matching.* 29/10/2009.

[2] Florian Daniel, Jin Yu, Boualem Benatallah, Fabio Casati, Maristella Matera, Regis Saint-Paul. *Understanding UI Integration: A survey of problems, technologies.* 2006.

[3] Florian Daniel, Jin Yu, Boualem Benatallah, Fabio Casati, Maristella Matera, Regis Saint-Paul. *A Framework for Rapid Integration of Presentation Components.* 2007.

[4] Florian Daniel, Maristella Matera. *Mash-up Context-Aware Web Application: a Component-Based Development Approach.* 2006.

[5] Serge Abiteboul, Ohad Greenshpam, Tova Milo, Neoklis Polyzotis. *MatchUp: autocompletion for mashups.* 2009.

[6] Matteo Picozzi, Marta Rodolfi, Cinzia Cappiello, Maristella Matera. *Quality-based Recommendations for Mashup Composition.* 2010.

[7] Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Gabriele Sprega, Donato Barbagallo, Chiara Francalanci. *DashMash: a Mashup Environment for End User Development*. 2010.

[8] Yahoo! Pipes. *http://pipes.yahoo.com/pipes/*. 2011.

[9] Roberta Ferrario, Nicola Guarino. *Towards an Ontological Foundation for Services Science*. 2009.

[10] Nicola Guarino, Claudio Masolo, Alessandro Oltramari, Aldo Gangemi, Laure Vieua. *La Prospettiva dell'Ontologia Applicata*. 2003.

[11] Repository dI ontologie Swoogle. *www.Swoogle.com*. Data ultimo accesso: 22/06/2011.

[12] Alessandro Cosola, Maddalena Losi. *Mutare le applicazioni web in componenti Mashup: proposte, modelli e soluzioni*. 2006.

[13] Gabriele Sprega, Matteo Picozzi. *DashMash: a Mashup Environment for End-User Development*. 2010.

[14] Hazem Elmeleegy, Anca Ivan, Rama Akkiraju. *MashupAdvisor: A Recommendation Tool for Mashup Development*. 2008.

[15] A.Kim, Y. Lee. *Quality Model for Web Services*. 2005.

[16] Eclipse. *http://www.eclipse.org/*. 2011.

[17] Open Mashup Alliance. *http://www.openmashup.org/oma- docs/v1.0/index.html*. 2010.

[18] ServFace. *http://141.76.40.158/servface/*. 2011.

[19] J. Wong, J. I. Hong. *Making mashups with marmite: towards end-user programming for the web.* 2007.

[20] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, A. Singh. *Damia: data mashups for intranet applications.* 2008.

[21] L. Zeng, B. Benatallah, M. Dumas. *Quality Driven Web Services Composition.* 2003.

[22] IBM. Qedwiki. *http://services.alphaworks.ibm.com/graduated/qedwiki.html.* 2011.

[23] JackBe. Jackbe presto. *Technical report, http://www.jackbe.com/.* 2011.

[24] Intel. Intel Mash Maker. *Technical report.* 2010.

[25] G. Fischer. *End-user development and meta-design: Foundations for cultures of participation.* 2009.

[26] F. Daniel, M. Matera, M. Weiss. *Web mashups: leveraging user innovation. Technical report, Politecnico di Milano.* 2009.

[27] F. Daniel, M. Matera. *Quando l'utente guida l'innovazione: Il web mashup.* 2010.

[28] Sven Casteleyn, Florian Daniel, Peter Dolog, Maristella Matera. *Engineering Web Applications.* 2009.

[29] D. Barbagallo, C. Cappiello, C. Francalanci, and M. Matera. *Applied Semantic Technologies: Using Semantics in Intelligent Information Processing.* 2010.

[30] SOA4All. *http://www.soa4all.eu/*. 2011.

[31] Web Semantico. *http://www.semanticweb.org/*. 2011.

[32] ProgrammableWeb. *http://www.programmableweb.com/*. 2011.

[33] S. Bechhofer, C. Goble. *Towards Annotation using DAML+OIL*. 2001.

[34] X.Liu, Q.Zhao, G.Huang, H.Mei. *A Mashup Framework for Web-delivered Service Composition*. 2011.

[35] Pellet. *http://clarkparsia.com/pellet/*. 2011.

[36] WordNet. *http://wordnet.princeton.edu/*. 2011.

[37] JBoss. *http://www.jboss.org/jbossas/downloads/*. 2011.

[38] JQuery. *http://jquery.com/*. 2011.

[39] WonderWeb. *http://wonderweb.semanticweb.org/*. 2011.

[40] JUNG. *http://jung.sourceforge.net/*. 2011.

[41] GoogleMaps. *http://www.google.com/apis/maps/*. 2011.

[42] Devis Bianchini, Michele Melchiori. *Mashing-up Semantic-enhanced Services*. 2010.

[43] P.Hitzler, M.Krötzsch, S.Rudolph. *Foundations of Semantic Web Technologies*. 2009.

[44] Search Computing. *http://www.search-computing.it/*. 2011.

[45] Devis Bianchini, Cinzia Cappiello , Valeria De Antonellis , Barbara Pernici. *P2S: A Methodology to Enable Inter-organizational Process Design through Web Services*. 2009.

[46] C. Cappiello, F. Daniel, M. Matera. *A quality model for mashup components*. 2009.

[47] C. Cappiello, F. Daniel, M. Matera, C. Pautasso. *Information quality in mashups*. 2010.

[48] F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. *Hosted universal composition: models, languages and infrastructure in mashart*. 2009.

[49] XAML. *http://en.wikipedia.org/wiki/xaml*. 2011.

[50] A. Puerta, J. Eisenstein. *Ximl: A universal language for user interfaces*. 2010.

[51] XUL. *http://en.wikipedia.org/wiki/xul*. 2011.

[52] UIXML. *http://download.oracle.com/*. 2011.

[53] MARIAE. *http://giove.isti.cnr.it/tools/mariae/*. 2011.