POLITECNICO DI MILANO

FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

End-User Development of Mashup: Models, Methods and Tools

Relatore: Prof. Maristella MATERA

Tesi di laurea di:
SPREGA Gabriele
Matr. 734892

Anno Accademico 2011 - 2012

Nella vita, a differenza degli scacchi,
il gioco continua dopo lo scacco matto.
Isaac Asimov

Mentre in fisica devi capire come è fatto il mondo,
in informatica sei tu a crearlo.
Dentro i confini del computer, sei tu il creatore.
Controlli, almeno potenzialmente, tutto ciò che vi succede.
Se sei abbastanza bravo, puoi essere un dio. Su piccola scala.
Linus Torvalds

I computer sono incredibilmente veloci, accurati e stupidi.
Gli uomini sono incredibilmente lenti, inaccurati e intelligenti.
L'insieme dei due costituisce una forza incalcolabile.
Albert Einstein

Ringraziamenti

Ho iniziato il mio cammino universitario ispirato dalla passione per l'informatica. Il primo passo l'ho compiuto quasi ingenuamente, guidato solo dal desiderio di approfondire e conoscere meglio ciò che riusciva ad affascinarmi. Il secondo passo mi ha reso consapevole che la passione va guidata dal metodo per sfociare in tutta la sua potenzialità. Il terzo passo mi ha fatto inciampare e cadere, ma anche scoprire di non essere solo nel cammino quando la mano di chi è al mio fianco ha saputo rimettermi in piedi. Il quarto passo rivelava la meta che volevo raggiungere. Compiendo il quinto passo per arrivare "al traguardo" ho compreso che ogni porta aperta ne svela infinite altre, in un cammino interminabile di scoperta di sé e del mondo.

Desidero ringraziare la Prof.ssa Matera e la Prof.ssa Cappiello per la fiducia, la guida e la stima che mi hanno sempre dimostrato, soprattutto in quest'ultimo anno di lavoro. Desidero anche ringraziare la Prof.ssa Francalanci per le sfide che ha posto, ispirando ulteriormente il nostro lavoro di tesi. Grazie al mio compagno di tesi Matteo e a Vincenzo, Lorenzo e Donato che hanno saputo rendere questa esperienza illuminante e sempre piacevole.

Il mio ringraziamento particolare, che supera i confini del lavoro di tesi abbracciando tutto il percorso universitario, va alla mia famiglia, il cui sostegno più "concreto" si è sposato alla fiducia e alla comprensione, a Martina, che con il suo amore è stata il mio "motore" in questi 5 anni, ai miei preziosi amici e "compagni di trincea" Leonardo, Maurizio e Alex che con l'amicizia, il supporto, la pazienza e la costante presenza hanno reso indimenticabile e agevole questo cammino.

Gabriele

Sommario

La tendenza corrente nello sviluppo delle applicazioni Web moderne – e in particolare delle applicazioni del Web 2.0 – punta chiaramente verso un notevole coinvolgimento degli utenti. Le cosidette applicazioni sociali sono la prova del valore, inizialmente inaspettato, che deriva dal coinvolgimento degli utenti finali nel processo di creazione di contenuti. Un'altra pratica emergente è lo sviluppo di applicazioni attraverso l'integrazione di contenuti e funzionalità che sono disponibili nel Web, sottoforma di API aperte o, in generale, di servizi riusabili. Un esempio classico di questa pratica è `www.housingmaps.com`, dove offerte immobiliari, estratte da Craigslist, sono visualizzate su mappe fornite da Google. Questo fenomeno è conosciuto come *Web mashup*, e mostra come gli utenti del Web siano sempre più coinvolti nel processo di sviluppo di applicazioni Web.

Le tecnologie per l'integrazione e la composizione di applicazioni sono state al centro degli interessi dei metodi di sviluppo del software degli ultimi 30 anni. Il problema da risolvere è la creazione di modelli di dati, applicazioni e interfacce utente attraverso l'integrazione di componenti già esistenti invece che optare per uno sviluppo *ex-novo* del software. L'integrazione può essere ottenuta in forme diverse, ma in generale è possibile distinguere tra tre classi principali: integrazione a livello dei dati, integrazione a livello di applicazione e integrazione a livello interfaccia utente (o, più in generale, a livello di presentazione).

L'integrazione a livello dei dati presenta diverse problematiche, dalla risoluzione di disallineamenti tra modelli di dati differenti, per esempio termini uguali con significati diversi, alla costruzione e, manutenzione di schemi virtuali e traduzione di interrogazioni tra schemi globali e locali. Ciò richiede interventi minimi a livello applicativo mentre necessita di una profonda comprensione della semantica del modello dei dati di ciascun componente – questo spesso è impraticabile.

L'integrazione a livello applicativo è stata largamente esplorata negli anni passati, con risultati tecnologici come RPC, object brokers e, più recentemente, Web services. In questo tipo di integrazione, un'applicazione composita ha una sua interfaccia utente (UI), ma la sua logica applicativa è sviluppata integrando le funzioni che le applicazioni componenti espongono. Questa pratica è molto efficente quando le applicazioni componenti espongono un API mantenuta stabile nelle varie versioni.

L'integrazione a livello di presentazione deve invece supportare l'integrazione delle interfacce utente dei vari componenti, lasciando la responsabilità della gestione dei dati e della logica applicativa ad ogni singolo componente. Questo tipo di integrazione è particolarmente adatta quando altri tipi di integrazione non sono possibili (come nel caso in cui le

applicazioni non espongano alcuna API), o quando lo sviluppo di una nuova interfaccia utente da zero è troppo costoso.

Il Word Wide Web è denso di esempi di integrazione di componenti eterogenei al fine di risolvere velocemente problemi contingenti, in alcuni casi totalmente divergenti dallo scopo per cui i singoli componenti sono stati sviluppati. Il sito Web http://www.programmableweb.com colleziona centinaia di tali esempi. Fino ad ora molti di questi mashup sono sviluppati a mano, utilizzando tecnologie diverse per garantire l'interoperabilità tra i vari componenti. Il risultato è spesso efficace dal punto di vista delle funzionalità offerte, ma è scarsamente documentato, difficilmente estendibile e in rari casi interoperabile. Inoltre, non esiste un modo sistematico e standard per descrivere le composizioni ed è difficile stabilire il confine tra la logica dei componenti e quella dell'integrazione.

Lo scenario delle tecnologie per l'integrazione è tuttora in evoluzione, sotto diversi punti di vista. Nel passato i tre tipi di integrazione sopra descritti erano chiaramente distinti – e infatti l'integrazione a livello di presentazione era alquanto inesistente. In più, l'integrazione era (ed è) complessa. I linguaggi di composizione ad oggi proposti (si veda per esempio BPEL [11] o i manuali utenti per strumenti ETL [6]) sono molto complessi, e lo stesso vale per lo sviluppo, l'esecuzione e l'analisi della logica di integrazione.

Recentemente i mashup hanno contribuito a creare maggiore confusione: per esempio *Yahoo!Pipes* [56] è considerato uno strumento per la creazione di mashup, ma il suo modello ha parecchie similarità con i modelli d'integrazione dei dati. Infatti è difficile classificare i mashup come strumenti di integrazione di dati, di applicazioni o di interfacce utente, perché essi ereditano diversi aspetti da ognuna di queste classi applicative.

Alcuni requisiti recentemente emersi nell'intergazione delle applicazioni, che sono estremamente rilevanti per questo lavoro di tesi, sono la semplicità, l'immediatezza e l'espressività, aspetti che sono tutti introddotti (o per lo meno potenziati) dal Web 2.0. In particolare la semplicità è un requisito chiave nello sviluppo delle applicazioni Web moderne. Funzionalità complesse e, recentemente, anche attività di sviluppo sono sempre più alla portata dell'utente finale. Si considerino, per esempio, gli ambienti di sviluppo di mashup offerti dai maggiori provider, come Intel, Google e Yahoo.

Il Web e lo sviluppo tecnologico rendono anche ogni risultato "visibile": siamo sempre più abituati a ritrovare in breve tempo ogni tipo di informazione, dalle informazioni sui voli aerei ai bollettini del traffico, alle statistiche di accesso sui nostri siti Web. La stessa visibilità è sempre più richiesta in ambito enteprise. Questa immediatezza e visibilità è la caratteristica principale dell' *end user development*.

Motivazioni

I mashup sono stati inizialmente concepiti nel contesto del "consumer Web", come strumento tramite cui gli utenti possono creare le loro applicazioni a partire da API pubbliche, come per esempio *Google Maps* o *Twitter API*, o da contenuti estratti da pagine Web. I mashup più popolari integrano API pubbliche disponibili sul Web, tuttavia la tendenza è di sviluppare applicazioni più critiche, i cosiddetti *entreprise mashup* [37], la traslazione

dei consumer mashup verso ambiti aziendali. Gli enterprise mashup abilitano i membri delle imprese ad utilizzare servizi interni che permettono di accedere agli asset aziendali, e mescolarli in modi innovativi che possono anche generare valore aggiunto. Per esempio, un obiettivo potrebbe essere l'automatizzazione di alcune procedure burocratiche ricorrenti. Basti pensare, per esempio, ai manager aziendali che vogliono creare da soli i loro cruscotti in modo flessibile e veloce, e all'enorme quantità di servizi corporate (per esempio quelli per l'accesso alle risorse informative aziendali), risorse Web e servizi aperti che, se integrati, possono enormemente facilitare la costruzione di applicazioni per l'analisi di dati e di processi. I mashup stanno perciò emergendo come tecnologia per la creazione di soluzioni innovative, capaci di rispondere ai problemi che sorgono ogni giorno nel contesto aziendale, così come in ogni altro contesto dove la flessibilità e la variabilità diventano variabili predominanti. La potenziale flessibilità dei mashup "aiuta gli utenti ad aiutarsi" [55], rendendo possibile la composizione "on demand" delle funzionalità di cui hanno bisogno.

Data la variabilità con cui i mashup sono (e saranno) usati, emerge il bisogno di fornire ambienti di sviluppo in cui gli utenti finali (gli attori principali del nuovo processo di sviluppo) possano facilmente e velocemente costruire autonomamente le loro applicazioni, senza dover necessariamente gestire le problematiche tecniche legate all'invocazione dei servizi e alla loro integrazione. Questo si applica in ogni contesto – non solo in quello aziendale: una "cultura della partecipazione" [30], in cui gli utenti evolvono dall'essere consumatori passivi di applicazioni all'essere co-creatori attivi di nuove idee, conoscenza e prodotti, è infatti sempre più condivisa [54].

Nonostante le premesse precedenti fino ad ora la ricerca sui mashup si è concentrata sulla definizione di tecnologie e standard abilitanti, ponendo scarsa attenzione sull'obbiettivo di facilitare il processo di sviluppo – in molti casi la creazione di mashup si basa ancora sulla programmazione manuale di servizi. È nostra convinzione che ciò che rende i mashup diversi dalla più tradizionale composizione di servizi o dall'integrazione di applicazioni e il loro potenziale come strumenti attraverso cui gli utenti finali sono abilitati a sviluppare le loro applicazioni e quindi ad innovare [54], tuttavia questo potenziale è scarsamente sfruttato. Alcuni studi recenti [12] hanno osservato che, nonostante le maggiori piattaforme di mashup (per esempio *Yahoo!Pipes* [56] o *Intel Mash Maker* [35]) semplificano lo sviluppo rispetto alla programmazione manuale, essi sono ancora difficili da usare da parte di utenti non tecnici.

Questa tesi cerca di rispondere alle precedenti questioni, proponendo una piattaforma Web, DashMash, che permette agli utenti finali di sviluppare i propri mashup facendo uso di un paradigma intuitivo. In particolare l'obiettivo del nostro lavoro è stato individuare delle astrazioni di alto livello che siano in grado, da un lato di catturare le proprietà di diverse classi di risorse che sono utili per l'integrazione di mashup, dall'altro di nascondere agli utenti i dettagli tecnici che derivano dalle diverse tecnologie adottate dai servizi e utilizzate per la loro integrazione.

Obiettivi della tesi

Esistono alcuni requisiti chiave che incoraggiano lo End User Development (EUD). Abbiamo identificato le più importanti proposte in letteratura, approfondendole e validandole nel contesto di un progetto proposto dal Comune di Milano. In questa occasione abbiamo suggerito i mashup come strumenti flessibili per la realizzazione di ambienti personalizzati a supporto della sentiment analysis [15]. Questa tesi si pone l'obiettivo di raggiungere tali requisiti. In particolare:

1. Promuove un processo di sviluppo leggero, nel quale un intuitivo paradigma di composizione viene contestualizzato all'interno di un ambiente visuale usabile, mascherando la complessità dovuta ai linguaggi di programmazione attualmente utilizzati per la gestione dell'esecuzione del mashup;

2. Propone dei modelli espressivi e dei linguaggi di descrzione per i servizi e per la composizione di mashup, capaci di racchiudere le caratteristiche salienti alla base del servizio di wrapping e di integrazione all'interno della composizione astraendo dai dettagli tecnici caratteristici di specifici componenti e tecnologie;

3. Presenta un framework di esecuzione che supporta il processo leggero di sviluppo assicurando:

   - Un supporto istantaneo all'esecuzione, basato sull'interpretazione delle azioni di composizione dell'utente e sull'esecuzione immediata del mashup secondo Un paradigma WYSIWYG (What You See Is What You Get);

   - La generazione automatica di modelli descrittivi risultanti dalle azioni di composizione degli utenti iche guidano l'esecuzione del masup;

   - La generazione di suggerimenti che aiutino l'utente finale ad identificare servizi candidati come utili da aggiungere all'interno del mashup e ad individuare le possibili modalità di integrazione.

4. Mentre da un lato fornisce un approccio sistematico al meccanismo alla base dello sviluppo leggero del mashup dal punto di vista del End User Programming, dall'altro mostra come questi meccanismi in generale validi per la composizione di ogni tipo di mashup possano essere specializzati per il dominio dei cruscotti (si veda il caso di studio sviluppato per il Comune di Milano).

5. Basandosi sui risultati di uno studio centrato sull'utente, presenta l'efficacia del nostro approccio secondo la prospettiva dell'utilizzatore finale.

Contenuti della tesi

La tesi è organizzata nel seguente modo:

- **Parte II: *Background***

- *Capitolo 2 - UI Composition*: questo capitolo illustra le tecnologie esistenti per l'integrazione a livello di presentazione e le differenze tra i vari approcci. Fornisce inoltre un overview dei più importanti approcci di ricerca e delle più importanti piattaforme commerciali.

- *Capitolo 3 - Data analytics and visualization*: questo capitolo illustra i concetti principali della data analytics e della costruzione di visualizzazioni avanzate di dati, sui quali abbiamo basato lo sviluppo di un insieme di componenti di mashup per la costruzione di cruscotti per la sentiment analysis.

- *Capitolo 4 - End-user development*: questo capitolo tratta l'End-User Development e declina i mashup su tale paradigma di sviluppo. Illustrando le principali attività che caratterizzano lo sviluppo di mashup, il capitolo sottolinea in particolare le potenzialità dei mashup come tool che possono essere utilizzati per l'end-user development.

- **Parte III: *Models and Architecture***

  - *Capitolo 5 - Models*: questo capitolo illustra i modelli che catturano le astrazioni su cui si fonda la composizione event-driven che caratterizza il nostro approccio. In particolare, propone un modello astratto per i componenti UI (UISDL – UI Service Description Language), un linguaggio dichiarativo per la specifica della composizione (XPIL – eXtensible Presenta- tion Integration Language), un modello per la rappresentazione dello stato (SMDL – State Model Descriptor Language) che facilita il monitoraggio e l'evoluzione della composizione, e un modello per il flusso dei dati (DFM – Data Flow Model), che definisce un formato comune per la rappresentazione dei dati scambiati tra i vari servizi nel mashup.

  - *Capitolo 6 - Architecture*: questo capitolo mostra l'implementazione client-side del middleware di composizione ed esecuzione; illustra i principali moduli architetturali che permettono (i) il paradigma di composizione visuale di DashMash, caratterizzato dalla definizione automatica e personalizzata dei binding tra servizi e la generazione di raccomandazioni sulla composizione, (ii) l'esecuzione event-driven dei mashup e (iii) le astrazioni e i meccanismi che stanno dietro alla generazione automatica di modelli di composizione e mashup.

- **Parte IV: *Validation***

  - *Capitolo 7 - Case study*: mostra all'opera i concetti presentati nei capitol precedenti e illustra la specializzazione della piattaforma DashMash per la costruzione di cruscotti per la sentiment analysis.

  - *Capitolo 8 - User testing*: presenta i risultati di un esperimento con la partecipazione di 35 utenti, che ci ha permesso di validare alcune ipotesi sull'usabilità di DashMash.

- **Parte V: *Conclusions***

– *Capitolo 9 - Conclusion*: traccia le conclusioni, riassumendo i principali risultati che sono stati raggiunti e delineando i possibili sviluppi futuri.

- **Part VI: *Appendix***

  L'appendice fornisce il materiale adottato per l'esperimento con gli utenti e un articolo relativo al lavoro svolto in questa tesi, pubblicato negli atti della conferenza International Conference on Web Engineering (ICWE 2011).

Contents

List of Figures

Part I

Rationale and Contribution

Chapter 1

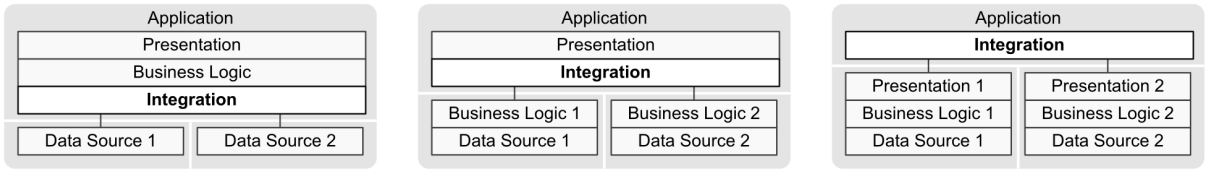
Introduction

The current trend in the development of modern Web applications - and in particular of those applications commonly referred to as Web 2.0 applications – clearly points toward a high user involvement. The so-called *social applications* prove the initially unexpected value of involving end users in the content creation process. Another practice that has emerged recently is the development of applications through the integration of contents and functionalities that are available on the Web in form of open APIs or reusable services. A "classical" example of this practice is `www.housingmaps.com`, which interweaves housing offers taken from the Craigslist with Google Maps. The phenomenon is commonly known as *Web mashups*, and it shows that Web users are increasingly also taking part in the development process of Web applications.

Integration and composition technologies have been one of the main focuses of software development methods and technologies for the last 30 years. The problem they try to solve is that of creating data models, applications, and user interfaces by integrating existing components rather than doing development from scratch. Integration may come in different forms, but in general we can distinguish among three main classes of integration: *data integration*, *application integration*, and *user interface* (or, more broadly, *presentation*) *integration*.

Data integration (see Figure 1.1(a)) presents several issues, ranging from the resolution of mismatches between component data models, such as the same terms having different meanings, to the construction and maintenance of virtual schemas and query mappings between global and local schemas. It requires limited cooperation from component applications; it however requires an understanding of the semantics of the data model of the components.

Application integration (see Figure 1.1(b)) has been thoroughly explored and researched in the past years, and the outcome of the research are technologies like RPCs and object brokers in the past and, more recently, Web Services. In application integration, a composite application has its own User Interface (UI), but its business-logic layer is developed by integrating the functions that component applications expose. Of course this works very well when the component applications expose an API that is kept more or less stable across versions.



(a) Integration of different data sources

(b) Integration of distributed business logic elements

(c) Integration of two autonomous applications

Figure 1.1: The different levels of integration: data, application, and UI

UI integration (see Figure 1.1(c)) should aim at composing applications in the presentation layer, leaving the responsibility of data and business-logic management to each component. UI integration is particularly applicable when application or data integration is not feasible (such as when applications do not expose business-level APIs), or when the development of a new UI from scratch is too costly.

The World Wide Web is full of examples of integration of heterogeneous components in order to quickly solve situational problems, even totally different from the purpose single components have been conceived for. The Web site `http://www.programmableweb.com` lists hundreds of such examples. So far, most of these mashups are crafted in an *ad-hoc* fashion, using different technologies as bridges between components. The result is often fully functional, but poorly documented, hardly extensible and scarcely interoperable. Moreover, there is not a standardized way of describing such compositions, and it is hard to tell where the boundary between the integration code and the actual components is.

The landscape of integration technologies today is in evolution, under many perspectives. In the past, the three types of integration described above were clearly separate, and in fact UI integration was almost non-existent. Furthermore, integration was (and still is) complicated. Composition languages proposed so far are very complex – see for example BPEL [11] or any user manual for ETL tools [6], as is the development, execution, and analysis of integration logic. Recently mashups contributed to mix-up things, creating further "confusion": for example, *Yahoo! Pipes* [56] is perceived as a mashup tool and indeed it mashes Web content, primarily RSS feeds; but its model shares many similarities with composition and integration models. Incidentally, it is hard to classify mashups as an information, application, or UI integration tool, as they inherits aspects from each of these application classes.

Some recent trends in application integration, which are extremely relevant for the topic of this thesis, are also those of simplicity, immediacy, and visibility, all introduced (or certainly accelerated) by Web 2.0. Simplicity is a trademark of modern Web application development. Increasingly, complex functionality and now even programming is brought to the end user, as in the case of recent mashup environments provided by major companies, such as Intel, Google, and Yahoo!. People are getting used to this simplicity and demand it more and more even for enterprise applications, even if the latter are more sophisticated and mission critical.

The Web, and technological advances also make everything visible: we are getting accustomed to having all sort of information, and in real time, from flight status to traffic

on the highways to detailed reporting on accesses to our Web site. The same visibility is now required by IT managers and CIOs on business processes and enterprise services. Interestingly, this immediacy and visibility is also characteristic of end user development, as for example pipes created with Yahoo!Pipes can be debugged "live".

1.1 Motivations

Mashups were initially conceived in the context of the consumer Web, as a means for users to create their own applications starting from public programmable APIs, such as Google Maps or the Twitter API, or contents taken from Web pages. The most popular current mashups integrate public programmable APIs. However, the vision is towards the development of more critical applications, for example the so-called enterprise mashups [37], a porting of current mashup approaches to company intranets. Enterprise mashups enable enterprise members to play with company's internal services that give access to the enterprise information assets, and to mash them up in innovative, hopefully value-generating ways. For example, a goal could be the automation of a recurrent bureaucratic procedure. Think for example to enterprise managers that want to compose by themselves and in a flexible way their dashboards, and to the plethora of corporate services (e.g., for the access to a variety of enterprise information sources), Web resources and open services that, if integrated, can largely ease the construction of applications for process and data analysis. Mashups are therefore gaining momentum as a technology for the creation of innovative solutions, able to respond to the different problems that arise daily in the enterprise context, as well as in any other context where flexibility and task variability become a dominant requirements variable. The potential flexibility of mashup environments helps people help themselves [55], by enabling the on-demand composition of the functionalities that they need.

Given the variability of contexts in which mashups are (and are going to be) used, the need arises to provide environments where the end users (i.e., the main actors of this new development process) can easily and quickly self-construct their applications without necessarily mastering the technical features related to service invocation and integration. This is true in every context - not only in the Enterprise: a "culture of participation" [30], in which users evolve from passive consumers of applications to active co-creators of new ideas, knowledge, and products, is indeed more and more gaining momentum [54].

Despite the previous premises, so far the research on mashups has focused on enabling technologies and standards, with little attention on easing the mashup development process – in many cases mashup creation still involves the manual programming of the service integration. We believe that what makes mashups different from plain Web service composition or application integration is their potential as tools through which end users are empowered to develop their own applications and, thus, innovate [54]. However, this potential is rarely exploited. Some recent user-centric studies [12] found that although the most prominent mashup platforms (e.g., Yahoo!Pipes [56] or Intel Mash Maker [35]) simplify the mashup development, they are still difficult to use by non technical users.

This thesis tries to respond to the previous needs, and proposes a Web platform, *Dash-*

Mash, that allows end users to develop their own mashups making use of an intelligible paradigm. The goal of our work has been in particular to identify high-level abstractions that on one side capture the most relevant properties of the different classes of resources, useful for their integration within mashups, while on the other hand are able hide to the users the technical details deriving from the heterogeneous technologies that characterize mashup services and especially their integration.

1.2 Goal of this Thesis

There are some key requirements that encourage End User Development (EUD). We identified the most relevant ones proposed by the literature and we also investigated and validated them in the context of a project funded by the Comune di Milano (Milan Municipality), where we promoted mashups as a flexible tool for the construction of personalized self-service environments supporting sentiment analysis [15]. This thesis work aims at addressing such requirements. In particular:

1. It promotes a lightweight development processes, in which an intuitive mashup composition paradigm is contextualized within a *usable* visual environment hiding the complexity of the composition languages actually managing the execution of the mashup, thus enforcing end-user programming.

2. It proposes expressive models and description languages for the mashup services and the mashup composition, able to capture the most salient features at the basis of service wrapping and service integration within mashups, abstracting from the technical details that are characteristic of specific components and composition technologies.

3. It presents a runtime framework supporting these lightweight development processes, that ensures:

   - An instant execution support, based on the "on the fly" interpretation of the user composition actions and the immediate execution of the mashup composition in a WYSIWYG (What You See Is What You Get) manner;

   - The automatic generation of descriptive models as resulting from the users composition actions, which then drive the execution of the mashup;

   - The generation of recommendations helping the end-user to identify candidate services to be added within the mashup and also possible ways to integrate them.

4. While on the one hand it gives a systematic view over mechanisms for the lightweight development of mashup based on end-user programming, through the case study developed for the Milan Municipality it also shows how these mechanisms, that are in general valid for the composition of any mashup class, can be specialized for the specific domain of dashboard construction.

5. Based on the results of a user-centric study, it discusses the effectiveness of our approach from the end user perspective.

1.3 Outline

The rest of this thesis is organized as follows:

- **Part II: *Background***
  - *Chapter 2 - UI Composition:* this chapter illustrates existing User Interface composition technologies and the differences between different integration approaches. It also overview the main research approaches and commercial platforms so far proposed for mashup development.
  - *Chapter 3 - Data analytics and visualization*: in this chapter we give an overview of some relevant notions of data analytics and advanced data visualizations, on which we based the development of a set of mashup components for the construction of mashup-based sentiment analysis dashboards for our case study.
  - *Chapter 4: End-user development*: this chapter discusses End-User Development and relates mashups to such user-centric development paradigm. By illustrating the main activities that characterize the mashup development, the chapter in particular highlights the potential of mashups as tools that can strongly enforce end-user development.
- **Part III: *Models and Architecture***
  - *Chapter 5: Models*: this chapter gives an overview of the models that capture the abstractions on which we founded the *event-driven* mashup composition that characterizes our approach. We in particular propose an abstract model for presentation components (UISDL - UI Service Description Language), a declarative composition language (XPIL - eXtensible Presentation Integration Language), a state model (SMDL - State Model Descriptor Language), easing the mashup composition monitoring and evolution, and a Data Flow Model (DFM), providing a common format for the representation of data to be exchanged among the different mashup services.
  - *Chapter 6 - Architecture*: this chapter shows a client-side implementation of the composition and execution middleware; it illustrates the fundamental architectural modules that enable (i) the DashMash visual composition paradigm, characterized by the automatic and custom definition of service bindings and the generation of composition recommendations, (ii) the event-driven mashup execution, and (iii) the abstractions and mechanisms behind the automatic generation of composition models and mashups.
- **Part IV: *Validation***

  - *Chapter 7 - Case study*: it shows the previous concepts at work, and illustrates an instantiation of our DashMash platform for the construction of sentiment analysis dashboards.
  - *Chapter 8 - User testing*: it presents the results of an experiment involving 35 users, which allowed us to investigate some hypotheses about the usability of our platform.

- **Part V: *Conclusions***
  - *Chapter 9 -Conclusion*: it draws the conclusions, summarizing the main results that have been achieved and outlining the future work.

- **Part VI: *Appendix*** An appendix also provides details about the material adopted for the user validation and an article about the work of thesis, published at the International Conference on Web Engineering (ICWE 2011).

Part II

Background

Chapter 2

UI Composition

UI (or, more broadly, *presentation*) *integration* allows an application to participate with others as if they were designed as a single application. By this way, it is not necessary to compile, package and release components composition as a single monolithic unit. More specifically, UI integration composes applications by reusing their own user interfaces. This means that the presentation layer of the composite application is itself composed, at least in part, by the presentation layers of the components. UI integration, is particularly applicable in cases where application or data integration is not feasible (e.g., the applications do not expose a business level API), or where the development of a new UI from scratch is too costly (e.g., the component application often change or its UI is complex) [26].

Facing the problem of presentation-level integration (PI) requires to get benefit from the great number of researches in the field of integration; it is also necessary to keep in mind that integration problems and their correlated solutions differ in the kind of integration needed, though certain approaches are common and seem to be more successful and applicable than others.

The starting point on which we need to focus our attention consists in finding a homogeneous way to describe the different components to be integrated. Moreover, the description should be simple, formal, human readable and modular: in fact, simplicity often leads to diffusion as well as readability, which is essential for developers who often need to read the specification directly. Modularization is also fundamental because it enables the incremental expansion of the integration model. Using these paradigms we can find a concrete example of how high is the probability of success in term of diffusion in a well-known language such as WSDL [22].

It is also useful to reflect about the importance of tool support: as a matter of fact, tools relevant for integration include both development environment as well as run-time middleware that handle component bindings and interaction. Moreover, tools also act as mediators between language representation and users, allowing reaching a larger number of developers.

It is quite ambitious to cover all these aspects in a single specification. A good approach could be to proceed incrementally, starting from basic requirements and then adding

functionalities if needed. This is for example the approach adopted by Web Services.

Another interesting lesson, borrowed from application integration, is the success of queue-based, publish-subscribe and bus mediated approaches to interoperability. The value of such solution has been proved by the success of message broker platforms and by the fact that even in Web services, originally born for fully decentralized interaction with no assumption on a common middleware, the notion of enterprise service bus quickly emerged and now is the common approach to implement SOAs, at least within the enterprise.

The following sections with deeper the above mentioned issues, by highlighting the PI peculiarities.

2.1 Peculiarity of Presentation Integration

One of the main features of UI integration is that it is typically event-driven, and specifically guided by end-users' actions: when the user interacts with the UI of a component, it will react according to its own logic, generally involving an application-specific state change. At this point, the rest of the components in the same composite application need to be aware of this change, so that they can update their UI accordingly. It follows that communication between components mainly consists of notifications of state changes. Therefore, in a composite application, the main difficulty is to manipulate a component state as well as to detect its state changes.

Presentation Integration clearly appears different from, for example, application integration: in application integration components offers an arbitrary set of method offering support for service invocation and data reply; integration is mainly procedural, achieved via the specification of fairly complex control logic (e.g., in BPEL or other workflow-like language) that causes the invocation of services typically in some predefined sequence.

Another peculiarity of UI integration is the notion of configuration parameters, for the purpose of design-time component customization: a developer might want to specify characteristics of components UI appearance, like background and zoom level. Moreover, in presentation integration the runtime middleware migh need to know if the UI is visible or hidden, minimized or maximized: the middleware should be able to monitor, query, and update the presentation modes of the components. In addition, components in PI also require proper layout management, related to properties such as the location, size, shape, transparency, and z-order of the presentation components.

2.2 The UI Integration problem

In UI integration is possible to identify four mainly issues:

- to define models and languages for component specifications.
- to define models and languages for component composition specifications;

- to choose communication styles for component interaction;
- to define discovery and binding mechanism (also during run-time) for identifying components.

Also taking into account the guiding principles illustrated before, i.e., simplicity, formalism, readability and modularity of specifications, it is possible to identify some possible solutions for managing components and the interactions with their presentation layer; each one characterized by a given level of complexity, technology and programming friendliness:

- **Component model** : in application integration components are characterized by an API (Application Programming Interface) and also a component model. In data integration, data source schema describe the component. Indeed, UI integration at presentation layer, besides the reuse of class libraries, requires a component model that can support complex interactions and coordination. Every single component is described by a software interface enabiling service invocation and a user interface that enables interaction. A UI component interface allows several levels of interoperability.
- **GUI-only**: all interactions with GUI-only components are performed through the component's UI logic. The only method to integrate a GUI-only component is to know its UI and be able to track the mouse position or key strokes and to understand what the component's UI shows so that is possible to execute actions that cause UI modification.
- **Hidden interface**: in many Web applications, the component has an interface that lets users control its UI, but it is not publicly described. Web applications interaction allows the access and manipulation of content by sending HTTP requests and displaying the response. These applications obey a general protocol for interaction among clients and applications, usually hard to identify.
- **Published interface**: in this ideal case, the component provides a public description of its UI and an API in order to manipulate it at runtime. A low level API might allow control of individual UI element. A high-level API would expose a set of entities and controllable objects, as well as operations to change entity status.

2.2.1 Composition language

A composition language supporting identification and specification of the elements involved in an orchestration and their interaction. As for as data integration, composition often occurs via SQL views that allow one to express a global schema as a set of views over a local schema.

In application integration the composition can be described either via general-purpose programming languages such as Java, or dedicated application integration languages, such as workflow or service composition languages. In UI composition there might be two kinds of solutions:

- **General-purpose programming languages**: developers can adopt third-generation languages for application composition. Such languages are very flexible but lack abstractions to coarse-grained components (such as facilities for component discovery and binding or high-level primitives for synchronizing what UI components display).
- **Specialized composition languages**: high-level languages are typically used with a XML syntax tailored to the composition of UI components at the level of abstract/external descriptions. The main benefit of such languages is higher-level programming of the compositions, that leverages the component model's characteristics.

2.2.2 Communication style

We have already underlined the importance of communication in UI integration: the need to monitor UI events within some component that can update the state (and the UI) of other components. It is possible to distinguish between two types of communication:

- **Centrally mediated communication**, in which the composite application has a central coordinator that receives events from the components and issues instructions to manipulate component UIs.
- **Direct component-to-component communication**, in which the composite application is a coalition of individual components.

These solutions clearly differ from data integration where components are typically passive and do not initiate communications with the integrating application. Application integration shows the same difference: a centralized entity (the composite application) invokes components as needed.

An additional distinction between these types of communication in UI integration is the one between RPC-style interaction, in which components exchange information via method calls and returned data, and publish-subscribe interaction [29], in which applications communicate in a loosely coupled way via messages exchanged through message brokers.

2.2.3 Discovery and binding

A relevant problem in UI integration is very important to discover the components involved in the composition and understand how to get reference to them.There are two different ways to do this: the first consists in defining the composition statically at design or deployment time and the second consists in doing this dynamically at runtime.

We observe that in data and application integration the binding between different data sources typically occurs at design time when the data global schema is defined and the discovery is performed dynamically by an integration middleware. However this approach lacks flexibility because of the difficulty of interacting with newly discovered components later added to the initial integration.

A *hybrid binding solution* is also used in which the application designer identifies and tests a set of potential components and the users then select a subset of them at runtime based on the task they need to tackle: in this case the discovery is static but the reference is dynamic. This hybrid approach is possible in UI integration, too.

2.2.4 Components visualization

There are several solutions for component visualization, whose distinction is based on the UI rendering paradigm. In fact, the component can display its UI in a first kind of solution, while the composite application receives UI markup code from the components and renders them in a second kind of solution.

The second one needs a markup description, that has to be interpreted using a rendering engine in order to obtain a translation in graphical elements: the markup specification describes often static UI properties, whereas scripting languages provide dynamic behavior. Such description can be done with document-oriented languages or UI languages that model sophisticated application interfaces. Examples of such language are XAML[3], XUL[4], UIXML[2], XIML[49] (Table 2.1).

XAML	Extensible Application Markup Language is a declarative XML-based language created by Microsoft which is used to initialize structured values and objects. XAML elements map directly to Common Language Runtime object instances, while XAML attributes map to Common Language Runtime properties and events on those objects. Anything that is created or implemented in XAML can be expressed using a more traditional .NET language, such as C# or Visual Basic.NET. However, a key aspect of the technology is the reduced complexity needed for tools to process XAML, because it is based on XML.
XUL	XML User Interface Language, is an XML user interface markup language developed by the Mozilla project. XUL provides a portable definition for common widgets, allowing them to move easily to any platform on which Mozilla applications run. While XUL serves primarily for constructing Mozilla applications and their extensions, it may also feature in Web applications transferred over HTTP.
UIXML	uiXML is a XML language for programming UIX applications. uiXML builds on top of the other UIX technologies, providing a XML language for specifying user interfaces and linking them to data and events. uiXML provides a declarative alternative to creating Web applications programmatically with the UIX Java APIs. The pages, events, and any other items defined with uiXML elements are all transformed into Java objects behind the scenes and are thus treated equally by UIX. No compilation is required.
XIML	eXtensible Interface Markup Language is a XML-based language that enables a framework for the definition and interrelation of interaction data items. As such, XIML can provide a standard mechanism for applications and tools to interchange interaction data and to interoperate within integrated user-interface engineering processes, from design, to operation, to evaluation. XIML is an organized collection of interface elements that are categorized into one or more major interface components.

Table 2.1: Overview about some markup description languages

It is possible to have two different UI rendering solutions:

- **Component-rendered UI:** the component handles the UI's rendering and display, thus the composite application is a collection of the component's UIs. This is the case of the classical desktop applications that leverage executable components of linked graphical libraries.
- **Markup-based UI:** the component returns UI code and delegates the final UI rendering to either the composite application or the running environment able to interpret the components' UI code and to allocate suitable layout space for compo-

nent rendering. In this case, user interaction with the component can be handled directly via a suitable scripting logic embedded in the component's markup, or the composite application can intercept generated UI events and forward them to the component for interpretation.

2.3 UI Composition technologies

We illustrate in Figure 2.1 a comparison between the different UI technologies considered in the context of UI composition. This comparison focused on the principal aspects of UI integration previously presented. The compared technologies are shortly described in the rest of this section.

	UI component model and external specification	Composition language	Communication style	Discovery and binding	Component visualization
Desktop UI components	Published, programmable API	General-purpose programming language	Centrally mediated and component-to-component communication could be supported	Static and dynamic binding	Component rendered
Browser plug-in components	Published, basic interface (startup configuration parameters)	Document markup code and JavaScript	Centrally mediated; very limited intercomponent communication via ad hoc JavaScript	Static binding	Component rendered
Web mashups	Hidden interface; published API	General-purpose programming language	Centrally mediated	Static binding	Typically markup based
Web portals and portlets	Standard interface based on public API; interface wrapped as a Web service	General-purpose programming language	Centrally mediated (interportlet communication under development)	Static and dynamic binding	Markup based

Figure 2.1: Comparison of current UI integration approaches

2.3.1 Desktop UI components

Historically UI composition was born for desktop applications, to create an environment in which applications developed with heterogeneous languages could interoperate. Think for example to ActiveX, which leverages Microsoft's COM technology for embedding a complete application UI into host applications, and the composite UI application Block (CAB) [9] that is a framework for UI composition in .NET with a container service that lets developers build applications on loadable modules or plugins. In particular, CAB components can be used with any .NET language to build composite containers and perform component-container communications. CAB provides an event broker for many-to-many, loosely coupled inter-component communication based on a publish-subscribe runtime event model.

Another example is Eclipse's Rich Client Platform(RCP)[1], which provides a similar framework, but it also includes an application shell with UI facilities. It further offers a module-based API that lets developers build applications on top of this shell. Eclipse

also allows developers to customize and extend UI components via so-called extensions points, a combination of Java interfaces and XML markups.

Desktop UI components typically use general-purpose programming languages to integrate components (C# for CAB and Java for RCP) because the component interfaces are language-specific programming APIs. Components manage their own UI rendering; they could support flexible communication styles. Both design-time and runtime bindings are supported as well, with the latter relying on language-specific reaction mechanism.

Many of the technologies for desktop UI components are OS dependent. Although CAB and RCP do not depend on the OS directly, they rely on their respective runtime environments. The lack of technology-agnostic, declarative interfaces makes interoperation between components implemented with different technologies difficult to achieve.

2.3.2 Browser plug-in components

Browser navigation experience usually involves advanced UI features. The main technologies used to create embedded UI components in markup-based interfaces are Java applets and ActiveX controls. After the definition of the components binding at page authoring time by Web designer, the browser downloads the components at runtime and instantiates them.

These components often provide their own rendering, with a little further communication between themselves and the containing Web page. The external interface of such components is very simple and usually requires only the proper configuration parameters when embedding components into the markup code.

2.3.3 Web mashups

Web mashups are Web sites that wrap and reuse third-party Web content. A developer thus performs the integration in an *ad-hoc* fashion by leveraging whatever programming language the content source provides, either on the client or server side.

Content providers typically provide content as markup code, and mashup developers integrate it in a centrally mediated way. Because content is markup based, the composite application usually renders the components. The lack of infrastructure makes component communication difficult and only provides a way to statically bind components.

2.3.4 Web portals and portlets

Web portal development explicitly distinguishes between UI components (portlets) and composite applications (portals). Portlets are full-edged, pluggable Web application components; they generate document markup fragments that adhere to certain rules, thus facilitating content aggregation in portal servers to ultimately form composite documents.

Portal servers typically let users customize composite pages and provide single sign-on and role-based personalization.

Analogous to Java servlets, portlets implement a specific Java interface to the standard portlets API, which was intended to help developers create portlets that can plug into any standard-conform portal server. For Java portlets, portal application are based on the Java programming language, whereas with Web Parts, a Web developer programs applications in .NET.[8] The portal application aggregates its portlets' markup outputs and manages communication in a centrally mediated fashion.

Portlets also allow both static and dynamic binding; during runtime, the portal application can make portlets available in a registry for user selection and positioning. Although portlets and Web Parts have similar goals and architectures, they are not interoperable. Web Service for Remote Portlets (WSRP) addresses this issue at the protocol level by exposing remote portlets as Web Services; communication between the portal server and portlets occurs via SOAP, which means developers can build the portal and portlets with different languages and runtime framework.

2.4 Tool-Assisted Mashup Development

To speed the overall mashup development process, but also to enable even inexperienced end users to mash up their own Web applications, numerous mashup-specific development tools and frameworks have recently emerged.

These instruments typically come with a variety of features and a mixture of composition approaches. A close look at them lets us identify the open issues and research challenges characterizing the mashup phenomenon. For presentation purposes, we selected the most popular or representative approaches of end-user mashup tools.

2.4.1 Yahoo Pipes

Figure 2.2: Yahoo Pipe logo

Yahoo Pipes [56] lets you mix popular data feeds in order to create data mashups via a visual editor. A pipe is a data processing pipeline consisting of one or more data sources (for example, RSS/Atom feeds or XML sources) and a set of interconnecting operators, each of which performs a specific task.

It includes operators for manipulating data feeds (for example, sorting or filtering) and operators for features such as looping, regular expressions, or counting. It also supports more advanced features, such as location extraction (for example, geocoordinates identified and converted from location information found in text fragments) or term extraction (for example, keywords).

Yahoo Pipes therefore aims to let users design data-processing pipelines that filter, transform, enrich, and combine data feeds and are again exposed as RSS feeds.

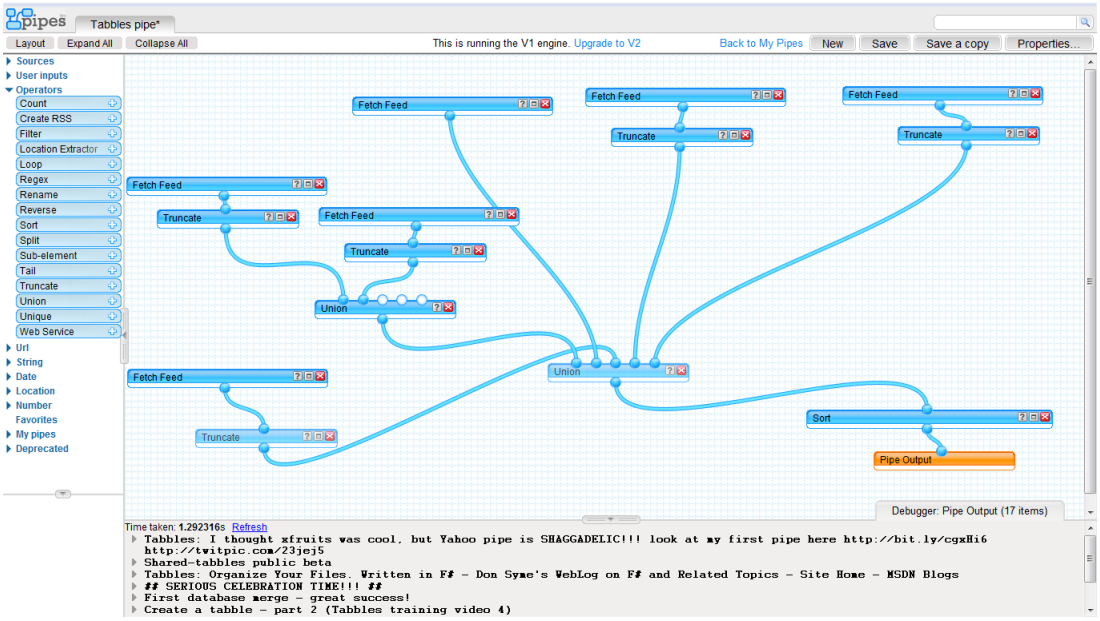


Figure 2.3: Yahoo Pipe Editor Example

2.4.2 Intel Mash Maker

Figure 2.4: Intel Mask Maker logo

Mash Maker[35] provides an environment for integrating data from annotated source Web pages based on a powerful, dedicated browser plug-in. Rather than taking input from structured data sources such as RSS or Atom, Mash Maker lets users annotate Web pages' structure while browsing and use such annotations to scrap contents from annotated pages.

Advanced users can leverage the integrated structure editor to input XPath expressions using FireBug's DOM Inspector (a plug-in for the Firefox Web browser). Composing mashups with Mash Maker occurs via a copy-and-paste paradigm, based on two modes of merging contents:

- *Whole page merging*, in which the user inserts a page's content as a header into another page;
- *Item-wise merging*, in which the user combines contents from two pages at row level, based on additional user annotations. You can use the two techniques to merge more than two pages.

2.4.3 Quick and Easily Done Wiki

Figure 2.6: QEDWiki logo

QedWiki [34] is the IBM's proposal for a wiki-based "mashup maker", fully running inside the client browser and allowing access to IBM's

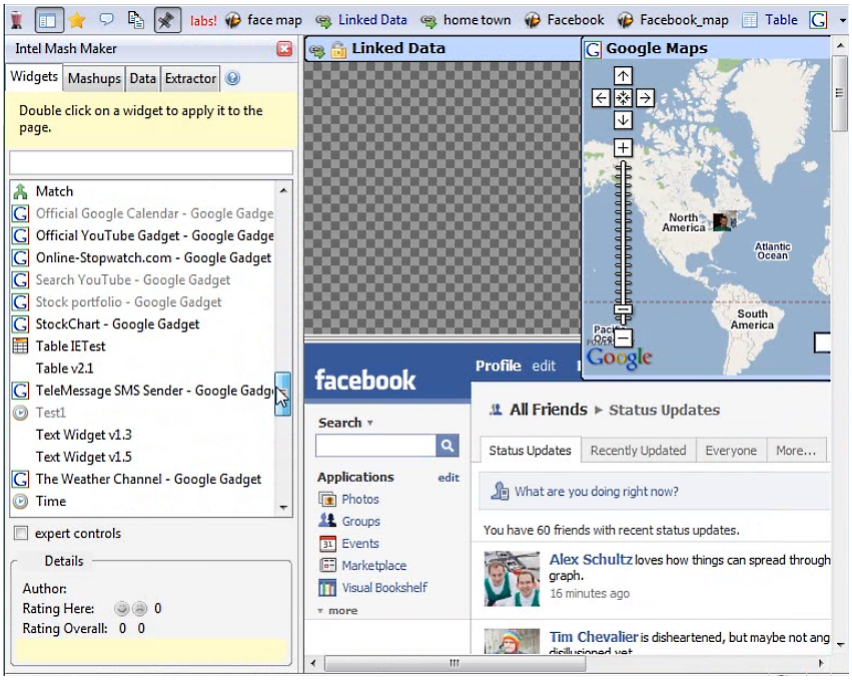


Figure 2.5: Intel Mash Maker example

Mashup Hub. The Hub supports the creation of data feeds and user interface widgets and incorporates Data Mashup Fabric for Intranet Applications (Damia) for data assembly and manipulation.

As a wiki environment, it lets users edit, immediately view, and easily share mashups. Mashups are assembled from JavaScript or PHP-based widgets, whose wiring determines the mashup's behavior. Widgets represent application components and might or might not have their own user interface.

To assemble a mashup, a user selects a page layout (an HTML template) and then drags and drops widgets onto the page grid and interactively configures them.

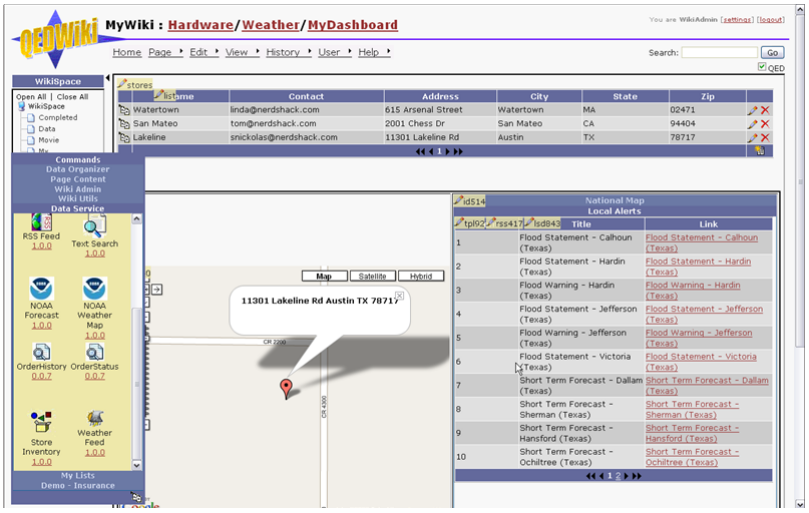


Figure 2.7: Quick and Easily Done Wiki example

2.4.4 Damia

Through a Web-based interface, *IBM DAMIA* [52] provides easy-to-use tools that developers and IT users alike can use to quickly assemble data feeds from the Internet and a variety of enterprise data sources. The benefits of such service include the ability to aggregate and transform a wide variety of data or content feeds, which can be used in enterprise mashups.

DAMIA lets you do the following:

- Import XML, Atom, and RSS feeds.
- Assemble feeds from both the Internet and from Excel spreadsheets. Database support is coming soon.
- Import data from local les in XML format and Excel spreadsheets.
- Aggregate and transform a wide variety of data or content feeds into new syndication services. When building a complete Web application that provides a user interface, additional tools or technologies are required in order to display the data feed provided by DAMIA. Mashup makers and feed readers that consume Atom and RSS can be used as the presentation layer in the enterprise Web application.

DAMIA is composed of the following:

- a browser-based Web application for assembling, modifying and pre-viewing mashups.
- services for handling storage and retrieval of data feeds created within the enterprise as well as on the Internet. In addition to creating data feeds from various sources, *DAMIA* can publish information such as Excel spreadsheets or XML documents in mashup formats.
- a repository for sharing and storing feeds or information created by *DAMIA*.

- services for managing feeds and information about mashups, search capabilities and tools for tagging and rating mashups.

2.4.5 JackBe Presto

Figure 2.8: JackBe Presto logo

JackBe Presto[36] is an Enterprise Mashup Platform that includes functionality for creating and syndicating enterprise mashups. It provides a support for application developers and power users to create, customize and share Enterprise Apps mashups.
JackBe Presto functionalities includes:

- service access engines for Web services, SQL, RSS, and Web clipping;
- mashup composers/creators including a graphical, drag-and-drop tool and a declarative enterprise mashup markup language;
- mashup connectors for popular enterprise tools including Microsoft Excel, HP Systinet, and Oracle WebCenter;
- mashup APIs for JavaScript, Java, REST, C#, and .NET;
- Enterprise Mashup Markup Language (EMML)[45].

In March 2010 JackBe lanched a cloud-based version of its Presto product hosted on Amazon EC2.

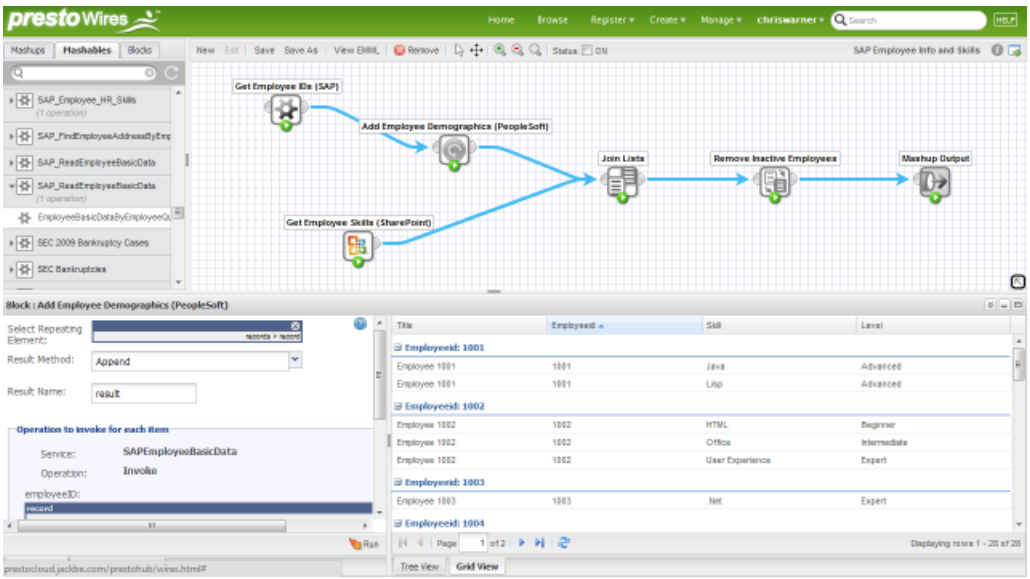


Figure 2.9: PrestoWires application example

2.4.6 Marmite

Marmite is a tool created by Jeffrey Wong and Jason I Hong [55]. It is presented as a solution that lets users create their own mashup without pre-required programming skills. It permits to:

- access to Web Service APIs;
- combine Web Service APIs with screen-scrape oriented programming;
- present a tool composed by an operators menu, a data flow view and data view.

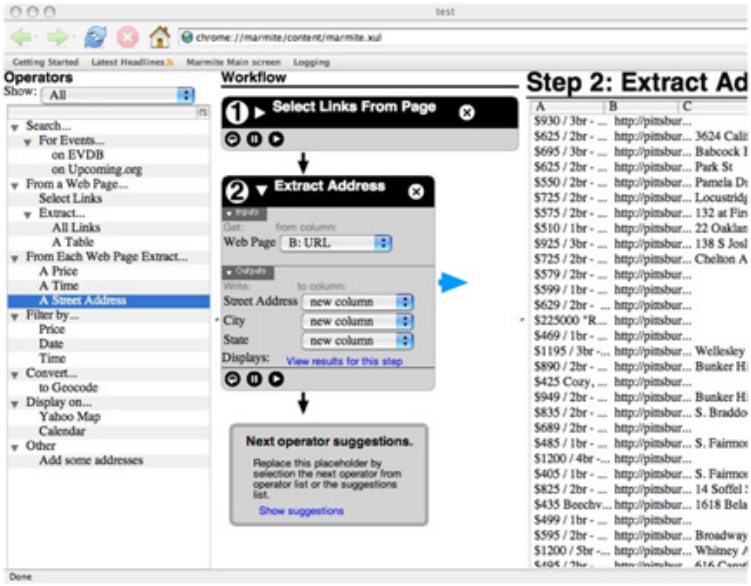


Figure 2.10: Marmite example

2.4.7 MashArt

MashArt [23] is a platform based on UI integration developed by University of Trento in collaboration with SAP. The core engine of MashArt is an evolution of MixUp, a previously delivered mashup engine which is also the basis of this thesis work. The UI composition is event-based, interpreted at runtime, and uses standard HTML templates for the layout. UI components in the composition uses events and operations to communicate/enact state changes. They are describe by an abstract descriptor (MDL).

MashArt provides a drag & drop editor that permits a visual definition of the mashup at design-time, during which user can define the integration logics and the placement of the components into a layout grid.

2.4.8 ServFace

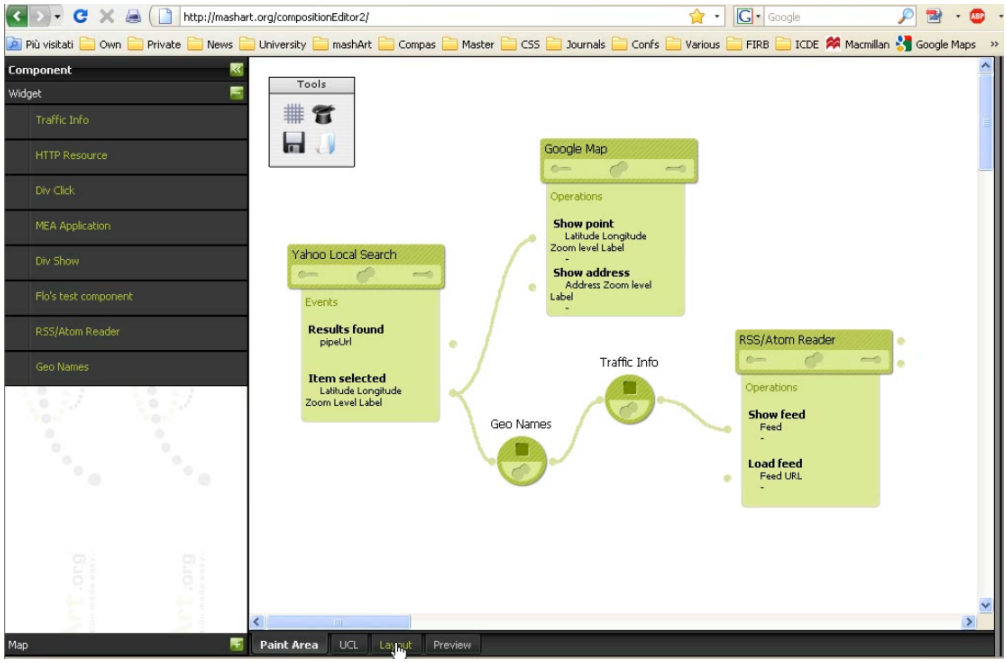


Figure 2.11: MashArt example

Figure 2.12: ServFace logo

The *ServFace*[51] project aims at creating a model-driven service engineering methodology for an integrated development process for service-based applications.

The set of Service Annotations identified in the ServFace project are captured in the ServFace Annotation Model. Together with technical service descriptions like WSDL, it provides the necessary input for an automated user interface inference mechanism that generates high quality user interfaces for the interaction between human users and annotated Web services.

For the composition of annotated services to complex applications, two alternative modeling approaches are investigated in ServFace.

- *Presentation-oriented service composition*, in which the application is modeled visually by composing the application's UI from parts that are generated using the service annotations. Moreover, a ServFace Builder tool is under construction: it will integrate an inference engine to generate user interfaces from annotated services. This approach is the one most oriented in towards end-user development.
- *Task-oriented service composition*: it is supported by a tool called MARIAE [5]. It provides a novel solution able to exploit task models (represented in the ConcurTaskTrees notation) and user interface models (in the MARIA language) for the design and development of interactive applications based on Web services for various types of platforms (desktop, smartphones, vocal, ...). The tool is able to automatically import service and annotation descriptions and support interactive association of basic system tasks with Web services operations. Then, a number of semi-automatic transformations exploit the information in such service and anno-

tation descriptions to derive usable multi-device service front ends.

Chapter 3

Data analytics and visualization

As we will show in Chapter 7, the case study developed for this thesis is a mashup development tool for the construction of Web reputation dashboards.

This class of mashups is characterized by components in charge of extracting data from a Data Warehouse and visualize them through advanced graphics visualization. Therefore, our work is also characterized by the use of Data Analytic and Visualization techniques.

In this chapter we will introduce the principal concepts that guided our work.

3.1 Data Analytics

Enterprise mashups enable enterprise members to play with company's internal services that give access to the enterprise information assets, and to mash them up in innovative, hopefully value-generating ways. In this context Enterprise Mashup aimed at analysts or decision makers. End-user would compose his own mashup to analyse data in order to draw conclusions and make strategic decisions.
Data analytics is used in Industry, to allow companies and organizations to make better business decisions, and in the Sciences, to verify or disprove existing models or theories.

Data analytics distinguishes from data mining by scope, purpose and focus of the analysis. Data miners sort through huge data sets using sophisticated techniques to identify unexpected patterns and hidden relationships. Data analytics focuses on inference, the process of deriving a conclusion based solely on what is already known by the researcher.

Data analytics is generally divided into:

Exploratory data analysis (EDA) where new features in the data are discovered;

Confirmatory data analysis (CDA) where existing hypotheses are proven true or false;

Qualitative data analysis (QDA) used in the social sciences to draw conclusions from non-numerical data like words, photographs or video.

The term *analytics* has been used by many business intelligence (BI) software vendors as a buzzword to describe quite different functions. Data analytics is used to describe everything from online analytical processing (OLAP) to CRM analytics in call centers. Banks and credit cards companies, for instance, analyze withdrawal and spending patterns to prevent fraud or identity theft. Ecommerce companies examine Web site traffic or navigation patterns to determine which customers are more or less likely to buy a product or service based upon prior purchases or viewing trends. Modern data analytics often use information dashboards supported by real-time data streams.

Our work is related to the dashboard construction. The tool we propose support the composition of Enterprise Mashup dashboards, where each viewer is a mashup component. The available mashup components are especially *ad-hoc* created "viewers", offering advanced visualization of some trend of data aggregations.

3.2 Visualization families

In order to support analyses, it is important to support users with data visualizations that can help grasping data property and relationship emerging from the underling data collection. For this purposes aggregate representations of data are more expressive and more useful.

There are different families of visualizations, each one having its own characteristics, this being suitable to represent some kind of data, or data aggregations, for particular analyses.

In the following, we will categorize and describe the visualization families that we considered in order to represent aggregated data extracted from the sentiment Data Warehouse.

3.2.1 Charts

Charts are the most popular and used type of visualizations for data analytic. They provide representations of aggregated data and could be of different types:

- *Pie Chart*
  A pie chart is a circular chart divided into sectors, highlighting data proportion. In a pie chart, the arc length of each sector is proportional to the quantity it represents. When angles are measured with one turn as unit then a number of percent is identified with the same number of centiturns. The pie chart is perhaps the most ubiquitous statistical chart in the business world and the mass media. However, it is difficult to compare different sections of a given pie chart, or to compare data across different pie charts. Pie charts can be an effective, in particular if the intent is to compare the size of a slice with the whole pie, rather than comparing slices

among them. In general other plots such as the *bar chart* or the *scatter chart*, or non-graphical visualizations, such as *tables*, may be more suitable for representing certain information.

- *Scatter Chart*
  A scatter chart is a mathematical diagram using Cartesian coordinates to display values for two variables for a set of data. The data set is displayed as a collection of points, each having the value of one variable determining the position on the horizontal axis and the value of the other variable determining the position on the vertical axis. A scatter plot can suggest various kinds of correlations between variables with a certain confidence interval. Correlations may be positive (rising), negative (falling), or null (uncorrelated).
  One powerful aspect of a scatter plot, however, is its ability to show nonlinear relationships between variables. Furthermore, if the data is represented by a mixture model of simple relationships, these relationships will be visually evident as superimposed patterns.

- *Line Chart*
  A line chart is a type of graph which displays information as a series of data points connected by straight line segments or interpolated using spline. It is a basic type of chart common in many fields. It is an extension of a scatter graph, and is created by connecting a series of points that represent individual measurements with line segments. A line chart is often used to visualize a trend in data over intervals of time – a time series – thus the line is often drawn chronologically.

- *Area Chart*
  An area chart displays graphically quantitative data. It is based on the line chart. The area between axis and line are commonly emphasized with colors and textures. Commonly, this chart is used to compare two or more quantities.

- *Bar Chart*
  A bar chart, or histogram, is a chart with rectangular bars with lengths proportional to the values that they represent. Bars can be plotted both horizontally and vertically. Bar charts are used for plotting discrete (or 'discontinuous') data.

3.2.2 Tree Map

Treemapping is a method for displaying tree-structured data by using nested rectangles. Each branch of the tree is associated with a rectangle, which is then tiled with smaller rectangles representing sub-branches. A leaf node rectangle has an area proportional to a specified dimension on the data. Often the leaf nodes are colored to show a separate dimension of the data. When the color and size dimensions are correlated in some way with the tree structure, one can often easily see patterns that would be difficult to spot in other ways. A second advantage of treemaps is that, by construction, they make efficient use of space. As a result, they can clearly display several items on the screen simultaneously.

3.2.3 Tag Cloud

A tag cloud or word cloud (or weighted list in visual design) is a visual depiction of user-generated tags, or simply the word content of a site, typically used to describe the content of Web sites. Tags are usually single words and are normally listed alphabetically, and the importance of a tag is shown through font size or color. Thus, it is possible to find a tag alphabetically and by popularity. The tags are usually hyperlinks that lead to a collection of items that are associated with them. Sometimes, further visual properties are manipulated, such as the font color, intensity, or weight.

3.2.4 Table

A table is a matrix of data, and is especially used to represent relational data. This kind of visualization is not appropriated to represent aggregate values but to show the whole information. For example, if an analyst is examining the Web reputation, as in our case study, he would read a user comment about a particular topic to understand the causes of a negative judgement. Using a table it is possible to show such detailed information, which instead would not be shown through aggregate visualizations.

3.2.5 Interaction approaches to chart visualization

Web mashups users are people with different skills. If the user is used to conduct analyses on a Data Warehouse and is familiar with dimension and measures will prefer to define visualization for aggregate analyses a similar approach but if the user never use before a Data Warehouse he probably will prefer specify his analyses setting visualization property or preference. In order to meet these different habits we describe this two interaction approaches to set visualizations.

The data warehouse approach The users who are used to work with data visualization tools on a Data Warehouse may prefer a similar approach also in the mashup's composition. In this approach user will specify dimension and measures among the available ones.

The preference approach The user without Data Warehouse skills do not want to deal with measures and dimensions. He may want to define visual proprieties, e.g. data represented on x or y axes of a chart and what series has to be shown in the chart.

Chapter 4

Web Mashups: a new paradigm for end user development

One of the main problems in software development is to deliver products that meet users needs and preferences. In all the software development models the critical phase is the validation by the final users of the produced applications: as a matter of fact, if the users do not find the software acceptable, the development cycle needs to iterate until the users expectations are fulfilled. This of course is one of the main cost items in terms of time and money. Let us for example consider the needs of different professionals working in disparate areas, such as engineers, physicians, geologists and physicist, who are not professional programmers. In these domains, end-users best know their requirements and need situational and super vertical functionalities. Their productivity would increase if they would be empowered with tools for the easy production of their applications through lightweight development processes.

Traditionally, in the world of software development, there has been a clear separation between designers and consumers of the applications. In this context, accommodating the users needs is challenging. To overcome such difficulty, some development models (e.g., the Iterative life-cycle [39]) have been proposed in alternative to the traditional top-down Waterfall model. In the last years, with the revolution of Web 2.0, the Web also started to change the development practices: the development of modern applications is increasingly characterized by the involvement of end users with typically limited programming skills. The Web 2.0 in particular promotes the *culture of participation* [30], where the separation between designers and users is reduced, and end-users are more and more enabled to take part to the construction of contents and applications. *End-user development* is therefore gaining momentum as a paradigm through which end-users, by means of adequate tools adopting high-level abstractions, can flexibly define (or simply customize) their own applications.

Although powerful for increasing the user involvement, end-user development is not a panacea, and we cannot affirm that it provides a comprehensive solution for software development. For example, in the case of mashup development, we strongly believe that quality mashups can be achieved only if quality components, programmed by skilled programmers, are available. More in general, some development aspects are too complicated

to be handled without the intervention of expert programmers. However, our work aims at demonstrating that under certain assumptions it is possible to provide not skilled end-users with tools that, being based on components amenable to flexible integration in several contexts, can potentiate end-users programming.

In this chapter we define and discuss end-user development, trying to classify the frameworks so far proposed in literature to simplify the development and the programming of applications at different levels of complexity and abstraction. Mashup development is then discussed, to highlight its potential as a specific end-user development framework.

4.1 Classification of approaches

The literature [30] distinguishes among different end-user development approaches.

End-User Programming (EUP) empowers and supports end-users to program with techniques such as: *programming by demonstration*, *visual programming*, *scripting languages*, and *domain-specific languages*.

End-User Software Engineering (EUSE) adds to EUP support for systematic and disciplined activities for the whole software lifecycle including: reliability, efficiency, usability, and version control.

End-User Development (EUD) focuses on a broader set of developments, e.g. creating 3D models with *SketchUp*, modifying games. It puts end-users as owners of problems, and makes them independent of high-tech programming.

Meta-Design provides a framework and a design methodology to explicitly "design for designers", by defining contexts that allow end-users to create content; it is applicable to different contexts and encompasses principles that may apply to programming, software engineering, architecture, urban planning, education, interactive arts, and other design fields.

Our work is focused on the EUD: our mashup environment allows the end-user to solve different kinds of problems as owner, through lightweight development tasks which hide the technological complexity behind the high-level abstractions. In this way the user can compose his/her own applications with extreme flexibility without programming efforts.

4.2 The mashup development

Web mashups have the potential of supporting end-user development. In this section we therefore identify the main steps in the mashup development process, to highlight how adequate support, i.e., adequate development tools, can increase such potential.

4.2.1 The development scenario

We can distinguish two basic approaches for mashup development:

Manual approach: The end-user is programming-skilled and therefore is able to write code to program components and their choreography.

Automatic approach: The end-user is not programming-skilled and has only to integrate ready-to-use components that expert developers have previously programmed. S/he uses a tool that simplifies the composition of the mashup.

How mashups are developed depends on the type of mashup. While current consumer mashups (for example, all the numerous mashups based on Google Maps) are mainly the results of some hacking activities by expert developers, enterprise mashups highlight different development scenarios. Therefore we start from some recent studies on the enterprise mashup domain [43, 44] and quality of mashups [20]; we try to outline the different contributions of users at different skill levels:

1. Mashups can be used by *expert developers* (for example implementers of an IT department or service providers) to deliver applications quickly. End users are not directly involved in the construction of such mashups but benefit from the shorter turn-around time for new applications.

2. *Expert developers create services* that can be composed into mashups and provides a sandbox where end users can combine them themselves. The role of developers is to create services, but mashups are constructed by users closer to the application domain.

3. *Expert developers deploy a tool* that lets anyone create their own mashups. This is analogous to how spreadsheets are used in organizations today: end users (e.g., business analysts) can create spreadsheets without involvement from an IT department. These mashups are often created for a single purpose and user. They are indeed also known as situational applications [14].

Figure 4.1 illustrates the previous scenarios. The corresponding solutions differ in terms of the heterogeneity of the services that can be combined, the diversity of user needs that can be met, and the level of sophistication of either the user or the tools that support their work.

When mashups are developed centrally (scenario A), the resources for developing mashups are limited to the expert developers in the IT department. Given the limited resources of an IT department, only frequently requested applications will be developed. However, these developers have the skills required to integrate widely heterogeneous services and data at a low level of abstraction, allowing (essentially) any type of application to be built.

When IT focuses its resources to developing components and supporting users on how to use those components (scenario B), IT will expose certain types of data and services in a format that can be more easily consumed and combined by users who are not themselves developers. These users will have a better understanding of the application domain than

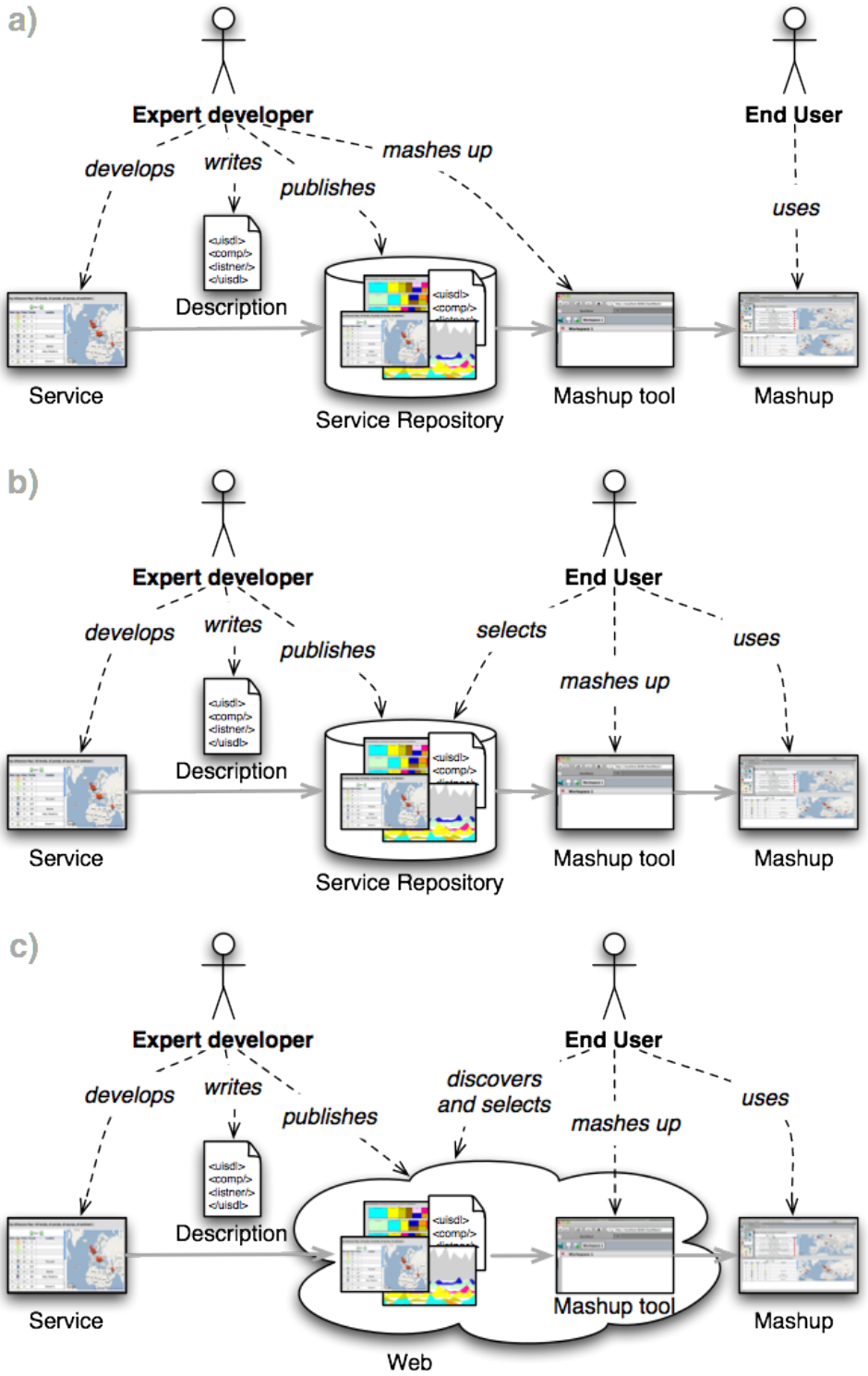


Figure 4.1: The mashup development scenarios [24, 25]

the developers in the IT department. As the resources for mashup development are not limited to IT staff, a larger diversity of user needs can be met in the second scenario. Yet, the type of mashups that can be built is still limited to the components that IT provides and will, generally, only enable a parameterization of components, as users lack the skills to perform deeper integration.

A tool for the creation of mashups (scenario C) will, initially, be the most challenging scenario to implement. However, it also provides the biggest pay-off. Using such tool, users can combine services and data to create their own mashups. The tool constrains what users can do, and, hence, gives warranties about the "composability" of mashup components. Users, however, are not limited in terms of the types of applications they can build: this scenario, therefore, supports the greatest diversity of user needs.

Another distinction between the different scenarios is the degree of control over the quality of mashups being created. In scenario A, the IT department fully controls what kind of mashups is being developed. Thus, IT ensures the quality of those mashups. However, not all mashups have stringent requirements in terms of security, performance, or reliability; they may only be used for a specific purpose, and a complex solution developed by the IT department would also be too costly. In scenario B, the IT department selects which components can be mashed up, and provides an environment for safely executing those mashups. Users can create mashups from those components to meet needs unanticipated or not served by the IT department. Mashups developed by users in scenario C may subsequently serve as prototypes for hardened applications developed by the IT department, should there be a need for the mashup to be exposed to many users within the enterprise, or if the mashup will be offered to outside users.

In von Hippel's terms [54], mashups "democratize" innovation, allowing end-users to meet their own needs, which a central IT department or, more in general, a service provider cannot always address. Their use also shortens the time by which users obtain the desired functionality. As described, one use of mashups is to prototype a solution to a problem faced by a specific user, and later generalize it to a larger user community. Mashup development is therefore similar to open source development (which was the source for von Hippel's metaphor) in two ways: the contributors to an open source project are also users of the software it produces; and open source projects provide a mechanism whereby contributors can progress from passive users to providers of feedback and feature requests and to code contributors. Similarly, mashup developers are often also users of those mashups (in scenarios B and C); and not all users of mashups need to be developers, but they can contribute to their development by providing feedback and feature requests, all the way to developing prototypes.

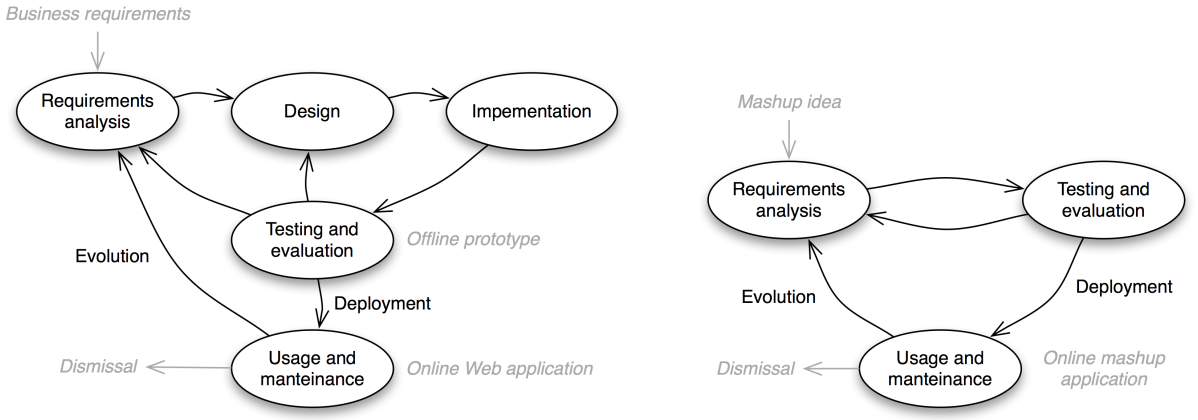
4.2.2 The lightweight development process

It is well recognized that the life cycle of Web applications is typically more dynamic than the one of other classes of software. First, development for the Web must be fast. Prototypes and a final application must be developed in "Internet time", that is, in days or weeks instead of months or years. Second, the possibility to log usage data of hundreds to millions of users opens the way for more advanced testing and usability

analyses. Third, once an application has been deployed, evolutions and improvements are applied while the application is actually online and used; in other words, an application undergoes continuous online evolutions. The development for the Web naturally spans over two main stages: the incremental development of the base version of the application and the post-deployment, incremental evolution.

Although dynamic and fast, this development process goes well beyond the requirements of mashup applications and the skills of average mashup composers. It is indeed oriented toward professional programmers that develop document-centric, data-intensive applications. Furthermore, it does not allow users to actively participate in the development and to innovate themselves.

The ideal mashup development process should reflect the innovation potential of mashups: to compose an application, starting from given contents and functionality responding to personal needs, and to simply run it, without worrying about what happens behind the scenes. The prototype-centric and iterative approach is accentuated: the composer just mashes up some open services and runs the result to check whether it works and responds to his needs. In case of unsatisfactory results, he fixes the problems and runs the mashup again. Given the situational nature of mashup applications, the role of application stakeholders must be put into perspective: requirements indeed correspond to the (short-lived) needs of the mashup composer that lead to the mashup idea. Also, design and implementation and testing and evaluation have a different flavour, and should be somehow simplified (or even hidden) through the use of adequate tool-kits.



(a) The life cycle model of current Web applications

(b) The mashup life cycle model

Figure 4.2: Comparison between the traditional life-cycle for web applications and the lightweight development process for mashups

These considerations can be summarized in form of the lightweight development process model shown in Figure 4.2(b), characterized by three main activities:

- *Discovery and selection*: Starting from an initial mashup idea, generally reflecting personal needs and preferences, the mashup composer will select suitable source services providing the necessary data, application logic, or user interfaces, which in most cases are open services available on the Web. This is a new and highly

specific activity for mashup applications, which precedes the creation of the mashup and implicitly incorporates requirements analysis and specification. The initial mashup idea can be indeed considered an informal expression of the application requirements. The selected mashup components provide a representation of these requirements in terms of enabling services, which proves the feasibility of the original idea and provides an initial draft of the organization of the final mashup. Either in the tool-assisted approach or in the manual approach, the services or the APIs, which will be used in the composition, have to be wrapped into a component that is the atomic – black-box – part of a mashup. This component creation task must be done by a programming skilled person – user or tool administrator – because who creates a wrapper has to deal with protocols and programming aspects. When end-user uses an automatic composition tool, this phase is done by programmers or tool's administrator which wrap all the services that will be used in the tool. Who wraps API or services for an automatic tool has probably also to describe the component to adapt it to an automatic environment that needs some description, e.g. to manage component bindings.

- *Mashup composition*: Dedicated mashup tools will enable also the less skilled Web user to graphically compose the selected components and to set up the necessary integration logic and the layout of the composite application. The actual integration logic will be based on intuitive formalisms and models and expressed via simple, domain-specific languages, which in most cases will however be hidden behind the graphical modelling tools. This activity provides a new flavor for the traditional design and implementation activities and, in particular, eliminates the need for cornerstone activities, such as hypertext design, that for long time have characterized the development of document-centric Web applications. The deployment of the mashup application will then be performed by simply saving it on a server for the hosted execution (one-click activity). The composer, in both the approaches, has to create bindings among the available components. In the manual approach he/she has to manually write code to embed and synchronize the components. In the automatic approach it is the composition tool that crates the binding writing code and the composer has only to interact with it doing simple operations, i.e., drag-and-drop or selection in menus.

- *Usage and maintenance*: Once saved on the hosted execution server, the mashup will immediately be ready for use. The maintenance of the application will be shared among the mashup composer (e.g., he will fix problems in the composition logic) and the provider of the mashup platform (e.g., he will fix problems in components and the hosted execution environment). This activity incorporates the former test and evaluation and usage and maintenance phases. By running the mashup, the composer can easily check whether the application works and responds to his needs, or he can also collect feedback from other users. The necessary evolution of the application then simply requires starting the mashup process anew from discovery and selection. In this phase end-users could be helped by a recommendation system based also on the components and composition quality and on the compatibility among components.

The described lightweight process model is conditioned to the availability of a tool-kit made of a dedicated, hosted mashup platform and a set of open services available on the Web that already provide a high value in terms of supported features and available data. The reuse of ready-to-use services is common to other component-oriented development practices. However, the adoption of open services easily accessible through the Web is peculiar of mashups and, if complemented with adequate development platforms, can enable end users to innovate. Also, the typical environments for component-oriented programming target expert programmers, who might develop even better mashups by writing code. Yet, they go far beyond the capabilities of average Web users or employees.

Finally, in order to eliminate the deployment task, which would be out of the capabilities of Web users, we assume mashup platforms will provide support for hosted solutions for both development and execution (partly this is already practice). As a consequence, we can say that mashups are applications whose life cycle naturally starts from the deployment point in Figure 4.2(a) and whose development occurs via incremental evolutions: indeed, once saved, mashups are immediately online, and there are no incremental development cycles.

4.3 The need for mashup tools enforcing EUD

As described in the previous section, the way to compose Web mashups could be positioned at different levels, characterized by different practices and programming efforts. Expert Web users generally prefer to compose their application manually, writing JavaScript, HTML, PHP, JSP or code for other languages for Web development, and integrating APIs and services directly in the page code. However, if users are programming-unskilled, they cannot write code. If a tool simplifying the composition is not available, then they have to deal with programming and – given the heterogeneity of components, programming languages, interaction protocols, the complexity of the necessary integration logic, and similar – this is an option only for highly skilled programmers. In conclusion, manual development is out of the reach of Web users.

In line with the end-user development vision, enabling a larger class of users (not only skilled developers) to compose their own mashups and to innovate requires the availability of intuitive and easy development tools and a high level of assistance [19]. As already introduced in Chapter 2, there is a considerable body of research on mashup tools typically featuring easy-to-use graphical user interfaces and drag-and-drop paradigms for combining mashup components. However, to the best of our knowledge, none of these tools provides an integrated development paradigm, instrument, and language easing the integration of heterogeneous components. This implies that each tool is suited only for some development tasks. None of these tools supports integration at all the three layers that characterize Web applications, i.e., data, application, and presentation (UI) logics (see Chapter 2). Also rarely the offered composition paradigm meets the skills and the expectations of non expert users.

The development of an environment that natively covers the previous requirements is the object of our work thesis, which in the end has produced an environment supporting

the agile, mashup-based development of Web applications. To ease the composition task, the environment is equipped with an AJAX-based visual editor, enabling the visual (i.e., based on drag-and-drop) composition of services. The composition logic is event-based (to cater for the needs of synchronizing the UI state of each component) and data-flow-based (to take into account the needs of service orchestration). As better described in the following chapters, such technical details are however masked by the paradigm for the visual composition of the components integration, and especially by some abstractions that allow translating actions of the composition paradigm into intermediate descriptive models driving the execution logics.

Part III

Models and Architecture

Chapter 5

Models

The design of the mashup platform developed within the work of this thesis has been driven by the need to alleviate as much as possible the mashup composition task. We in particular tried to strengthen some key EUD factors [13, 30], which also emerged from the analysis of user requirements conducted for the development of our case study application:

- *Abstraction from technical details*: as also observed in a recent user centric study [13], the representation of services as visual objects that abstract from technical details (e.g., their programmatic interface), the immediate feedback on composition action, and the immediate execution of the resulting mashup to reveal the service look& feel, help users realize the service functionality and the effect that the service has on the overall composition. As described in Chapter 6 and 7,in DashMash, users are asked to manipulate (e.g., add, remove or modify) visual objects focusing on the service visualization properties rather than technical details of service and composition logic. As also confirmed by our experimentation (see Chapter 8), this increases user satisfaction and, in particular, the user-perceived control over the composition process.

- *Composition support*: the composition task is guided in multiple ways. On the one hand, the composition engine creates *default bindings* among services included in the mashup composition, which however users can easily modify and extend. Default bindings are defined on the basis of compatibility rules that the composition engine automatically infers from each component's descriptive models. The same compatibility rules are also adopted to "rank" available components, thus providing suggestions for additional bindings.

- *Continuous monitoring*: users are provided with mechanisms that allow them to understand the current state of the composition and to explore options about how to complete or extend the current composition.

Given the previous requirements to potentiate EUD, the organization of the platform has been centered around a lightweight paradigm in which the orchestration of some registered services, the so called *components*, is handled by an intermediary framework, in charge of managing both the *definition* of the mashup composition, by means of a visual com-

position mechanism, and the *execution* of the composition itself. One peculiarity of our approach is that, differently from the majority of mashup platforms, where mashup design is separate from mashup execution, in DashMash the two phases strictly interweave. The result is that high-level composition actions, executed by users through the visual composition environment, are "automatically" translated into intermediate models describing the composition; these models are immediately interpreted and executed. Users are therefore able to *interactively* and *iteratively* define and try their composition, without being forced to manage complicated languages or even *ad-hoc* visual notations.

Such a composition and execution paradigm is possible thanks to some abstractions supporting the definition of an intermediate description level that is: i) close to, thus easily derivable from, the lightweight concepts that the users exploit to compose the mashups; ii) amenable to the rigorous interpretation of the engine managing the mashup execution. While Chapter 6 illustrates the architectural component and the mechanisms behind the construction and execution of the mashup, this Chapter is devoted to clarifying the abstractions and the corresponding models at the basis of our approach.

Usually, a component uses an API or a Web service to retrieve its own dataset. However there could be the necessity to use a proprietary database for the component visualization, for example to use domain specific data as it happens for visualization components (e.g services providers visualizations). The Data Flow Model has been introduced to represent the most relevant proprietaries data.

5.1 Component model

The need for an abstract model for components requires the use of a component descriptor that has not to be tied to specific implementation technologies and, at the same time, has to be sufficiently powerful to describe the component characteristics in terms of interaction and properties. The model must also capture the component properties that potentiate the adopted composition and execution paradigm.

Our platform is based on the integration at presentation level. Conceptually, if a mashup must be managed at UI level, a component can be characterized by a *state*, which defines what the composite application can see and control in terms of changes to the UI. The state can be complex and consist of multiple attributes (e.g., map location and zoom level). A set of events allow notification of state changes, while operations allow for querying and modifications of the state.

In addition, presentation components typically have configuration parameters that reflect UI appearances, such as font face and background color. Parameters are specified at design time (or component creation time) and exposed via a set of properties.

In general, the attributes of the component state are high level and conceptual (e.g., location and zoom level), while configuration parameters are related to preset graphical attributes (font faces, background colors, etc).

However it is up to the component developer to define which characteristics are part of

the state and which characteristics are configuration parameters. Ideally, the state should be kept as simple as possible to facilitate integration and reuse, as state changes are what cause events to be exchanged among components and therefore need to be handled in the composite.

The external interface (implementing the component model) of a presentation component consists of a set of events, operations, and properties, which allow the component to expose its state and configuration parameters.

A presentation component may expose:

- **Events**: a component may expose a set of events to notify other components of its state changes, which are initiated either by user actions on the UI, by requests from other components, or by its internal logic. Note that our component model is only concerned with component-defined events, not native UI events defined by the underlying UI toolkit. (E.g. If the user drags the mouse to change the viewpoint, we are not interested in the click and drag operation, but in the state change implied (viewpoint changed). Essentially, user actions trigger both native UI events and component-defined events. However, native UI events are captured by the underlying UI toolkit and processed by the components internally, whereas component-defined events, which signal state changes, are exposed externally. It is up to the component (or the component's wrapper) to define and implement the relationship between native UI events and component events that signal state changes.
- **Operations**: a component can expose a set of operations that allows for queries and modifications of its state. An operation typically supports a list of input parameters which allows the caller to pass in values, and a return value which allows the caller to retrieve the result. The support for multiple input values allows an operation to set an attribute of the component state with various options, or even to set multiple attributes of the state at the same time.
- **Properties**: at component creation time, properties can be used to expose the initial state and the configuration parameters of the component. For example, properties allow the design-time customization of the map component's configuration parameters such as font face and background color, and initial state such as the default map location.

5.1.1 UISDL - The UI Service Description Language

In order to represent the component properties described above, we have adopted and re-elaborated the UISDL specification of UI components already defined in *Mixup* [23]. UISDL is a XML language for the abstract description of the component properties according to the abstract model previously described.

UISDL container: `<uisdl>`

It is the root of the XML element in a UISDL document. It's composed of the following attributes:

- *Language namespace*: http://www.openxup.org/2006/08/xpil/component
- *xmlns:tns*: this attribute contains the component namespace reference. The value is an URI that can univocaly identicate it. It is unnecessary that it points to a concrete resource.

```
<uisdl xmlns="http://www.openxup.org/2006/08/xpil/component"
        xmlns:tns="http://namespace-for-the-target-application">
        ...
        <!-- components -->
        ...
</uisdl>
```

Listing 1: <uisdl> example

Component: <component>

This element models a presentation component. It could contain a description, a list of events, a list of operations or a list of properties.

It is composed of the following attributes:

- *Id*: it specifies the identifier of this components.
- *Address*: it specifies the component position as relative path. It can point to any strings that allows to the adapter to localize the component and instantiate it.

```
...
<component id="GoogleMaps" address="./GoogleMaps/GoogleMaps.js">
        ...
        <!-- operations and events -->
        ...
</component>
...
```

Listing 2: <component> example

Operations: <operation>

This element specifies an operation that can be executed by a component. Operations allow a status change through the middleware runtime.

The <operations> element is characterized by the attribute:

- *Name*: it specifies the name of the operation.
- *Address*: this is the reference string to the operation implementation in the component wrapper.
- *Type*: data type of the parameter. The value is a XML Schema type and it can be a data type defined by XML Schema as well as a more complex type defined in <types> element.

The element <operation> could contain <input> elements, that specifies operation input parameters, that in turn contain the attribute "element" (it specifies the element name) and the attribute "type" (the value is a XML Schema type and it can be a data type defined by XML Schema).

```
...
<operation name="showPosition" address="showPosition">
        <input element="locality" type="xsd:string" />
</operation>
...
```

Listing 3: <operation> example

Events: <event>

This element specifies an event that can be launched by a component. Events permit the notification of a component status change.

The principal attribute is: *Name* which defines name of the event; it allows other components to identify it and listen for it. The value has an identifying target and has to be unique between other events of the same component.

The <event> element contains the tag <description>, that provides a brief description useful for an end user to identify the semantics of the component's event. It could also include parameters, <param>, that specifies the data associated to the event. A parameter is defined by:

- *Name*: name of the parameter.
- *Type*: data type of the parameter. The value is a XML Schema type and it can be a data type defined by XML Schema as well as a more complex type defined in <types> element.

```
...
<event name="onPositionChange" description="Changing position event">
        <param name="localityName" type="xsd:string" />
</event>
...
```

Listing 4: <event> example

Properties: <property> This element specifies the properties of an element. They are used for: species the initial status of the component and its graphic features at creation; query the component status and its interface features at runtime.

It is characterized by these attributes:

- *Name*: property name. It has to be unique in the component.
- *Type*: species the data type of the property. The value is a XML Schema type and it can be a data type defined by XML Schema as well as a more complex type defined in <types> element.

- *Required*: specifies if the value of the property has to be specified in the composition document.

Types: <types> This element contains a list of data types XML Schema defined for events, operations and properties parameters.

Moreover, it is possible to extend the component description, adding annotations to represent values about attributes on qualities and component characteristics. Those annotations are then used, as described in Section 6.2, 6.5.1 and 6.5.3, to calculate aggregated quality and to make recommendation.

5.2 Composition Model

The aim of the composition model is to declare the components used in the composition, how they interact and how they are synchronized. The composition model adopted in DashMash also derives from the *Mixup* environment. It mainly includes *event subscription* information in order to facilitate the communication among presentation components, where each event corresponds to a user or system action that changes the state of a component[1]. Before introducing the XML-based format for specifying the component model, in the next Section we first illustrate the event-driven composition logics that characterizes our approach.

5.2.1 Event-driven composition

As better illustrated in Chapter 6, our mashup paradigm requires that components exchange events through an *event broker* that facilitates loose coupling. The composition model supports a "one-to-many" publisher/subscriber relationships among presentation components. So, through this mechanism, a component publishes an event and the other components subscribe to it (i.e., to declare that they will listen to and handle this event).

The publisher/subscriber relationship can be specified via *event listeners*. Each listener species an *event publisher*, *event subscriber*, and an *operation* of the subscribing component. In addition, multiple event listeners can be used to support multiple event subscribers for a single event from the event publisher. We stress that, in order to facilitate loose coupling, event listeners are elements belonging to the composition model, not to the component model of the subscribing components.

As illustrated in Figure 5.1, as soon as events occur, the involved components publish them. The occurrence of events causes a state change in the subscribed components. Based on this paradigm, the mashup execution engine plays the role of a controller in an MVC pattern [31], where the model corresponds to the application logic of mashup components and the view relates to the presentation layer of each component and of

[1]If needed, it allows additional integration logics to be specified within event listeners, in the form of simple inline scripts or references to external code.

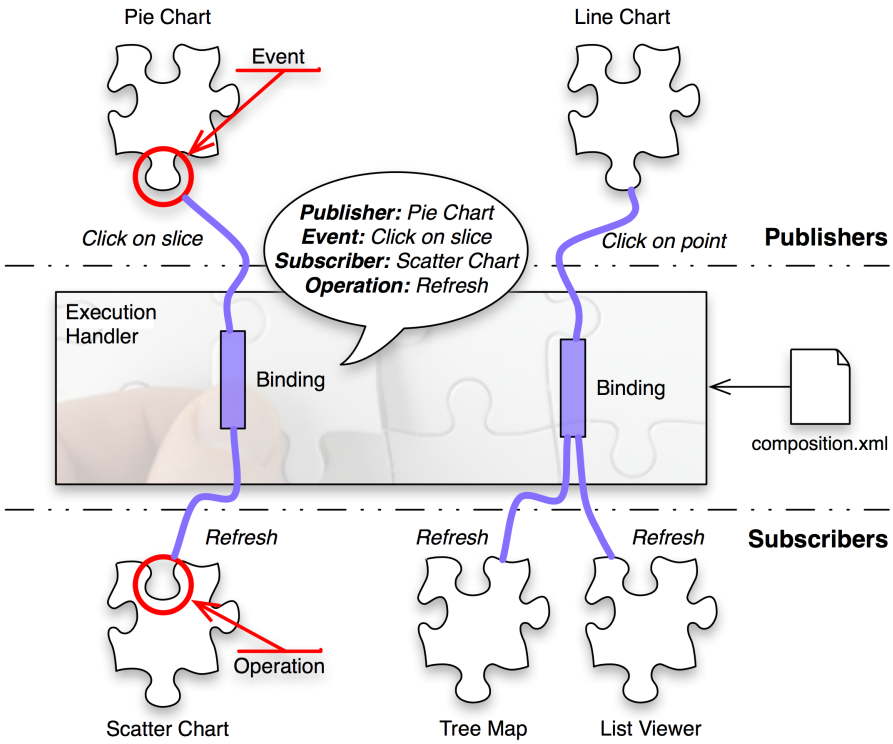


Figure 5.1: Event-driven paradigm for service binding definition and service synchronization

the overall mashup. Each component keeps running according to its own application logic.

5.2.2 XPIL - eXtensible Presentation Integration Language

The composition elements previously described can be specified in a declarative composition language, the eXtensible Presentation Integration Language (XPIL) [26]. Such language contains two sets of XML elements: the first declares the components in use, and the second describes the composition model.

Xpil container: <xpil>

It is the root element of XPIL document. It has the following attributes:

- **Language namespace**: http://www.openxup.org/2006/08/xpil/integration
- **xmlns:tns**: this attribute contain the component namespace reference. The value is an URI that can univocaly identified it.

Components: <component>

This element points to a presentation layer component. It is defined by these attributes:

- **Id**: it specifies a unique id for the component in the XPIL document.
- **Ref**: it specifies the localization of an XPIL or UISDL document. the value could be a le path, an URL, and it can be a relative or absolute reference. If the value is referred to an XPIL document, the component is a made-up component, but it will be treated as a single component in the composition. The <component> element can contain a list of properties.

```
...
<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"/>
...
```

Listing 5: <component> example

Event listeners <listener>

This element specifies an event listener that links a component event to an operation of another component. It is characterized by these attributes:

- **Id**: it specifies a unique id for the listener in the XPIL document.
- **Publisher**: it contains the id of the source component that launches the event.
- **Event**: it specifies the name of the event. The value refers to the "name" attribute of the <event> tag in the component descriptor. The combination between this value and the attribute "publisher" identifies the event.

- **Subscriber**: it contains the id of the component that has to run an operation because of the event launched.
- **Operation**: it specifies the name of the operation that it will be invoked after the event notification reception. The element could contain XSLT/XQuery for data transformation and conversion. Moreover, it could contain script or references to external code to add additional integration logic.

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

Listing 6: <listener> example

5.3 State Model

XPIL provides supports the representation of composition schema by showing which components are involved in the composition and how they interact (through pairs <event-publisher, operation-subscriber>). Moreover, XPIL provides formal description of events and event parameters. However, these elements could become insufficient if we have to describe an "instance" of a mashup at execution time. In that case, in fact, we must trace each parameter is change to user interaction and component state change.

Through the definition of a State Model, we enrich the composition description to introduce new capabilities. In the following list, we present the advantages derived by the introduction of a State Model.

- **Real time composition and execution**: in the tool we developed (see Chapters 6 and 7) a user can add (or remove) a component to the composition with a drag and drop interaction. The addition/removal of components causes the automatic insertion/removal in an XPIL descriptor. Then, we need to re-instantiate the composition. To keep the previous components configuration parameters that results from the previous user interactions, we can analyse the State descriptor to restore them after reloading the composition.
- **Restore a customized composition**: combining XPIL and state description permits to save and to reload a composition in terms of involved components, bindings between them and user customization (caused by state change interaction effects between components or from user). Moreover, it is also possible to create an history function exploiting the state description.

- **Data selection and management**: state description can be further useful if we associate a proprietary database. As we show in the following Chapter, state descriptor permits to analyse user settings, in terms of filtering data or selecting data sources, to automatically generate query on databases. Moreover, if components support data chunking, we can use this descriptor to store, for example, the current page number in a text viewer component.

- **Quality and recommendation**: through state description analysis, it is also possible (as it is currently under consideration) to define mechanism of recommendations and quality analyses in order to help users to create their composition. This mechanism, at state level, should be focused on component configuration parameters.

Through the state model, it is possible to provide a distinction between customized typologies of components. For example, driven by the requirements of our case study application, in DashMash we identify three typologies of components:

- **Filters**: these components are characterized by functionalities of data sources selection, both proprietary or public, and their filtering. We classify as filters components such as tag/keyword filters, time filters, etc.

- **Viewers**: viewers are components that do not have their own data sources, but only provide a presentation layer with a little logic of data presentation and aggregation, but without a proper individual meaning (e.g. a pure graphic component that provides data visualization in a pie chart form, but that needs other components data or external data sources to be useful/meaningful)

- **Generic components**: these are generic components that typically have individual meaning and significant functionalities.

To better explain the difference between viewers and generic components, we provide an example of different uses of Google Maps service. On the one hand, we classify as generic component, a component that provides an interface to Google Maps service as generic geographic map viewer; on the other hand, we classify as viewer a component that uses Google Maps to show a map with a data visualization (i.e. , markers) extracted by a proprietary database (as in our case of study, emphthe Top influencer Map 7.3.8 in which there is a map with some flags and a data table.)

It is worth noting that component classification depends on the specific domain addressed by the mashups to be built. However, some classifications are in general valid for entire classes of mashups; also, some concepts are reusable across different classes. For example, data sources and viewers are particularly effective in the domain of mashup-based dashboard construction; this is indeed the domain addressed by our case study. This classification can be however adopted in any case where some *ad-hoc* viewers are used to visualize data coming from any data source (local or remote).

The component typization exploited for the construction of the state model is, at the moment, statically defined, but it is being developed a configuration tool through which users can declare and define relevant component classes.

5.3.1 SMDL - State Model Descriptor Language

In line with the descriptive approach adopted for the other models, the State Model also makes use of a XML-based specification that consists of the elements described in the following.

State container: <state> It is the root element of State document.

ViewerBox element: <viewerbox> A viewerbox is a workspace in which a user can create a composition with a drag and drop interaction to add components, as it will be deeply presented in Chapters 6 and 7. A viewerbox is characterized by the following attributes:

- **Id**: it specifies an unique id for the viewerbox in the state document.
- **Name**: this attribute stores a name assigned by the user.

```
<state>
        ...
        <viewerBox id="viewerBox2" name="Mashup composition market">
        ...
        <viewerBox id="viewerBox5" name="Traffic">
        ...
</state>
```

Listing 7: <viewerbox> example

A viewerbox contains generic <components>, but also domain-specific classes of components, like <filters>, <viewers> in Dashmash.

Generic component element: <component>

This element points to a generic component. It is defined by these attributes:

- **Id**: it specifies an unique id for the viewer in the state document.
- **Type**: it classifies the type of the viewer, assigned from the platform (The considerations, previously made about typification, are still available).

```
...
<components>
        <component id="viewerBox4_RSSreader2" type="newsRSS">
                <preference name="currentPage" value="3" />
                <preference name="font" value="verdana" />
                <preference name="rowPreference" value="20" />
                <preference name="colorCombination" value="RedYellowGreen" />
                ...
        </component>
        ...
</components>
...
```

Listing 8: <component> example

Filter element: <filter>

This element points to a filter component. It is defined by these attributes:

- **Id**: it specifies an unique id for the filter in the state document.
- **Type**: it classifies the type of the filter, deriving from the user selection od filters and assigned from the platform (i.e. tags, time, source,...).
- **Value**: this stores the value of the filter (the keyword for a tag filter, the date selection,...).

```
...
<filters>
        <filter id="filterSource1" type="Source" value="Twitter Search" />
        <filter id="filterTag2" type="Tag" value="General" />
        <filter id="filterTag3" type="Tag" value="Arts\&amp;Culture" />
        ...
</filters>
...
```

Listing 9: <filter> example

Viewer element: <viewer>

This element points to a viewer component. It is defined by these attributes:

- **Id**: it specifies an unique id for the viewer in the state document.
- **Type**: it classifies the type of the viewer, assigned from the platform (The considerations, previously made about typification, are still available).

```
...
<viewers>
        <viewer id="viewerBox1_viewerTimeVolumeHC1" type="TimeVolumeHC">
                <preference name="series" value="brand" />
                <preference name="domain" value="date" />
                <preference name="range" value="volume" />
                ...
        </viewer>
        ...
</viewers>
...
```

Listing 10: <viewer> example

Both viewers and filters contains one or more <preference> elements that are characterized by these attributes:

- **Name**: it specifies the name of the preference (i.e series, domain, range,..).
- **Value**: it stores the value of the preference.

5.4 Data Flow Model

A mashup component commonly accesses a service (public API or in general any local or remote service) invoked through the component wrapper. Let us consider for example *Google Maps*, the most famous and used API in the Web; when a user makes an operation that needs to retrieve data, i.e., a zoom-in, the component that wraps *Google Maps* has to send a request to the service provider communicating the new parameters in order to cause the map redrawing. In the same way, all the components send requests in order to obtain new data according with the user actions. Each component manages by its own the requests, because in general there are not common data sources among them.

In general, when the mashup component is derived from a "public" service available on the Web, the format of the data retrieved and exchanged with the other components in the same mashup depends on the specific requirements of the component; thus, it is difficult to define and expect a common data format. However, as better explained in Chapter 6, in the case of domain-specific components that share the access to a common local data source, e.g., a company data warehouse, it is possible to "normalize" the data accesses through a unique component, called *DataService*, that will send request to the proprietary server from which data are taken are placed. In this case, it is therefore possible to define a common data format to describe the retrieved data. The main advantage is that such a description can facilitate the interpretation of data by the components in charge of visualizing them, i.e., viewers.

5.4.1 The result set representation

The results are represented, as previously described, with a XML file that is passed from the server to the client. In the following we describe the structure of the XML file and provide an example of a result set file.

Structure

Figure 5.2 and 11 respectively report the DTD and an example of the result set descriptor that we have adopted within our case study application, where viewers are in charge of visualizing data coming from a local data source. The result set is indeed aggregated by viewer boxes and viewers.

Figure: The result set dtd

```
<!ELEMENT dataset (viewerbox+)>
<!ELEMENT viewerbox (viewer+)>
<!ELEMENT viewer (series*)>
<!ELEMENT series (item*)>
<!ELEMENT item (domain, range*)>
<!ELEMENT domain (#PCDATA)>
<!ELEMENT range (#PCDATA)>

<!ATTLIST viewerbox id      CDATA   #REQUIRED>

<!ATTLIST viewer    id      CDATA   #REQUIRED>

<!ATTLIST series    name    CDATA   #REQUIRED
                    type    CDATA   #REQUIRED>

<!ATTLIST domain    type    CDATA   #REQUIRED>

<!ATTLIST range     type    CDATA   #REQUIRED>
```

Figure 5.2: The result set DTD

For each viewer, the results are aggregated by series. For each series there is a list of items with their domain and range. This structure allows us to describe data that will be visualized by a chart by using domain and range. With this solution we can describe data when they have to be visualized through a table or other kinds of visualization.

Let us consider a table: it has only a series of data and a domain, which represents the table header, but it has more ranges for each item. Each item is a row in the table and each range is a cell in the row.
Moreover, if we want to represent data with a tag cloud that shows, for each tag in the cloud, information on quantity and quality of a tag, the quantity could be represented by the dimension of the tag and the quality by its colour. Our result set model can be used also to meet this need: each series in the result set will be drawn in a separate tag cloud or in a different part of the tag cloud creating a cluster; the domain of the items will be the tag and the ranges will describe the quantity and the quality which are to be associate to a tag.

To our case study (Chapter 7) we will describe an implementation of DashMash that has different charts as viewers, each of them needing a result set, as described before, to be drawn.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <viewerbox id="viewerBox1">
    <viewer id="viewerBox1_viewerTimeVolumeHC1">
      <series type="brand" name="London">
        <item>
          <domain type="date">2010-06-13</domain>
          <range type="volume">35</range>
        </item>
        <item>
          <domain type="date">2010-06-14</domain>
          <range type="volume">27</range>
        </item>
      </series>
      <series type="brand" name="Milan">
        <item>
          <domain type="date">2010-06-13</domain>
          <range type="volume">18</range>
        </item>
        <item>
          <domain type="date">2010-06-14</domain>
          <range type="volume">19</range>
        </item>
      </series>
    </viewer>
  </viewerbox>
</dataset>
```

Listing 11: `A result set XML file example`

Chapter 6

Architecture

Based on the considerations and requirements highlighted in the previous chapter, we have implemented the framework to facilitate presentation integration, backed by the intuitive yet strong conceptual model for mashup composition and execution illustrated in the previous chapter. Figure 6.1 describes the overall architecture of the proposed framework for the execution of composite applications. A composite application to be executed by the platform consists of:

- One or more *components*, i.e., services described according to the UISDL component model and suitably wrapped for facilitating their invocation during the mashup execution. Component descriptors and wrappers are stored in a *component registry*;
- A specification of the *composition model* (i.e., the integration logics that coordinates the components at runtime);
- A middleware in charge of managing the event-driven execution of the composition based on a publish-subscribe paradigm.

Users defines the mashup through a visual front-end. Users can select relevant components, to include them into the composition, and combine them for synchronization purposes, by defining listeners through a simple, intuitive visual paradigm.

This chapter will present the main modules of the outlined framework, namely:

- The middleware in charge of managing the *event-driven execution* of the mashups;
- The mechanisms supporting the *visual composition* of the mashup, which are characterized by:
  - The *automatic generation of the composition model*, also covering the optional addition of some *default bindings* ensuring a minimal synchronization among included components, especially in case of domain-specific components;
  - The *immediate execution of the mashup*, without the need of a design phase preceding the mashup execution, and with the possibility for the users of getting immediate feedback about the results of the composition actions;

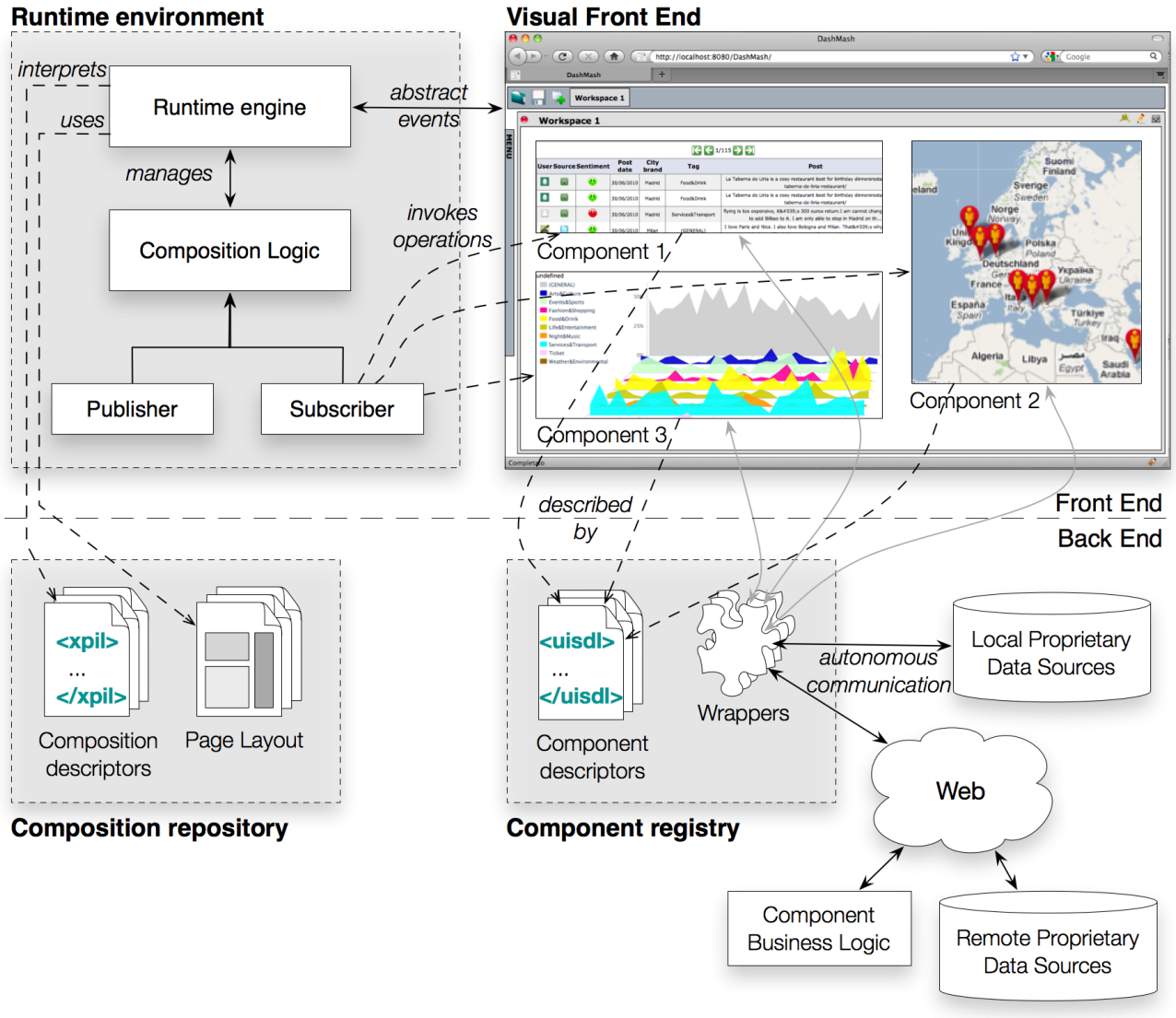


Figure 6.1: Logical view of the environment

  - The production of *quality-based recommendations* guiding the users in the choice of components and in the definition of additional component bindings, which ensure the production of correct mashups with an increased quality level.

- The modules and mechanisms enabling the integration of *proprietary data sources*, which implies some assumptions about the structure of the data flow and the representation of the exchanged result set.

6.1 General overview

Before describing in more details the architectural elements, we here provide a general overview of the behavior of the framework and the steps involved in the definition and execution of a composite application.

As illustrated in Figure 6.1, the composition and execution environment is a Web front-end. A *composition repository* contains an HTML page that serves as a layout for the composite application. All the style information related to the positioning of the components are defined within this file. The basic setup consists of a `<div>` element for each component. Additionally, the code for the framework initialization and reference for every other piece of code needed (e.g., components wrappers) must be placed in the `<head>` section.

At the beginning of the user session, an initialization script prepares the execution environment for the instantiation of the components. It initializes the composition descriptor namely XPIL composition and SMDL status descriptor files, and instantiates a basic composition that involves all the services already declared into the two descriptors, if any. As soon as users add components into the mashup, the framework adds pointers to the added components in the XPIL descriptor, and stores the parameters for components configuration and invocation in the SMDL descriptor. The translation of user actions into proper elements of the two descriptors is handled by the *runtime engine*. During composition, such module also provides support for the immediate visualization of added components and for the production of recommendations both for component selection and binding definition.

For each component declared in the XPIL, a corresponding "constructor" is called. The constructor can be either part of the component itself, as in the case of native components ad-hoc built for this environment, or a function in the *component wrapper*, if it is an existing Web component and some kind of adapter is used to make it usable within our mashup environment. The component constructor is responsible for the initialization and the rendering of the component in the `<div>` dynamically added into the page layout, which is identified with the same "id" value as the "id" of the component instance in the XPIL.

After the components identification and initialization is done, an XPIL parser proceeds to analyze the <`listener`> elements. For each listener, the relation *<(publisher, event)-(subscriber, operation)>* is saved in a suitable data structure (a chain of associative arrays)

for easy reference at execution time. The data structure is shown in Figure 6.2.

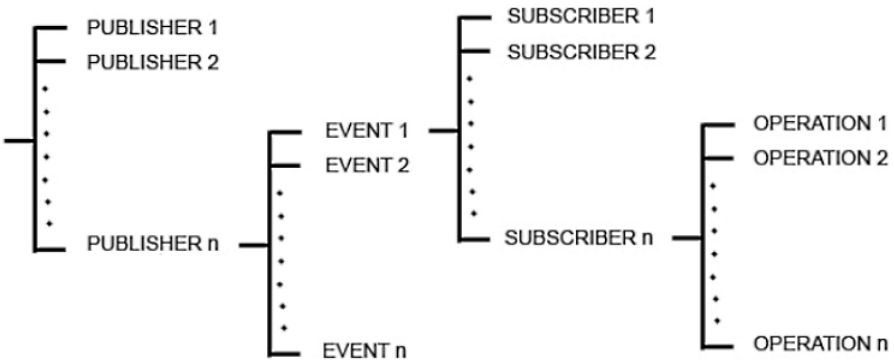


Figure 6.2: Data structure (chain of associative arrays) that represent the composition listeners

At this stage, every component should be running its own business logic and drawing its own interface. The framework module named *"runtime engine"* in Figure 6.1 waits for function calls notifying events from a component. As soon as events occur, the runtime engine intercept them and, based on the listeners defined in the XPIL, notifies the subscribed components, if any, and triggers the execution of their corresponding operations.

6.2 The runtime engine

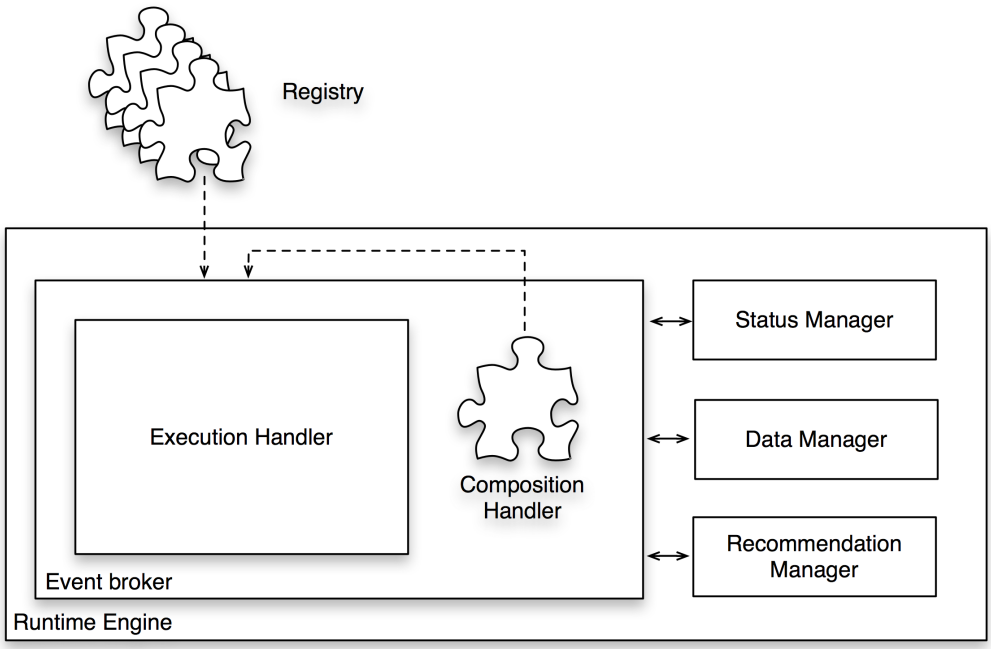


Figure 6.3: Main modules of the runtime engine

As represented in Figure 6.3, the runtime engine basically plays the role of an event broker: it intercepts events that occur during the definition of the mashup composition by users and during the mashup execution. Depending on their nature, intercepted events are then dispatched to the modules in charge of their handling, namely the *Execution Handler* and the *Composition Handler*.

As shown in Figure 6.3, besides these fundamental modules, the runtime engine also exploits some other components:

- The *Status Manager* aims at managing the SMDL-based status descriptor, which has a primary importance for enabling immediate execution and the recovery of previously defined mashups. Accordingly, it is necessary to keep it updated after every change of components properties, or after the addition/removal of a component. Modifications can be related to default or specific parameter values (e.g., the value of a parameter for querying a data source), to layout properties (e.g., the colors used to show values on a chart) or to any other property that the user can set to control the component status, contents and appearance.
- The *Data Manager* handles a data structure, the *Data Buffer*, which stores structured (i.e., complex) data that mashup components exchange for synchronization purposes. When such data are extracted from a local data source, e.g., a company warehouse, through the Data Manager intermediation we also guarantee a mutual exclusion during data access and modification. As better illustrated in Section 6.6.1, a specific component, the Data Service, is in charge of accessing a local data source and storing the retrieved data into the Data Buffer.
- The *Recommendation Manager* supports mechanisms for producing quality-based recommendation during the definition of components bindings. It computes quality indexes starting from quality annotations in the UISDL descriptors, that are defined taking into account a component quality model [20]. The Recommendation Manger evaluates also the syntactic and semantic compatibility between components. As described in [10], the goal is to guide the user in the selection of quality components.

The following sections will illustrate the functionalities that are supported by the previews architectural modules.

6.3 Event-driven execution: the *Execution Handler*

The *Execution Handler* is the functionality group that manages the execution of the mashup. In particular, it is in charge of interpreting the XPIL descriptor, to identify the components involved and the publish-subscribe bindings defined among them, and to instantiate the composition specified in the XPIL descriptor. During the mashup execution, it plays the role of an event broker: each component keeps running according to its own application logic, within the scope defined by an HTML `<div>`.

When an event occurs, the component (or its wrapper) notifies to the framework the component that triggered the event and the event name (and event "parameters", if

any). The *Execution Handler* looks up in the composition data structure, looking for the component that is "interested" in the pair <*publisher, event*> that identifies the event, and what is the associated operation to execute. The operation is then invoked, using the event parameters (optionally applying the data transformations needed).

Based on this paradigm, the DashMash engine and, in particular, its Execution Handler, plays the role of a controller in an MVC pattern where the model corresponds to the application logic of mashup components and the view relates to the presentation layer of each component and of the overall mashup.

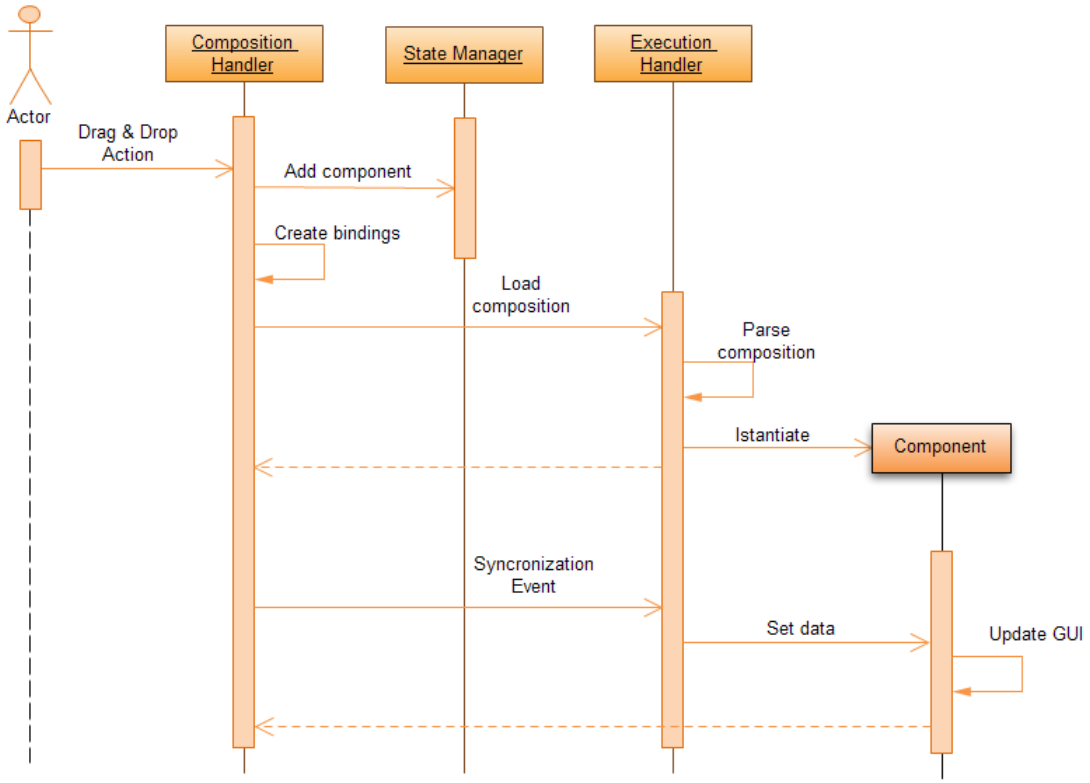


Figure 6.4: Interactions among the modules

6.4 Generation of the composition models: the *Composition Handler*

In our tool, due to the intermixing between mashup composition and execution, events can be related not only to users and system actions occurring during the mashup execution (those ones managed by the *Execution Handler*, which causes a change to some other component's state), but also to the dynamic definition of the composition (e.g., the drag&drop of a component icon into the composition area). The *Event Broker* intercepts events and dispatches them to the modules in charge of their handling. The *Composition Handler* manages composition events, interacting with the other architectural components as described in Figure 6.4:

- It automatically translates the addition of a component into listeners and, based on them, it immediately creates or updates (if already existing) the current composition model. As represented in Figure 6.5, the addition of a component is performed by moving the representative icons into a composition panel, the so-called *workspace*, a container implemented as an HTML `<div>` helping contextualize the effect of the composition actions on the components it clusters. Since the composition is immediately executed, the component UI rendering is also immediate in the workspace.

- It dispatches the composition events to the *status manager*, to maintain the description of the properties, in the format defined by the SMDL model, that can be useful to recover the status of a previously defined mashup for a later execution, but especially to let users modify their composition "on the fly". Modifications relate to default or specific parameter values (e.g., the value of a parameter for querying a data source), to layout properties (e.g., the colors used to show values on a chart) or to any other property that the user can set to control the component status, contents and appearance.

- It dispatches the `saveState` event to a *Save Manager* that provides the support for saving an instance of the composition. This process involves the two descriptors of the composition: the XPIL file, to know components involved and bindings between them, and the SMDL file, to know the last parameters settings of the components. Regarding data, there are two possible policies: *to save the data structures* (see Section 6.6.2) to create an instance of the mashup that does not need to be linked to a database/Web service for component visualization, or *to not save any data structure*. With the "not to save" policy, it is necessary to request data from the database/Web service for component visualization.

- It dispatches any composition events to an *History Manager*, to add the last user action into the history line. The History Manager provides the support for undo/redo the last user actions. As the matter of fact, one of the key features of direct manipulation systems is to make exploration safe for users, so that they can interact without fear of permanently loosing information.

- As soon as the composition and the status update is complete, it notifies the Execution Handler that in turn reloads the mashup composition and executes it according to its event-driven, publish-subscribe logics.

It is worth noting that the Composition Handler itself is a mashup component: any user composition action generates a Composition Handler's event, which is notified to and managed by the Execution Handler. An interesting side effect of this architectural choice is that event handling, whatever is the nature of the events, is encapsulated within one module, i.e., the Execution Handler. Even more important, the logic behind the automatic composition is "programmable" and, therefore, flexible: it depends on a set of pre-defined listeners, whose knowledge is inside the Composition Handler which, being a mashup component, can in turn be easily unplugged and replaced.

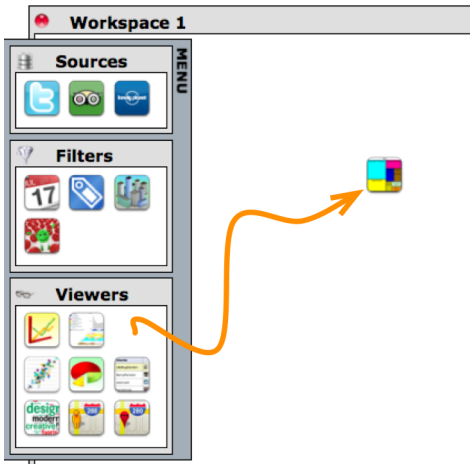


Figure 6.5: The drag & drop mechanism

6.5 Bindings definition

DashMash supports the definition of *default* and *custom* bindings among mashup components.

The former are automatically defined by the Composition Handler when a composition action is intercepted. This ensures a minimum level of inter-component synchronization that does not require users to define service coupling. When a component is added, the Composition Handler includes related default bindings in the XPIL, according to an embedded (but still configurable) logics.

Custom bindings are instead user-defined. Nevertheless, the Composition Handler offers also support by generating compatibility– and quality– based recommendations. To this aim, it dispatches the composition events to the *Recommendation Manager* that is in charge to evaluate the quality of the current composition and provides suggestions about the selection of possible components to add or substitute to the existing ones in order to achieve or improve the mashup quality.

6.5.1 Custom Bindings

Custom bindings are user-defined: first, the user selects which event exposed by the added component (that is the publisher) s/he wants to map. Second, s/he chooses which components (the subscribers) have to be related to this event. Third, s/he selects the operations of the subscribers that have to be called-back by the event rising. Figure 6.6 shows the dialog box supporting definition of custom bindings.

When defining bindings, the selection of suitable services must be primarily based on functional requirements. However, the quality of each involved services, as well as compatibility issues, can drive the production of recommendations that can help user building quality mashups. Many are the proposal for tools to develop mashups, but there is a lack of proposals for the quality of mashups we tried to fill this gap by adopting the approach

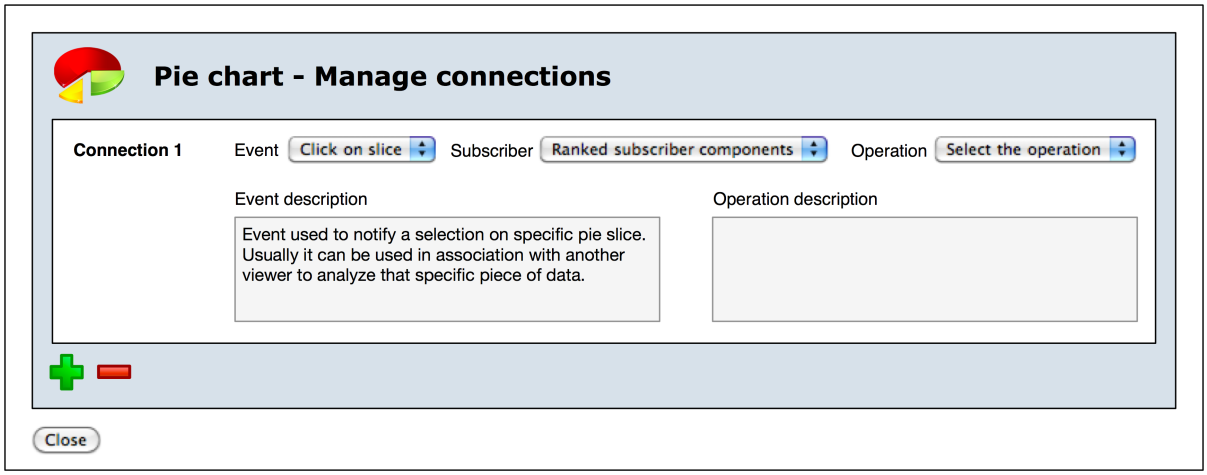


Figure 6.6: The dialog box for bindings definition

described in [48]. We indeed believe that, beyond quality in use, several issues must be considered, which are strictly related to the activities that characterize the mashup development process.
The typical scenario for mashup development spans from the production of single *mashup components* to the integration of selected components into a final *mashup composition*. Given the nature of mashups as applications integrating other resources, throughout the whole process the quality of the component resources and the adopted composition patterns play a fundamental role.
The component developer creates component services for mashups. We assume that developers correctly implement the service functionality, taking into account well-known principles, best practices and methodologies for guaranteeing the internal quality of the code. However, when used in a mashup composition, component services can be selected by considering especially some external properties. Since our aim is to investigate the quality of the final mashups, this is the perspective we are interested more, which is related to aspects such as the architectural style (e.g., SOAP services vs. RESTful services vs. widget APIs), the adopted programming language (e.g., client side such as JavaScript vs. server side such as Ruby), the data representation (e.g., XML vs. JSON), the component operability and interoperability (e.g., the multiplicity of APIs targeting different technologies). Such external aspects indeed affect the "appeal" of the component from the mashup composer perspective.
The component developer should try to maximize them, and should also make these quality properties visible for the mashup composers, who can therefore base on them the choice of components. The component developer therefore builds the component having in mind the quality properties to be maximized. S/He can also document such quality properties by means of suitable component descriptors. Listing 12 shows an example of a UISDL descriptor extended with quality annotations. Such annotations addressing the previous quality concerns requires the definition of sound models (such as the one defined in [20]), first of all focusing on the quality of components as stand-alone ingredients, but also on their attitude to generate quality mashups when combined with other components.

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

Listing 12: UISDL quality annotation example

6.5.2 Default Bindings

The basis for the automatic definition of default bindings is the classification of components into *data services*, *viewers*, *filters* and *generic components*, already presented in Chapter 5 for our domain-specific tool DashMash. Classifying components is in general useful to identify the minimum set of bindings that components in the defined classes requires in order to get synchronized. Such bindings are valid for any mashup instance. Therefore they can be automatically defined, without requiring the users to program explicitly the service couplings.

Although the classification we have adopted satisfies the specific requirements of our case study, it is quite general and reusable in several mashup contexts. For example JackBe Presto also exploits a similar classification of *mashables* [28]. What is new in our approach is that this classification of services allows us to codify default data flows that can be automatically extended by the platform through the addition of ad-hoc listeners. For example, filters are *producers* of parameters to be used to select data through the data service. Viewers, instead are *consumers* of data, as they elaborate contents extracted through the data service and produce aggregate data. Some components, e.g., data services, then can be both consumers and producers. The synchronization between the Composition Handler and the other components is then necessary to manage composition events. The composition visual environment is itself a mashup, where the Composition Handler plays the role of publisher for all possible composition events.

Figure 6.7 highlights the synchronization managed through the main default bindings. For example, any time a component is added into the mashup, the Composition Handler publishes a corresponding event that triggers a Data Service Client operation through which a pertinent portion of the updated state is sent to the Data Service Server (see

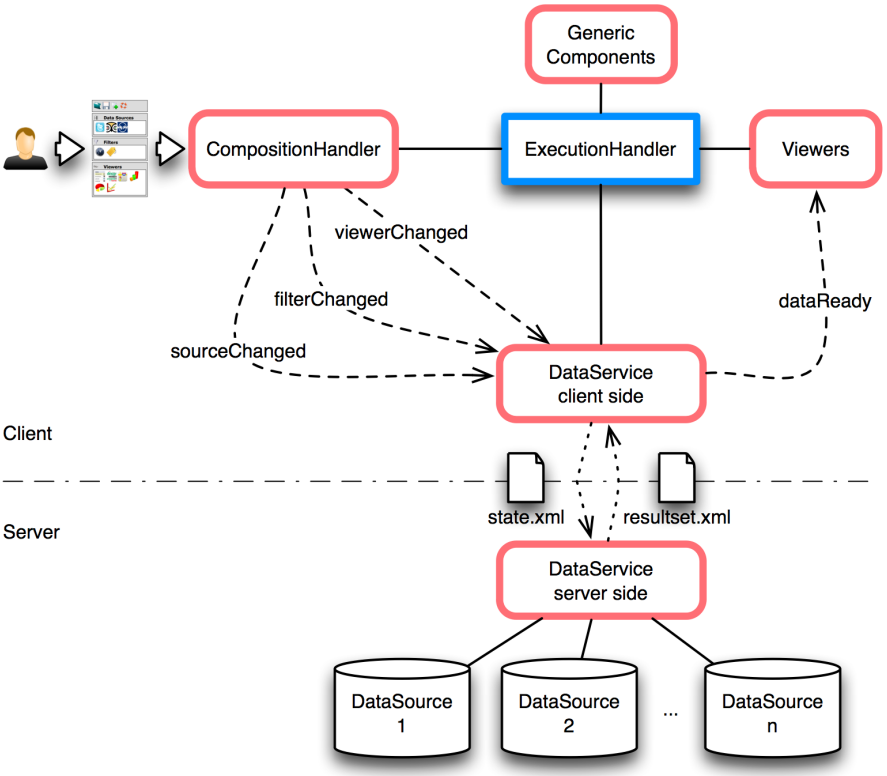


Figure 6.7: An example of bindings executions between CompositionHandler, DataService and viewers

Section 6.6.3). Based on the state information, the Data Service Server queries pertinent data sources and sends the result set back to the Data Service Client. Based on another default listener, the Data Service Client raises an event so that subscribed viewers know about the new result set and, thus, refresh their status. As better illustrated in Section 6.6.1, data communication between the Data Service and the viewers is managed by means of a local buffer in charge of storing aggregate data.

The "knowledge" about possible default mappings is coded inside the Composition Handler and can be easily configured, with the advantage that DashMash can be easily adapted to domains with possibly different services and classifications. In the end, DashMash can also work without classification, allowing users to associate components in any possible way, through the definition of custom bindings.

6.5.3 Quality-based Recommendations

While lightweight development processes are needed to alleviate the effort of end-users, the development of services is a demanding activity, to be performed according to traditional development processes by professional programmers. If on the one hand the success of a mashup is influenced by the added value that the final combination of services is able to provide, on the other hand it is self-evident that the quality of the final combination is strongly influenced by the quality of individual services.

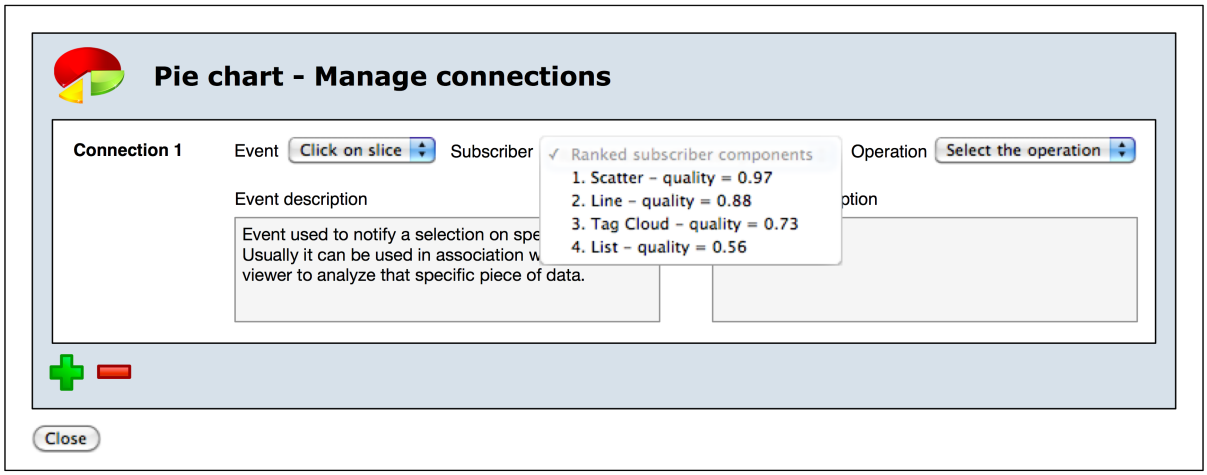


Figure 6.8: Recommendations about all the possible rankings

Defining models and techniques for developing high-quality services and for assessing their quality is a promising research direction to aid user innovation [20]. Anyway, it is necessary to consider that the quality of a mashup is not a simple aggregation of the quality of individual components, but it depends on the particular combination of components into a composite logic, layout and, hence, user experience.

During the design of DashMash we have considered all these aspects to propose an overall approach that can support users in the construction of high quality mashups. In particular, we have identified three opportunities to provide users with quality assessments: (i) the registration of new services, (ii) component ranking, and (iii) the composition of selected components.

Registration of new services

As described in Section 6.1, the set of components C available in the DashMash platform is composed of services that can be created ad-hoc or can be public. In both cases, the quality of the component for a given mashup depends on its functional requirements and non-functional characteristics. Since the Web provides a large amount of functionally equivalent data sources and component services, most of the times quality can be the driver to select the most dependable ones. In our approach, quality evaluation focuses on the assessment of both the intrinsic quality properties of the component and the reliability (i.e., reputation) of the Web information sources accessed to retrieve data. The intrinsic component quality considers all the dimensions defined in the quality model proposed in [20]. According to this model, the quality properties that should be maximized to obtain a high quality mashup represent traditional software quality aspects such as functionality, reliability, and usability of the APIs, data and presentation quality criteria.

As regards the reputation of Web sources, the data quality literature provides a set of quality dimensions that can be adopted for the evaluation of the reputation of structured and unstructured data. The variables that affect the overall reputation of an information source are, in general, related to the institutional clout of the source, to the relevance of

the source in a given context, and to the general quality of the source's information content [18]. In case of sentiment analysis, Web 2.0 sources (i.e., blogs and forums) represent a primary source of information. As defined in [18], the reputation of this type of sources can be evaluated by considering the traditional quality dimensions together with other specific dimensions such as the volume of information produced and exchanged, the overall range of issues on which the source can provide information, the degree of specialization of the source in a given domain, and the responsiveness to new issues or events.

All the data useful for the assessment of quality criteria are documented during the registration phase in the component descriptors. Formally, each component $c_i \in C$ is associated with a *component descriptor* and a *quality vector*. As illustrated in Listing 12 the component descriptor lists all the operations and related parameters and all the technical details needed to evaluate quality. The quality vector $QD_i = [qd_{i1}, qd_{i2}, \cdots, qd_{in}]$ contains the list of metrics through which it is possible to define the values of the component quality CQ_i and source quality SQ_i as aggregate quality indexes for the i-th component and data source, respectively.

Component ranking

When composing a mashup, users select components from the DashMash repository. In the selection phase, the quality of individual services represents the main decision driver. As described in the previous section, each component $c_i \in C$ is associated with the component quality index CQ_i and source quality index SQ_i. These two indexes can be linearly combined in an aggregate quality index $Q_i = w_1 * CQ_i + w_2 * SQ_i$ where w_1 and w_2 are user-defined weights. The values Q_i are used to define a ranking among similar components and support the user during the component selection phase.

Composition of components

When a component is selected, recommendations about how to further select components can be provided. The Composition Handler sends all composition events to the Recommendation Manager, which suggests actions to achieve or improve the quality of the mashup. To do so, we have adopted the approach described in [48], for which both the quality of component services and the overall mashup quality are the key drivers of recommendations. Components are ranked on the basis of their *mashability* [48] defined as the capability to be combined with previously selected components and to maximize the quality of the overall mashup. *Mashability* can be seen as a combination of compatibility and aggregate quality. Compatibility is an estimate of whether it is possible to combine a component with those already included in a composition. It is based on (i) Syntactic compatibility and (ii) Semantic compatibility. *Syntactic compatibility* checks the compatibility between the input/output parameters exposed by the components. *Semantic compatibility* checks whether input/output parameters and operations belong to the same or similar semantic categories, assuming that syntactic compatibility is satisfied.

The compatibility index $comp_{ij}$ provides a preliminary measure of the compatibility of a service to be added to a given mashup. A value equal to zero indicates the incompatibility from a syntactic point of view while a positive value provides a measure of semantic compatibility. The Recommendation Manager stores this index in form of a matrix where events and operations of the available components are related to each other and their compatibility is scored. The matrix is constructed every time a new component is added to the repository C.

The notion of aggregated quality is an estimate of the final mashup quality achieved by aggregating the quality of individual component services. Aggregate quality is needed to build a ranking and suggest an ordered list of possible bindings. Note that the quality of mashup QM_k cannot be quantified as a simple sum of the quality of individual components, but it is necessary to weigh quality by taking into account the role and the importance of each component [21]. As shown in Figure 6.8, DashMash provides the user with recommendations about all possible bindings ranked on the basis the aggregate quality value of the final mashup.

Once a composition is in place (including at least a couple of components), quality can be assessed by considering the composition status and the result set in order to suggest alternative mashups that may provide the same functionality of the current composition with greater aggregate quality. In DashMash, each result set is indeed structured so as to highlight how the dimensions of the retrieved data relate to the visualization dimensions of the selected viewers. For example, in the scenario depicted in Section 7.6, a user might filter data based on a specific brand category. The amount of resulting data series to display might drive the selection of the graph to visualize them. Components that build line charts and 3D graphs perform the same functionality but with a different expressiveness. The former are more clear in case of a limited number of categories (e.g., lower than five) while the latter are better in case of a higher number of categories. At the end of a mashup composition, DashMash analyses the choices of the user and, on the basis of the information about data and presentation extracted from the composition status, can make recommendations for a new composition.

6.6 Integration with proprietary data sources

In the "consumer" mashups each component in the composition takes its own data from remote data source. For example *Google Maps*, the most famous and used API in the Web, has its own data source storing all the information required in order to draw maps. The component which wraps the *Google Maps* API has to send a request to the service provider communicating the new parameters needed to redraw the map.

When end users need to use their proprietary data sources, components need to access this data source in order to retrieve data, with the advantage that the organization of data is known because the service is locally managed. This is not only the case of Enterprise Mashups, in which the mashup is a decisional or analytic dashboard. For example, as shown in [41], a personal trainer could be interested in providing a "diet service" that helps other people to monitor their diet, which extracts data from a "personal" database

with food and calories. All these very heterogeneous cases could be resolved using a domain-specific component, called Data Service, that knows the schema of the datasource and therefore is able to extract and represent data, so that they can be best exploited by other components within the mashup.

6.6.1 Data Service

Data Service (DS) is a double-sided component: it has a client side and a server side. Figure 6.7 shows this architecture and the communication paradigm between DS client and DS server. The DS server should be considered *de-facto* as a Web service [32], which in our case also needs a UISDL description to interact with our mashup engine.

DS server is strongly dependent on the data source technology, model, schema and instance, but DS client is a generic mashup component that accesses the server side service as a common API wrapper and is independent from the data source.

In the following we will describe in details both the DS client and the DS server.

6.6.2 DS client

DS client is a common mashup component that exposes events and operations. Its role is to catch the composition events that implies data updating operations, and sends the corresponding requests to the DS server, in charge of retrieving data and returning the XML description of the result sets.

In order to retrieve data from the data source, the DS should know the context which determines the query to be executed. Our solution is to make the DS client pass to the DS server a part of the status document. The DS server makes a query for each viewer that is passed to the service so the client send only the necessary requests.

There are three the main events that trigger the construction of a new query:

1. `filterChanged.` Since filters are associated to viewer boxes, when a filter changes all the viewer boxes need to be redrawn. In order to achieve this effect the Composition Handler raises the event `filterChanged` and the corresponding operation is invoked in the DS client, which retrieves the portion of the status needed. This portion relates to the viewer boxes in which the filter changed.

2. `viewerChanged.` If a viewer is added or some of its parameters are modified, the event that is raised is `viewerChanged`. This event is bound with the operation in DS client that sends to the service the correct portion of the status model. This portion corresponds to the viewer box filters and the interested viewer refer to.

3. `compositionReload.` If this event is raised, the status model is passed to the server side.

In all the three cases, after handling the events as describe above, the DS server executes the needed queries and returns the XML description of the result set. This XML is parsed

and sliced into viewers subset of data. Such data are locally stored in a *data buffer*. Each viewer can read from this buffer in order to retrieve data and draw itself. Figure 6.9 shows the matrix representing the data buffer. The rows represent the viewer boxes and the columns represent the viewers.

When are uploaded from the data buffer, the DS client raises an event of `dataReady` to the right viewer.

	$viewer_1$	$viewer_2$	...	$viewer_m$
$viewer\ box_1$	$result\ set_{1,1}$	$result\ set_{1,2}$	...	$result\ set_{1,m}$
$viewer\ box_2$	$result\ set_{2,1}$	$result\ set_{2,2}$	...	$result\ set_{2,m}$
...	...	...	...	...
$viewer\ box_n$	$result\ set_{n,1}$	$result\ set_{n,2}$	...	$result\ set_{n,m}$

Figure 6.9: The data buffer matrix structure

6.6.3 DS server

DS server is strongly dependent on the data sources on which it has to make queries. However, is possible to describe a common behaviour that can guide a domain developer to develop this module.

In the following we will describe the main steps that Data Service server side performs to provide the result set to the client side.

Parsing of the status As described in the previous Section 6.6.2, the Data Service client side sends the status descriptor to the server side and here the status is parsed in order to construct queries on the data sources.

Query generation and execution According to the status descriptor passed to the server side, all the needed queries are generated and executed. Different approaches could be adopted in order to improve the performance of this step, e.g., assigning the execution of multiple queries deriving from different viewers to different threads.

Result set formatting and response composition The result sets returned by the query execution are not in the format that the composition viewers could interpret in order to draw charts. So in this phase the result set is formatted according to the DFM schema described in Section 5.4.1.

6.7 Implementation and deployment choices

The current implementation of the DashMash architecture consists in a client-side module with a modular composition engine core.
The natural environment for applications that lend themselves to presentation integration these days is the Web. Most of the existing mashups and mashup environments link together different services with different underlying technologies, often time using JavaScript as the "glue" that make the composition possible. Nearly every technology currently used for Web applications, such as Java, .NET and Flash, allows some kind of interaction with JavaScript scripts in the hosting Web page. Therefore HTML/JavaScript was the natural choice of platform for the implementation of the runtime framework.

The choice of a client-side solution is due to the need of keeping our solution simple, even from the point of view of the platform deployment. However, in order to simplify the client-side architecture and to enable new features, such as the multi-user and cooperative definition and the immediate execution of mashups, it is also possible to enrich the server-side logics. In order to meet this aim, it is necessary to implement a session paradigm and a multi-client orchestration by defining a server-side cooperative module. This cooperative module should be able to manage the SMDL and XPIL descriptors also assuring the mutual exclusion during multi-users interaction. In this scenario, synchronization becomes a critical factor, because the different instances of the same composition running on different clients should be aligned each other. This can be easy done using a two-step paradigm that implements a lock mechanism on composition modification by users, and a sync notification that broadcast updated composition and status descriptors to re-instantiate the composition client-side.

6.8 The DashMash evolution: Peudom

Although the models described in Chapter 5 about Dashmash are not linked to a specific domain, we have considered our case of study during the development of the architecture. This has been done in order to reach goals such as performance and effectiveness of data representation to meet the needs of the Comune di Milano.

Peudom is the result of a re-engineering process studied in depth, that starts from Dashmash Platform to eliminate any possible link to a specific domain; we have also introduced some improvements concerning the separation between the User Interface management and the core framework, the data flow management for generic components and the runtime engine organization.

At the end of the process we have obtained a more flexible architecture whose schema is shown in Figure 6.10.

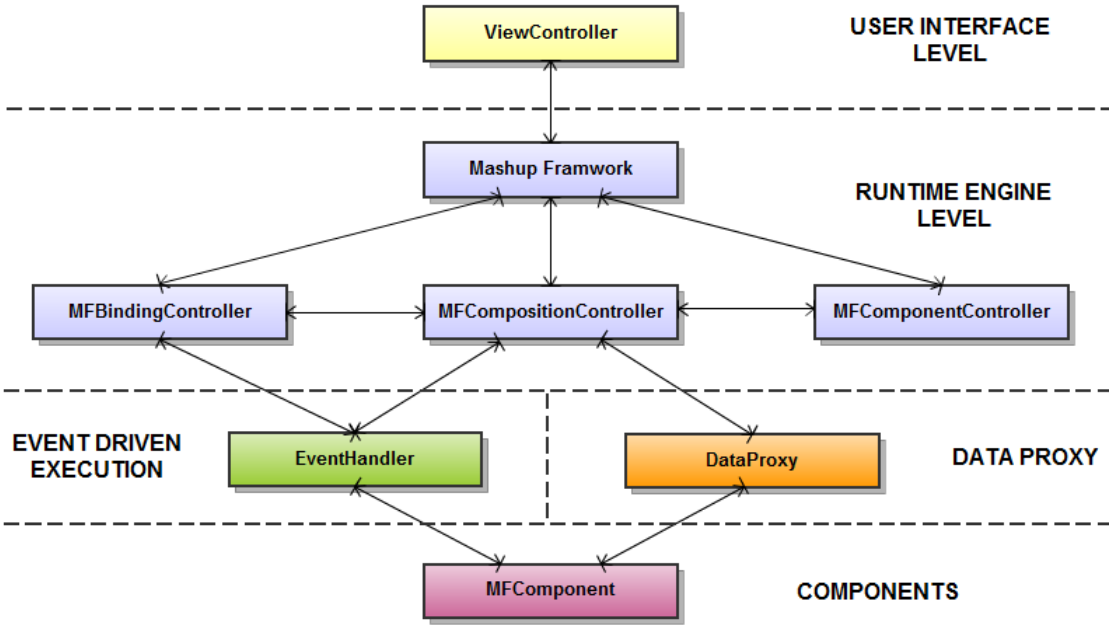


Figure 6.10: The Peudom architecture

In Peudom, the UI level in not integrated in the framework, but is a natural extension of it. Such level handles the initial rendering of the development environment, managing simple UI actions such as adding or removal of a component or bindings. It is made up of a single module, the WebViewController, that initializes the mashup composition environment starting from a template EJS (that could be optionally specified or defined by default) and produces a web page that will be the user operational environment.

Another aspect of the UI management re-engineering process consists in introducting the use of templates for component graphical rendering: we have defined a mapping paradigm between the properties exposed by the component and the graphical rendering of the elements specified by a template linked to component itself. So each component is shown through the interpretation of a template that, if it is not specified during the initialization, is a default template already provided for each component.

This paradigm will make easier the introduction of an automated creation meachanism of the graphical interface: an editor will enable the End Users to easy define how data coming from local or remote data sources have to be represented by graphical elements of a visualization template.

As concerns the Data Flow Model, in Peudom we have introduced a DataProxy module that replaces the DataService defined in DashMash. The DataProxy has to manage requests of information retrival for those components that aren't tailored to work with specific services (i.e. wikipedia, twitter, youtube, flickr, etc.), but needs to use queries.

This module can handle two different types of data sources: the first type needs to be locally recorded from the system administrator and allows to receive raw data from several formats (i.e. excel sheets, xml, etc.) providing that they are standardized. The second type involves non local sources such as remote services, propriertary data sources

protected by copyright, or heavy databases that couldn't be included locally.

The communication among components is based on an object called MetaQuery, in which components specify parameters such as domain, series, range, filter values (i.e. minimum or maximum) and, if in case, clusters; this object returns a data structure that is given to the component interface for data rendering.

This mechanism allows to exploit the use of template for components: in order to represent data using a template, it is necessary to define a markup description language that, in connection with the MetaQuery mechanism, matches a general typology of a component with the data that it can visualize. In this way, problems caused by data sources containing data format that are incompatible with a component are avoided (i.e a series of temporal data without geolocalization information can't be represented on a map).

As far as Runtime Engine Level is concerned core functionalities have been reorganized in four modules that manage the composition paradigm.

- `MashupFramework`: it initializes BindingController and EventHandler modules. It is an interface among the different modules that composes the framework and the graphical interface, providing a series of methods that allow an easier integration. When the user interacts with the graphical interface, a viewController initializes the basic elements and uses the functionalities exposed by the MashupFramework in order to manage the core functionalities. Moreover, it implements functionalities such as framework state save and load using a Json structure.
- `MFBindingController`: this module manages the bindings among components, interpreting a compatibility matrix defined in a configuration file that describes all allowed bindings (pairs of <event, operation> between a pair of components). The MFBindingController is able to understand which are the possibile bindings and to return a list of compatible events or operations, on the basis on the component typology.
- `MFCompositionController`: it manages the composition in terms of addiction or removal of components and bindings. As our composition paradigm allows the realtime execution and definition of the mashup, it is necessary to mantain the consistency of the system state. It modifies the structure of the mashup using appropriate primitive functions; it also interacts with MashupFramework and an EventHandler allowing callback in case of a component rises events that is involved in the binding). At last, it generates a unique identifier for every binding.
- `MFComponentController`: this module instantiates and removes an instance of a component, according to user actions. It also manages the retrival of all properties connected to the component istances (i.e. in order to generate a unique identifier for the component instance).

The Figure 6.11 shows an example of the system dynamic behaviour and the interaction among the different modules.

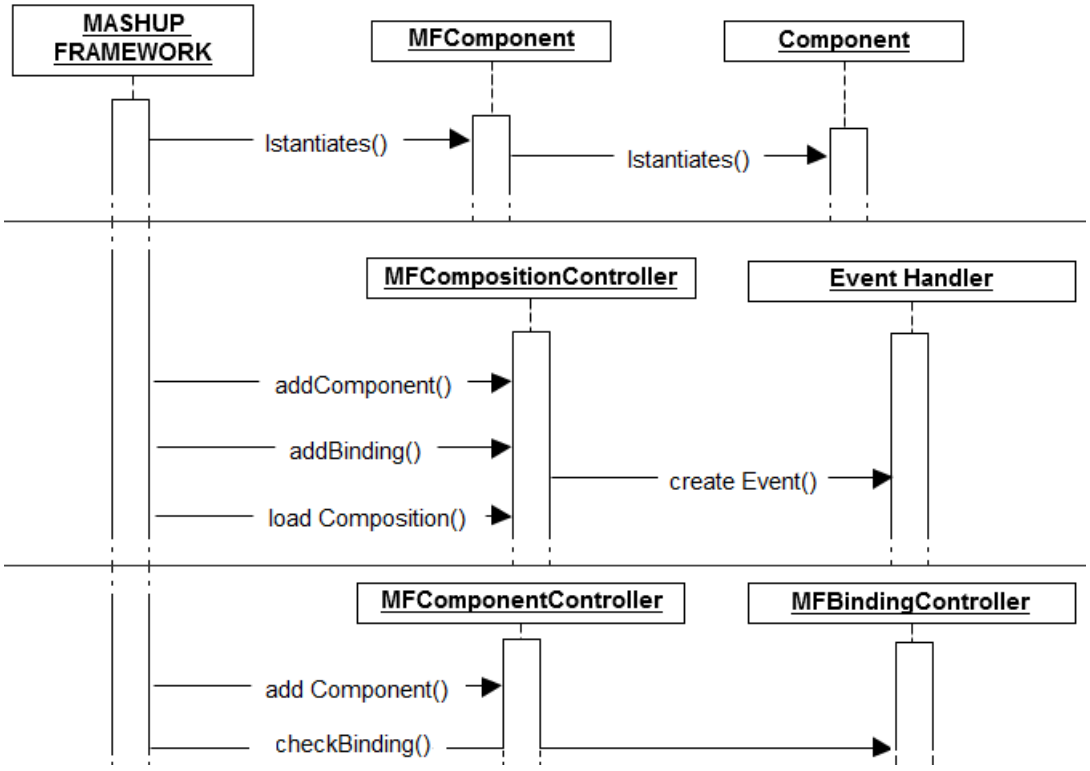


Figure 6.11: Peudom dinamic behavior

Part IV

Validation

Chapter 7

Case study

In the context of a project funded by the Comune di Milano (Milan Municipality) we have worked on the construction of a Web platform through which end users can construct their dashboards for *sentiment analysis*. Sentiment analysis focuses on understanding market trends starting from the unsolicited feedback provided by users comments published on the Web (this will be better explained in Section 7.1). The project focuses on the definition of a city brand marketing model and on the design of an engine in charge of the automatic extraction of sentiment indicators summarizing the opinions contained in user generated contents [17].

After preliminary attempts with a traditional static dashboard fo the presentation of analysis results, we have realized that end users could benefit from the ability to compose their analysis flexibly, playing in variable ways with sentiment indicators, and also complementing such indicators with "generic" external Web resources. This latter feature would indeed help them to improve their analyses, by interpreting sentiment indicators with a view on the events that cause trends and behaviours. This observation led us to introduce mashups to accommodate flexibility and openness towards external resources.

Within this thesis we therefore customized DashMash for this specific context. This implied the development of specific components supporting sentiment analysis tasks, namely *ad-hoc* implemented viewers and a Data Service in charge of assessing the local data source where sentiment analysis data are stored. Such customizations are described in Sections 7.5 and 7.4 . The first sections of this Chapter also provide an overview over sentiment analysis, to highlight the requirements that guided us in the DashMash customization activity.

7.1 The context: sentiment analysis

With the Web 2.0 revolution there are a lot of Web sites enabling people to publish their contents about specific areas of interest and people can easily publish and share their information and opinions further increasing the huge amount of contents available in the

Internet. In this way Web users play an active role on publishing information being authors themselves.

The most important Web sources where people publish their contents can be divided in two types: *social sources* and *content sources*. The *social sources* are social networks that are focused on users and the social relationships among them. In these Web sites users mainly focus their attention on WHO are the senders and WHO are the recipients of the contents. The *content sources* are Web sites in which people are virtually connected by having common interests rather then having real social relationships. In this kind of Web sites users are mainly interested on WHAT are contents and WHAT are the experiences that they share. This classification is not exclusive but can be applied to better understand the aims of Web sources. Some social sources can include content-based sections where users concentrate on developing threads on some specific topic while some content sources can be further extended with the possibility to associate users with relationships. People can also have different approaches to the Web sources. Especially in social networks, people tend mainly to insert their original personal information and social relationships (this is the case of *Facebook* and *AsmallWorld*). The greater part of the Web sites, instead, holds pseudo-accounts or fake accounts: in these cases people can hide their personal identities and feel free to express their personal opinions about what they are interested on (this is the case of the forums).

Thanks to these new technologies, institutions and enterprises have the opportunity to exploit the availability of this invaluable and huge amount of new marketing information to obtain direct feedbacks from people. This is very important because year by year people are always more influenced by Web sources. Besides traditional ways of communication, people often look for many useful advices in the Internet. This information supports users in decision-making processes that are strictly bound to the reputation of any company's products and services.

Sentiment analysis or *opinion mining* aims to determine the judgement or evaluation which a user gives on a particular topic.

The rise of social media such as blogs and social networks has improved interest in sentiment analysis. With the proliferation of reviews, ratings, recommendations and other forms of online expression, online opinion has turned into a kind of virtual currency for businesses looking to market their products, identify new opportunities and manage their reputations. As businesses look to automate the process of filtering out the noise, understanding the conversations, identifying the relevant content and actioning it appropriately, many are now looking to the field of sentiment analysis.

For companies it is impossible to manually analyse all the information from social and content services, automatic analysis tools are therefore needed platform for the Web reputation analysis.

Milan Municipality project focuses on several aspects of sentiment analysis: market analysis to understand the domain of analysis, the definition of main categories in which the information retrieved from the Web should be partitioned, the semantic analysis of the Web surfers' posts, the storage of these information in a unique data source and the development of an engine for the automatic extraction of sentiment indicators. DashMash

has than been used as a front-end environment for the lightweight mashup-based composition of sentiment services.

7.2 Sentiment analysis techniques

The basic task in sentiment analysis [27] is classifying the polarity of a given text at the document, sentence, or feature/aspect level — whether the expressed opinion in a document, a sentence or an entity feature/aspect is positive, negative or neutral.

Another interesting analysis is subjectivity/objectivity identification. This task is commonly [47] defined as classifying a given text (usually a sentence) into one of two classes: objective or subjective. This problem can sometimes be more difficult than polarity classification [42]: the subjectivity of words and phrases may depend on their context and an objective document may contain subjective sentences (e.g., a news article quoting people's opinions).

The more fine-grained analysis model is called the *feature/aspect-based sentiment analysis* [46]. It refers to the study of determining the opinions or sentiments expressed on different features or aspects of entities, e.g., a cell phone, a digital camera, an insurance, or a bank. A feature or aspect is an attribute or a component of an entity, e.g., the screen of a cell phone, or the picture quality of a camera. This problem involves several sub-problems, e.g., identifying relevant entities, extracting their features/aspects, and determining whether an opinion expressed on each feature/aspect is positive, negative or neutral.

The technique for sentiment analysis adopted for our case study [16] performs an evaluation of the average opinion on a given subject of interest measured on a qualitative 5-point scale, ranging from very negative to very positive. The input data to the sentiment analysis are a set of snippets, i.e., fragments of text providing a comment on the subject of interest. Sentiment is evaluated on individual snippets first and then aggregated over a set of snippets.

Most commercial tools providing sentiment analysis services aggregate the evaluation of sentiment on individual snippets by calculating the mean value of sentiment over the entire set of snippets or over a subset of snippets that have been published on the Web within a given time frame. Snippets that comment on the same subject of interest are usually stored in the same set even if they are gathered from multiple heterogeneous Web sources. The mean value of sentiment across a set of snippets is usually complemented by the total count of snippets that have been retrieved, in order to provide users with an indication of the level of interest on the subject. [50] provides a survey of existing tools that clearly indicates that the volume of talk on a subject and the average evaluation of sentiment on large data sets retrieved from a broad range of sources represents a common approach. This approach, however, raises a number of information quality issues which could lead decision makers to wrong interpretations and decisions. These issues are related to: the selection of Web information sources and the characterization, interpretation and

evaluation of the content of Web information sources.

The classification of information sources and the assessment of the quality of their information improves the reliability of reputation assessments and provides users with a tool to select dependable sources in relation to the analysis that they need to perform. Users are thus allowed a more focused search and more reliable interpretations.

7.3 Sentiment analysis tasks

During the initial requirement analysis, some specific analysis tasks have been identified. In the following we will describe these tasks as they are supported within the DashMash customization for sentiment analysis. These analysis tasks are supported by means of *ad-hoc* developed viewers, providing advanced visualizations for sentiment indicators. The remainder of this Section illustrates the identified analysis tasks, by clarifying the data dimensions and measure involved in each analysis task and the corresponding visualization components.

The design of the underlying data-warehouse will be illustrated in Section 7.4.2.

7.3.1 City brand comparison

This analysis allows to *compare the city brand volumes* in a time interval. The dimensions selected are *Brand* (Milan, London and Madrid) and *Date* and the measure is the *Volume* of snippet. Figure 7.1 reports the charts as it appears in the implementation for the *Milan Municipality*. The brands are represented as series, the time is the domain of the charts and the volume is the range of the chart.

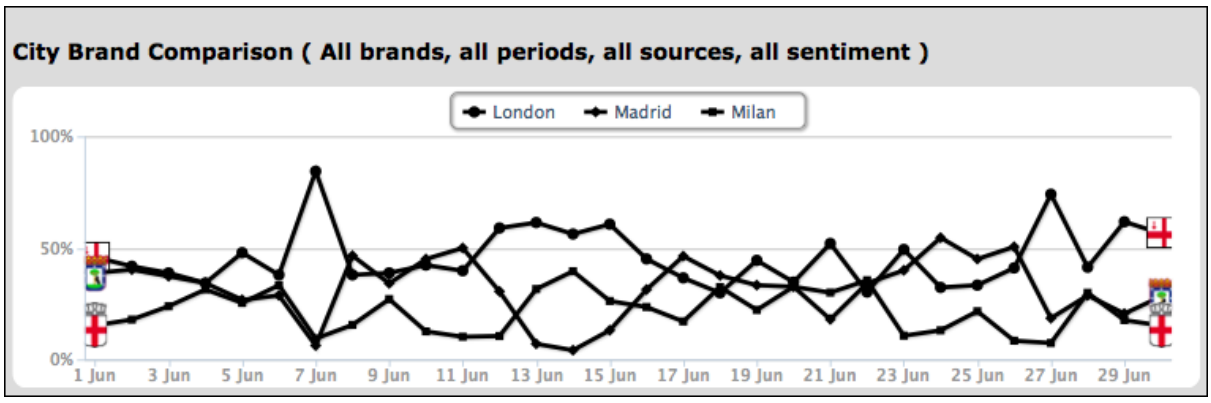


Figure 7.1: The city brand comparison analysis

7.3.2 Comparison by category

The *comparison by category* analysis shows the evolution in the time of the volumes of user snippets of the different categories. The involved dimensions are *Tag* (category) and

Date and the measure is *Volume*. In Figure 7.2 we can notice that the categories are represented as series and the time and the volume are represented respectively as domain and range of the chart.

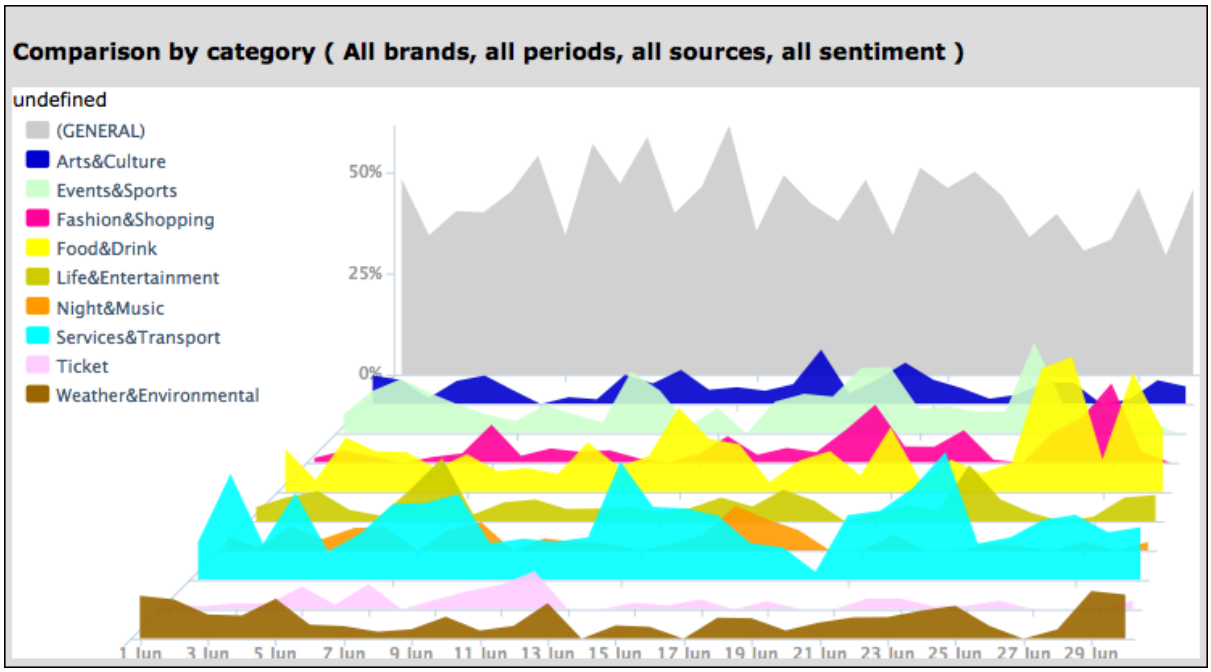


Figure 7.2: The comparison by category analysis

7.3.3 Volume distribution

This analysis provide the volume of tags (categories) with respect to the *Brand* (Milan, London and Madrid) and the *Tag* dimensions – obviously the measure is the *Volume*. Figure 7.3 shows the chart that we adopted to represent this analysis. This kind of visualization, as described in Section 3.2, is a *tree map*. The three series in the Figure are the three brands that are divided into the categories rectangle. Each rectangle has a dimension that represents the relative volume of snippets that talk about each tag.

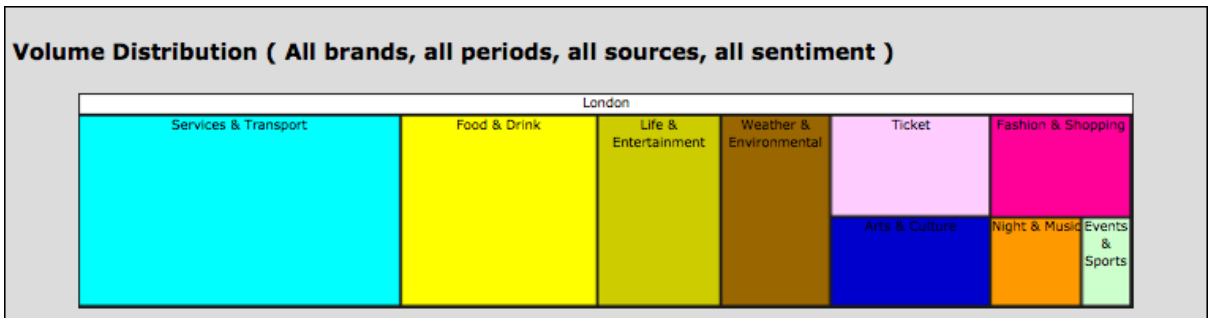


Figure 7.3: The volume distribution analysis

7.3.4 Sentiment distribution

This analysis aims to show the *sentiment* and the volume *distribution* among categories and subcategories. From a data warehouse point of view, the dimensions are *Tag* and *Product/Service* and the measures are *Sentiment* and *Volume*. As represented in Figure 7.4, the scatter chart supporting this analysis has as domain and range the two measures, respectively *Volume* and *Sentiment*. Instead the two dimensions are used to represent the point identified by the function that has *Volume* and *Sentiment* as range and domain. The color is associated with a colour code to the category and each point represent a subcategory (*Product/Service* dimension).

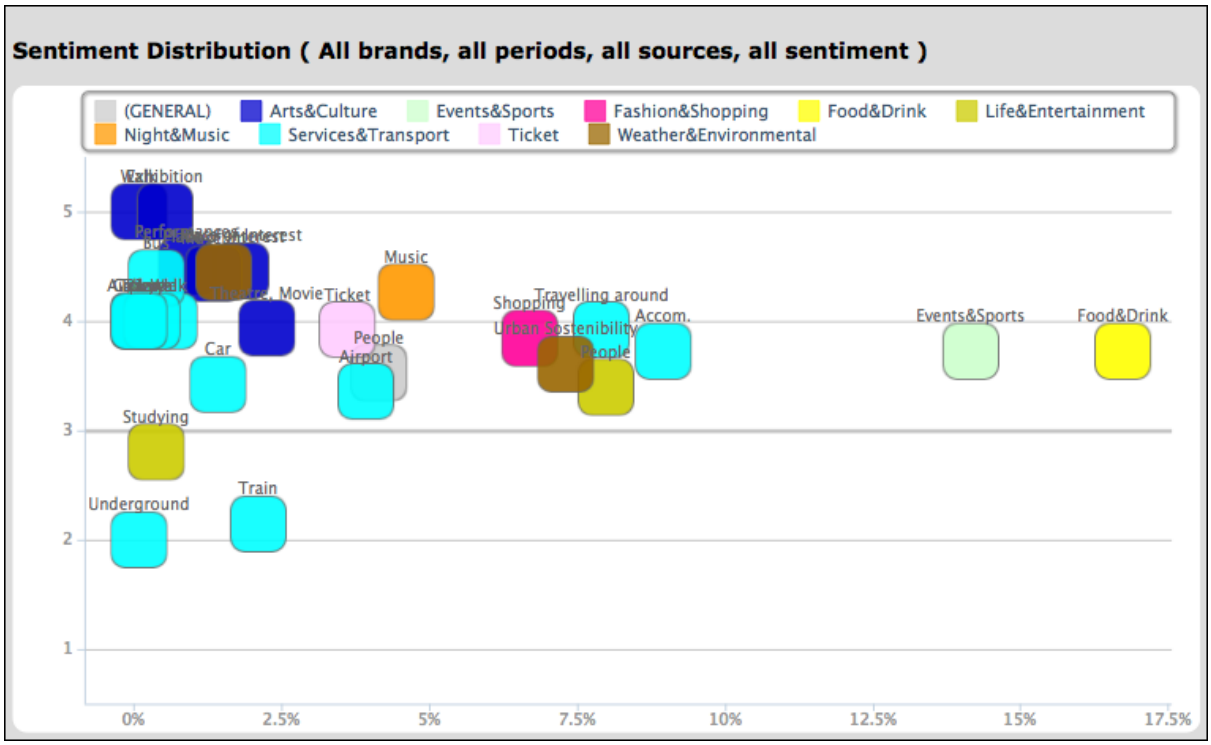


Figure 7.4: The sentiment distribution analysis

7.3.5 Polarity pies

This analysis is used to understand the polarity distribution of the judgements expressed by users with respect to polarity and tag. In Figure 7.5 is shown the chart used to visualize the results of this analysis. We can notice that in this case the dimensions are *Polarity* and *Tag* and the measure is the *Volume* of snippet. Polarity is represented as series and for each tag there is a slice of the pie.

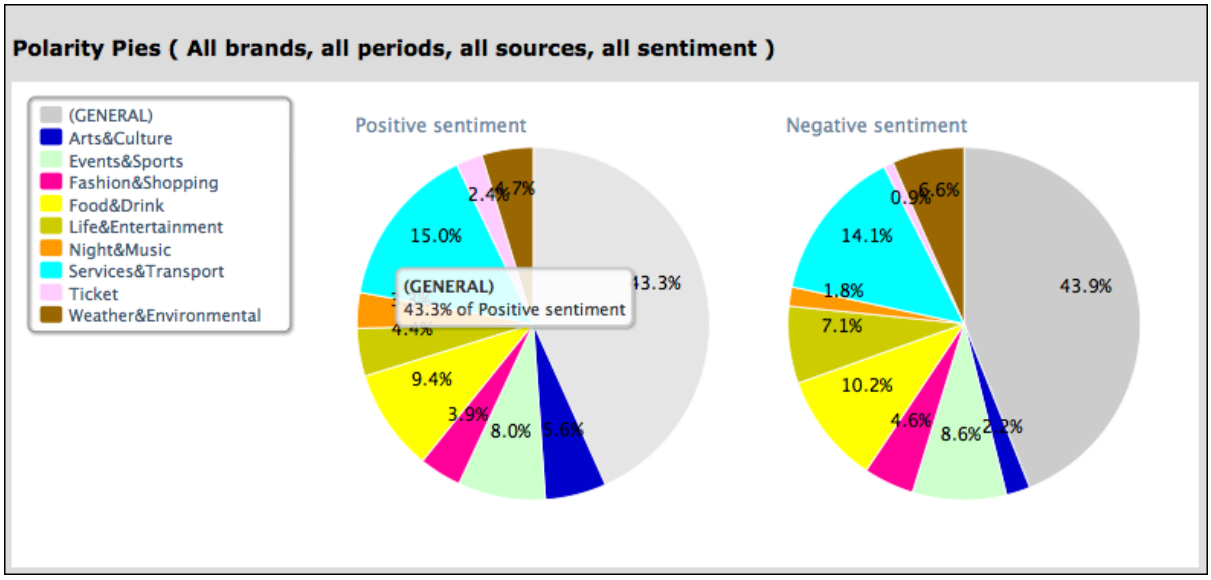


Figure 7.5: The polarity pies analysis

7.3.6 Opinion flow

The *opinion flow* is an analysis that shows a detailed view on the post and the users. The representation of this analysis is, as we can see in Figure 7.6, a table. The table represents the user with his profile image, the source where the post is published, the sentiment, of a snippet of the post, the corresponding brand (Milan, London and Madrid), and the tag and the post text. On click on a row of this table, a detail of the post is shown.

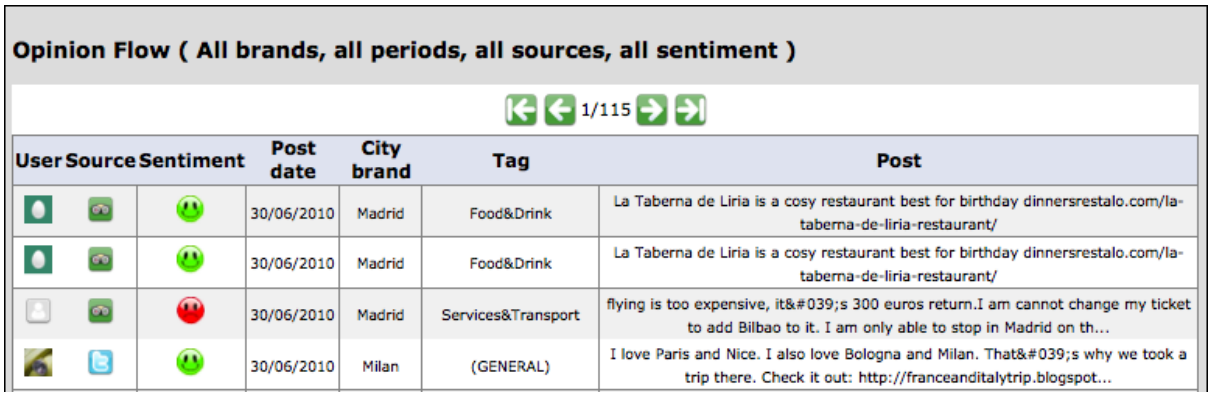


Figure 7.6: The opinion flow analysis

7.3.7 Sentiment cloud

The *sentiment cloud* aims to show the top 100 words used in the posts related with a brand. Each word in the tag cloud is associated with the sentiment value and the word volume. This component was not reliable on the Web, so we implemented it using an *ad-hoc* JavaScript business logic. The available public APIs are indeed difficult to customize for the display of data extracted form data base (not just text).

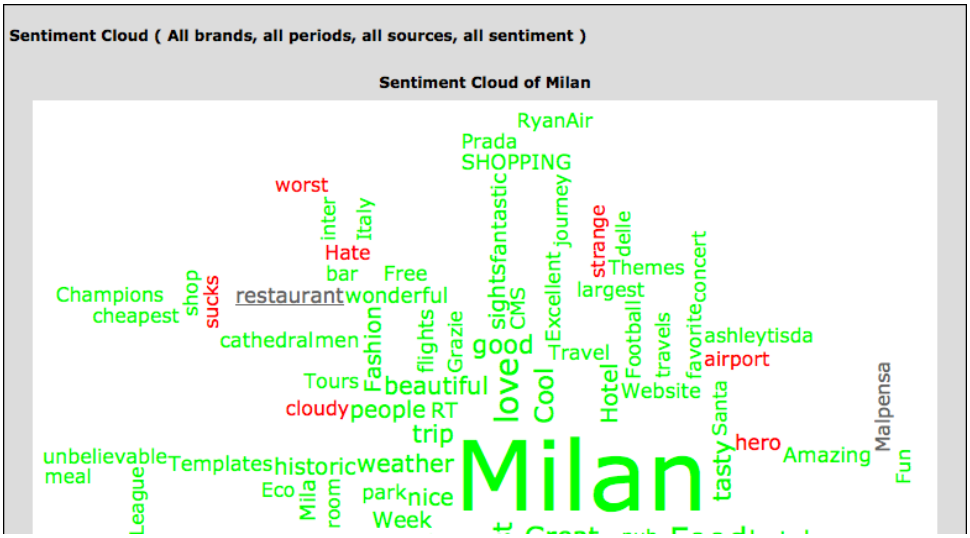


Figure 7.7: The sentiment cloud analysis

7.3.8 Top influencers map

This analysis aimed to show the top influencers. With different techniques the top influencer are individuated and are monitored. In the Figure 7.8 are shown the top influencer listed in the table and located in the map. The map component is derived from *Google Maps* API. This analysis is very important because the top influencers are to be monitored because their judgement could influence a large amount of people.

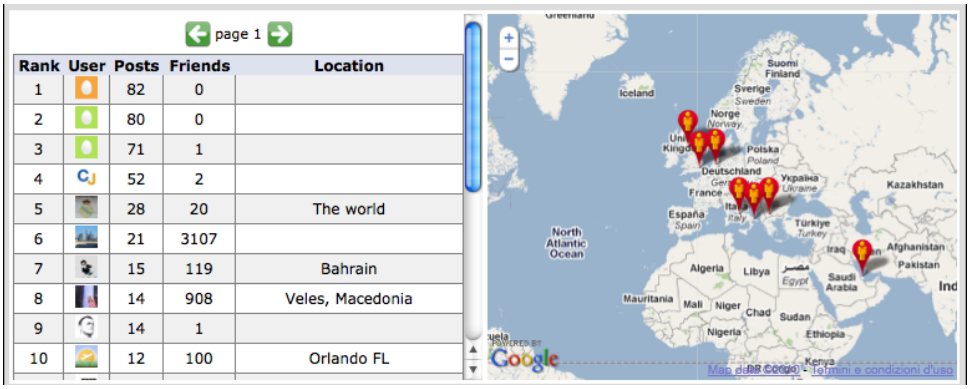


Figure 7.8: The top influencers map analysis

7.3.9 Posts map

This analysis shows the location on a map of the posts of a top influencer.

Figure 7.9 shows the post location in a map and the list on the left shows the posts. In the example represented in this figure the influencer is a travel agency and the places located on the maps represent the different sites in which the travel agency is placed.

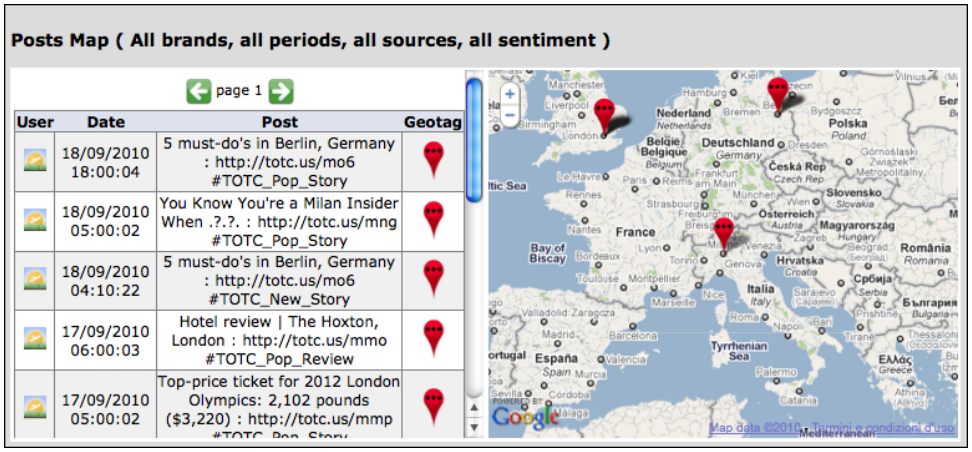


Figure 7.9: The posts map analysis

7.4 Data Service

7.4.1 Data integration

The analysis tasks described in the previous section require the integration of data coming from the different monitored sources. The information available in the Web is generally non-structured because it is presented in HTML format. It is also different to derive its semantics. However the schema of such information can be inferred analysing each particular Web site structure. Crawler programs can extract information from such site and give them a structure (and also a semantic). Each crawler is in charge of reading the raw contents of a Web site and converting them in an global schema that is the database of the platform. This operations can be done through two different approaches: HTML parsing or API call.

HTML Parsing The HTML parsing has the disadvantage that it must be designed ad hoc because it mainly depends on the presentation layer of a Web information system. Then this approach is enough volatile but has the advantage that has not particular limitations on HTML requests. This approach can be more complex when dealing with high dynamic Web sites based on *JavaScript* and *Ajax*.

API call The API call is the best solution because it can be integrated in the system and it can be considered as a *black-box* function that returns the data we need, hiding the specific architecture of the Web site. This approach is more durable because the structure of the database of Web applications is more stable then their presentation. However API calls have the disadvantage that they can be limited by the Web source system to prevent DOS attacks or brute forcing. Unluckily APIs are not available for all the Web sources, thus HTML parsing is often a forced choice.

The sentiment analysis the *Milan Municipality* is interested in needs to compare multiple Web sites. Each Web site has its own pages structure or provide its own API. In the project each crawler writes on its database according to its schema. In order to visualize and compare data from different data sources there was the need of an integrated schema.

Moreover the sentiment analysis process is very heavy and it could not done "on the fly", therefore also the results of sentiment analysis has to be stored with the crawler data.

The crawler data were not the only information stored. Also third party data were stored and has been integrated. These data are the result of the sentiment analyses of a consulting society that have its own schema to represent crawled data and analysed data.

To meet this needs an integrated data source was designed and all the crawling data and all the analysed data has been stored in the integrated database.

Layers

As shown in Figure 7.10 different data layers are identified in the integration process.

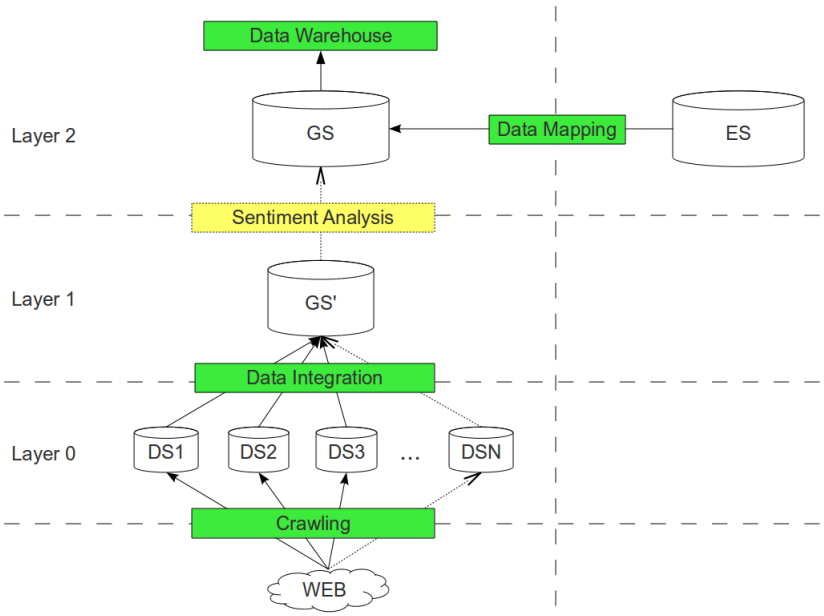


Figure 7.10: The data integration layers

Layer 0 Crawling processes extract data from Web sources and store them in local data sources.

Layer 1 The Data Integration system merges all the local data sources into a general global source that will be the input layer for the Sentiment Analysis module.

Layer 2 Sentiment Analysis processes elaborate data that can be analysed by the Data Warehouse system.

Integration approach

The schemas of the different data sources are heterogeneous and this introduces complexity in the mappings between local and global schema for entities or relationships that are implemented in different approaches.
Since the crawlers are designed *ad-hoc* for each Web site and the Web site databases schema are stable, the data sources are enough stable; the main problems can be due to changes to the Web page visualizations of the Web sites analysed by page-parse crawlers.
Even though we have chosen a materialized database, the mapping between data sources and global schema follows the *Global As View* approach [[40]]: the global schema is expressed in terms of the data sources schemata. There are not global view because they are represented by materialized tables in the global schema. Under this approach, when adding a new data source, the global schema does not need structural changes. To add a new data source, it is only required to create a stored procedure for the asynchronous conversion of the local source following the mapping rules. The schema mappings drive the implementation of the global database.

Figure 7.11 shows the global schema. In the following we are going to describe the different entities in the global schema:

`SOURCETYPE` contains the types of the sources (blog, forum, etc...)

`SOURCE` contains the sources (Twitter, TripAdvisor, LonelyPlanet, etc...)

`USER` specifies the authors of the posts

`FOLLOWING` specifies the relationships among users of the sources

`THREAD` contains the threads found in the sources

`POST` contains the posts found in the sources

`BRAND` contains the keywords searched (in our case brands are Milan, London and Madrid)

`SEARCH` specifies the keywords searched in the sources

`TAG` contains the tag model that is the categorization of the topics that could be identified by the semantic analyser of the texts

`QUALITY` contains the quality model which is a set of quality that can be associated to a tag to better express the user thought

`PRODUCTSERVICE` contains the product-service model (another information that could improve the tag categorization)

`FEATUREPART` contains the feature-part model (another information that could improve the tag categorization)

`SNIPPET` specifies the snippets of the posts, a snippet is the atomic part of a sentence which contains the user's judgement

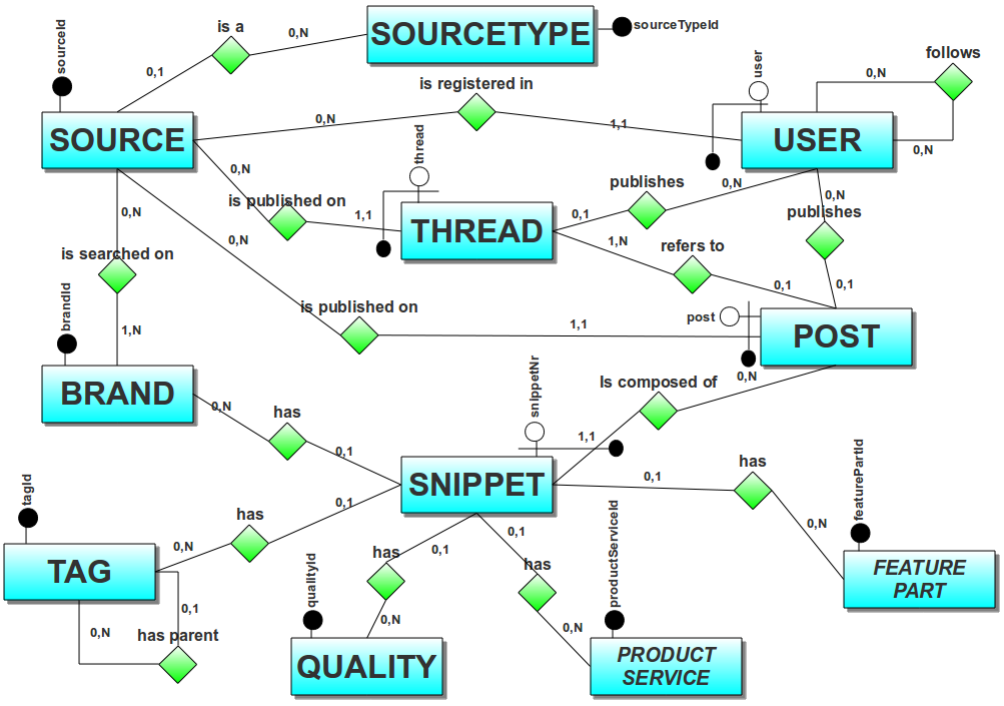


Figure 7.11: The global schema

7.4.2 Data warehouse approach

In order to meet the analysis needs, we also design a Data Warehouse that allows to aggregate the data and show them to the reputation analyst.

Fact identification

Analysing the integrated schema we derive that the fact is *Snippet*. Typically the Data Warehouse fact is a relation among the dimensions but in our case the table *Snippet*, is involved in relation among all the dimensions, as we can see from the ER schema of the integrated data base reported above.

The table *Snippet* contains information about the source and the post from which the snippet is taken, the text of the snippet, the sentiment value, the tag, brand and quality associated with the snippet after the sentiment analysis. All information that an analyst needs in order to understand the reputation. Counting the number of snippets which talk about a specific topic, the analyst will be able to understand how much people talk about this topic and if its judgement is positive or negative, thus understanding the causes and improving the negative aspects.

Attribute tree

According to the ER schema, it is possible to generate the Attribute Tree diagram in Figure 7.12. The root of this tree is *Snippet* and the child nodes are the entities taken from the ER diagram.

Starting from the Attribute Tree diagram, we can edit the tree pruning and grafting the appropriate nodes in order to obtain the tree which we will use for the Fact schema.

The measure which we defined are two: *Sentiment* and *Volume*. The glossary about how them are calculated is reported below.

```
Volume = COUNT(*)
Sentiment = AVG(SNIPPET.sentiment)
```

Beside the measures, the other nodes, which are children of the root, are defined as dimensions. We identify as dimensions the nodes *Date*, *Source*, *Tag*, *User*, *Quality*, *Post* and *Brand*.

Figure 7.12 shows the Attribute Tree edited, with the dimensions and the measures previously defined with the nodes coloured as indicated in the legend.

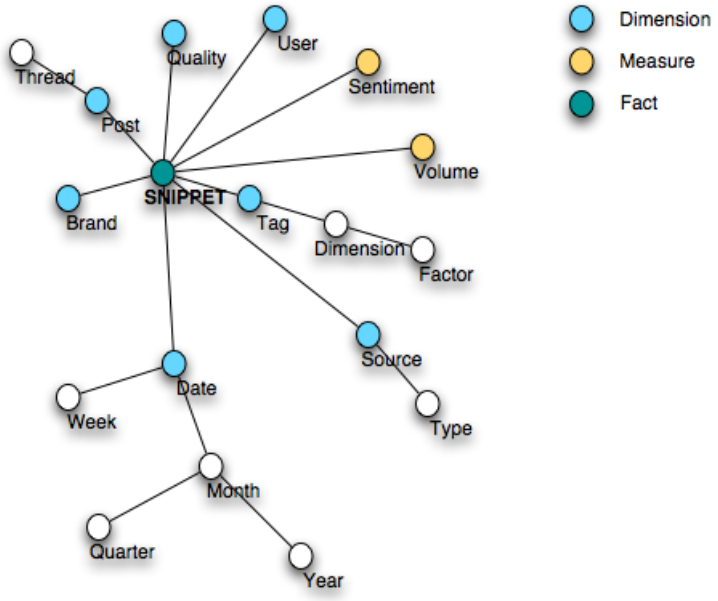


Figure 7.12: The attribute tree edited

Snowflake schema

The fact schema is a simple transformation of the Attribute Tree, therefore the dimensions, and all the root's child nodes, are reported and the measures are represented as attribute of the fact *Snippet*.

We choose to represent the fact schema with a snowflake schema. According with the diagram in Figure 7.13, the hierarchy of the dimension *Source* is respected but the hierarchy of the dimension *Date* is flatted because it is easier for the date to use SQL function instead of using keys and making join.

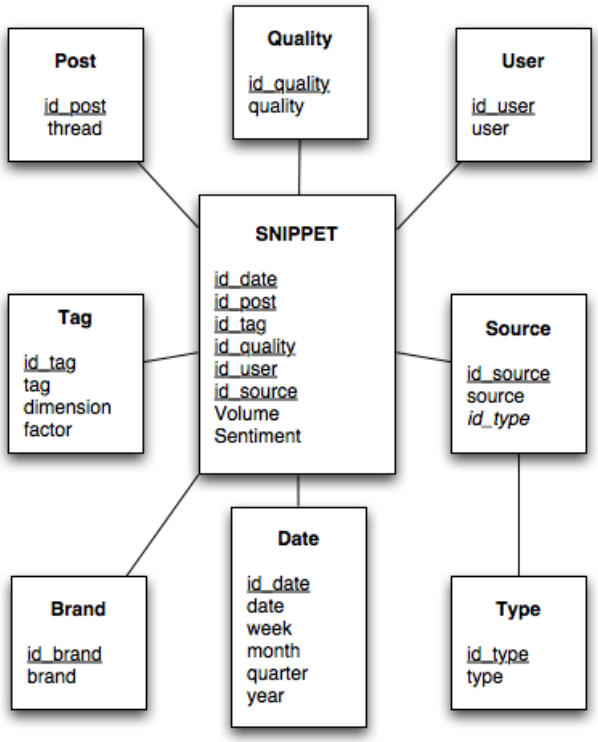


Figure 7.13: The snowflake schema

7.5 Domain customization

In order to meet the *Milan Municipality* requirements and in order to provide the analyses above described, starting from the architecture described in Chapter 6, some domain customizations are needed. In this Section we will describe these customizations.

7.5.1 Data Service

In our case study, in order to meet the *Milan Municipality* aims, we have to analyse data from a proprietary data source principally with data warehouse queries.
As explained in Section 6.6.1, Data Service has two sides a *client side* that is general and a *server side* that is strongly dependent to the data sources. Therefore the Data Service server side is implemented considering the particular schema described in the Section 7.4.1 and the queries are executed considering the global schema and the data warehouse project described in Section 7.4.2.

7.5.2 Custom bindings and compatibility matrix

In order to capitalize on the flexibility and the expressive power of mashups, the only default bindings are not enough to meet the analyses needs. The better solution is to synchronize the viewer components among them creating custom bindings. However, if we consider that bindings are relations between two components we will assert that the binding relation is not a total relation or rather is not possible to synchronize all the components with all the other components.

Components must be semantically and syntactically compatibles. In order to describe these compatibilities we have used a compatibility matrix (Figure 7.14) that maps which components are compatible with which of the other ones.

	Tag cloud	**List**	**Pie**	**Scatter**	**Tree map**	**Line**	**Line 3D**
Tag cloud	0	1	0	0	0	0	0
List	0	0	0	0	0	0	0
Pie	1	1	0	1	0	1	0
Scatter	0	1	0	0	0	0	0
Tree map	1	1	0	1	0	0	1
Line	0	1	0	0	0	0	0
Line 3D	1	1	1	1	1	0	0

Figure 7.14: The compatibility matrix

The compatibility matrix is implemented as a XML document that contains all the possible bindings between compatible components. Figure 13 reports the particular example implemented for the case study.

```
1  <matrix>

2  <binding source="RandomTagCloud" event="clickOnMe" target="List" operation="setParameters"/>

3  <binding source="PieChartHC" event="clickOnMe" target="ScatterHC" operation="setParameters"/>
4  <binding source="PieChartHC" event="clickOnMe" target="TimeVolumeHC" operation="setParameters"/>
5  <binding source="PieChartHC" event="clickOnMe" target="RandomTagCloud" operation="setParameters"/>
6  <binding source="PieChartHC" event="clickOnMe" target="List" operation="setParameters"/>

7  <binding source="ScatterHC" event="clickOnMe" target="List" operation="setParameters"/>

8  <binding source="TreeMapJIT" event="clickOnMe" target="List" operation="setParameters"/>
9  <binding source="TreeMapJIT" event="clickOnMe" target="ScatterHC" operation="setParameters"/>
10 <binding source="TreeMapJIT" event="clickOnMe" target="RandomTagCloud" operation="setParameters"/>
11 <binding source="TreeMapJIT" event="clickOnMe" target="TimeVolume3D" operation="setParameters"/>
12 <binding source="TreeMapJIT" event="clickOnMe" target="PieChartHC" operation="setParameters"/>

13 <binding source="TimeVolume3D" event="clickOnMe" target="List" operation="setParameters"/>

14 <binding source="TimeVolumeHC" event="clickOnMe" target="List" operation="setParameters"/>
15 <binding source="TimeVolumeHC" event="clickOnMe" target="RandomTagCloud" operation="setParameters"/>
16 <binding source="TimeVolumeHC" event="clickOnMe" target="PieChartHC" operation="setParameters"/>
17 <binding source="TimeVolumeHC" event="clickOnMe" target="ScatterHC" operation="setParameters"/>
18 <binding source="TimeVolumeHC" event="clickOnMe" target="TreeMapJIT" operation="setParameters"/>

19 </matrix>
```

Listing 13: The XML of the compatibility matrix

7.5.3 Components development

The component used in our case study are *ad-hoc* components and have been implemented using graphic libraries available on the Web, called *Highcharts* [7]. Initially the viewer components were generic viewers or rather were developed to read the result set document and to draw the charts preserving a generic behaviour. But in order to show the *Milan Municipality* analyses the viewer components have been customized. The viewer components, as described in Section 5.3, are components that should be generic and should visualize the results from the result set document. However if the analyses are domain specific and a general visualization is not enough, the viewer components should be customized. In our case study the visualization components have been modified in order to better meet the analyses needs.

7.6 Complete example

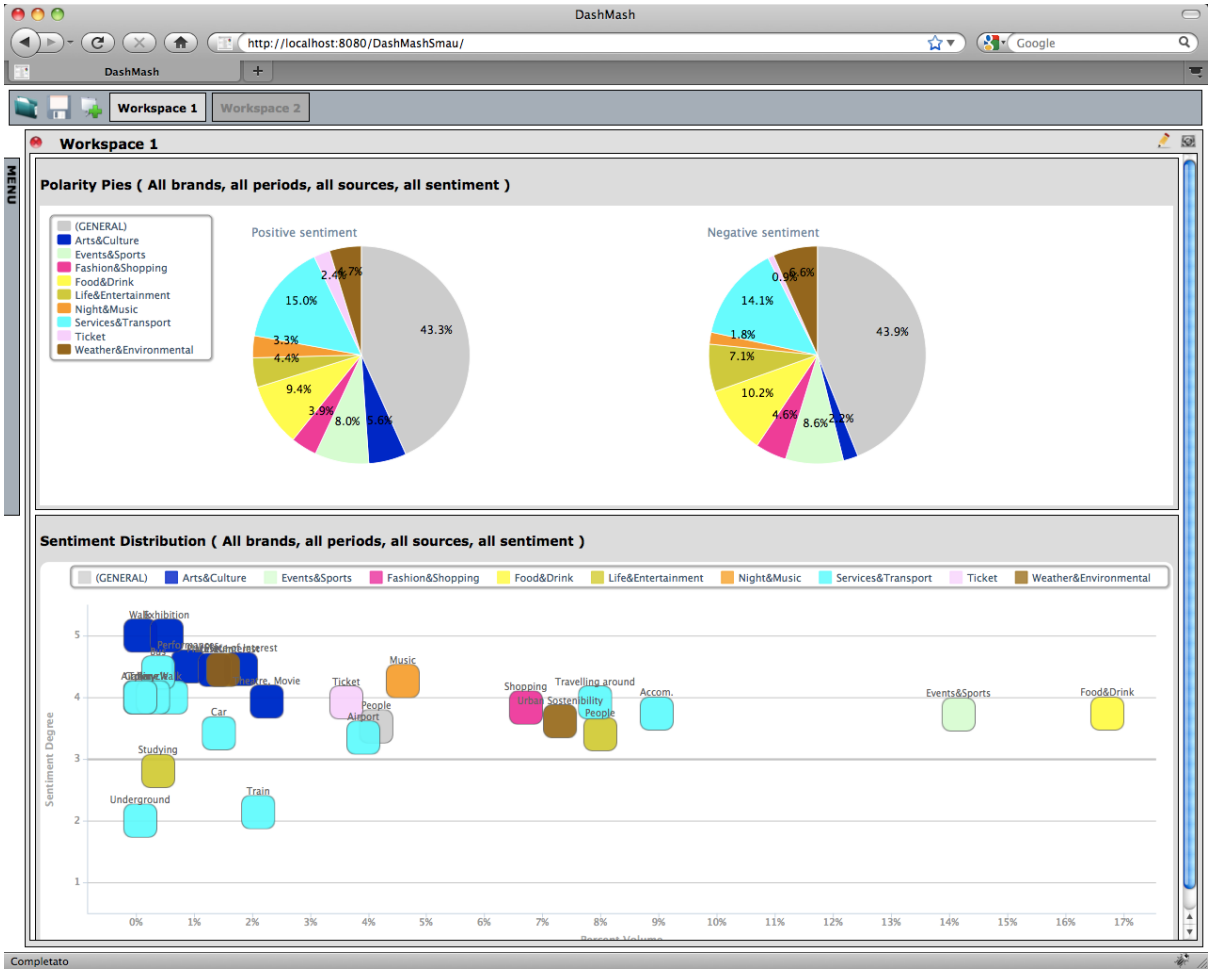


Figure 7.15: First mashup example

Figures 7.15, 7.16, 7.17 and 7.18 shows an example of use of the Web front-end of our platform, DashMash, for the composition of sentiment analysis mashups. A left hand

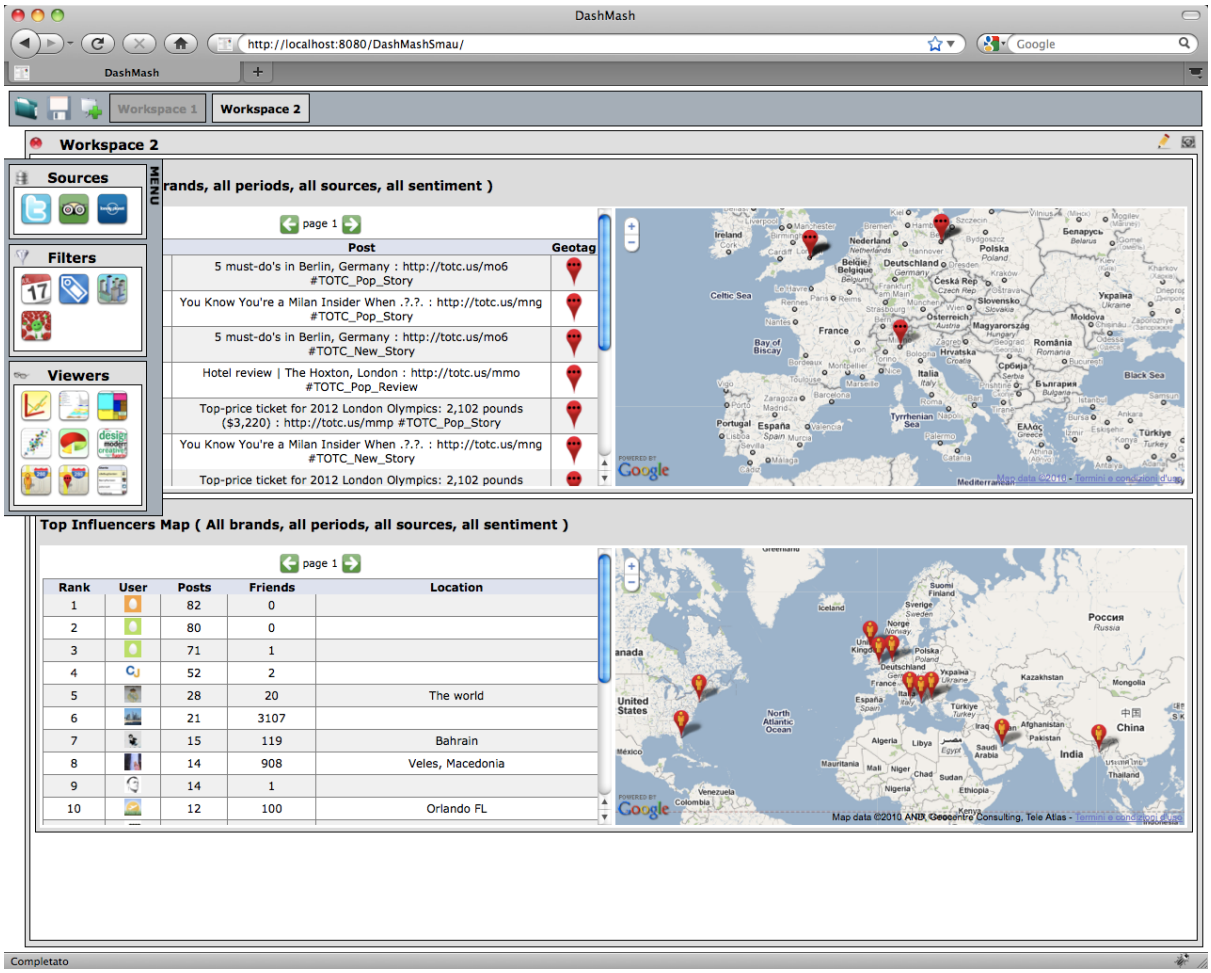


Figure 7.16: Second mashup example

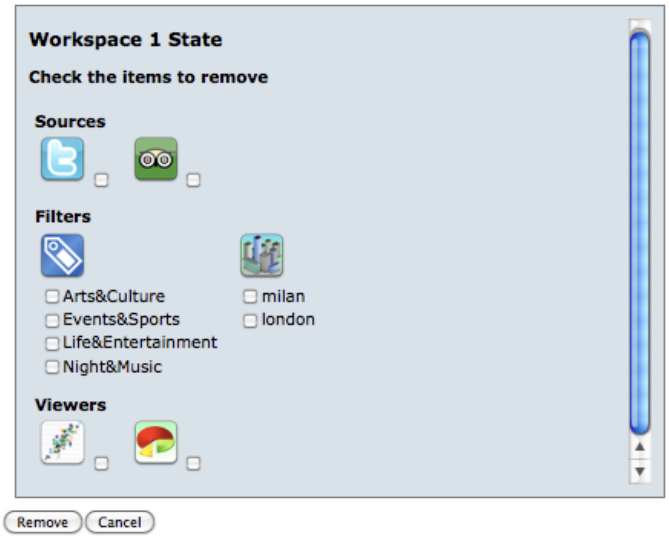


Figure 7.17: Status description

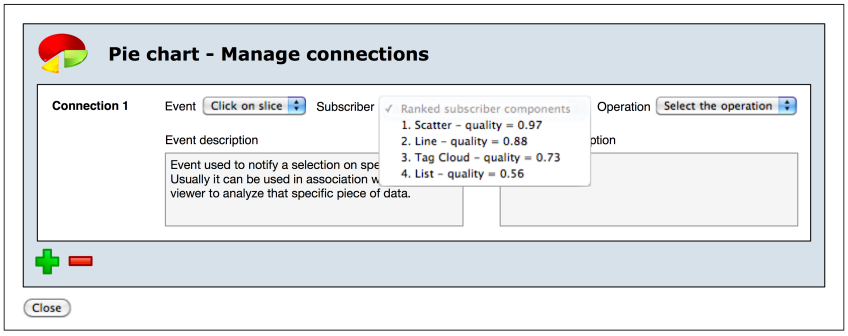


Figure 7.18: Custom binding dialog box

menu presents the list of components, currently data sources that materialize contents extracted from community sites, several types of filters, a multiplicity of viewers to visualize data, which are both open APIs, e.g, the Google APIs for maps and charts, and *ad-hoc* developed services, and utility open API/services, such as feed RSS FEEDS and calendars. Components can be mashed up by moving their corresponding icons into the so-called *workspaces*.

Figure 7.15 shows a mashup in which the user has selected two data sources, storing contents extracted from Twitter and TripAdvisor, and has filtered them by using a keyword based filter, with `key = Milan` [18]. The effect of this integration is that the workspace is now associated with the resulting data set. Contents are then presented adding a pie chart viewer, visualizing the percentage of comments related to categories of interest in the tourism domain (e.g., food, entertainment, art, and other relevant entities), and a scatter plot visualizing the average value of sentiment for the same set of categories.

Figure 7.16 shows a second mashup defined on top of the same data sources as the previous one. In this case, the filters select comments from users that are considered opinion leaders called *influencers*. Users are visualized through a list viewer, which is integrated with Google Maps to show the user location. A further synchronization with another map and another list viewer allows the user to see the original posts of each influencer, as well as the geo-localization of their posts, if available.

Users can iteratively modify the composition, by adding or dropping components. Their changes are enacted at real time. They can also access a text description of the status of the current composition (see Figure 7.17), and easily modify sources, filters, viewers or even configuration properties of single each filter or viewer.

Users can add further synchronization behaviors. For example, starting from the mashup shown in Figure 7.15, the dialog box presented in Figure 7.18 allows the user to set a coupling so that a click on a pie slice contextualizes the analysis offered by the map viewer to that selected label. As explained in Section 6.5, based on descriptive models of components, the dialog box presents the events exposed by the components selected by the user, plus a short description. The system provides suggestions about other candidate components based on compatibility rules and quality criteria.

Chapter 8

User testing

In order to validate the composition paradigm of DashMash with respect to EUD, we conducted a user based study. We have observed domain experts and naive users completing a set of tasks through our platform. Our goal was to assess how different skilled users would be able to easily develop a composite application.

The experiment specifically focused on the efficiency and intuitiveness of the composition paradigm, trying to measure this factors in terms of user performance, ease of use and user satisfaction. In particular, we expected all users to be able to complete the experimental tasks. However, we expected also a greater efficiency (e.g., reduced completion task times) and a more positive attitude (in terms of perceived usefulness, acceptability and confidence with the tool) by expert users. Their domain knowledge and background could indeed facilitate the comprehension of the experimental tasks, and improve the perception of the control over the composition method, and thus, their general satisfaction.

The following Sections describe the users sample, the test procedure, the experimental tasks and the questionnaires. Finally, we present the results analysis, deriving our conclusion about users performances, perceived ease of use and user satisfaction.

8.1 Users sample

The study involved 35 participants. Six of them were real end users, i.e., analysts and decision makers that are supposed to actually use DashMash for their analyses. In order to prove to which extent the tool was intuitive even for naive users, we also involved undergrad students of the Computer Engineering Programme at Politecnico di Milano which have a moderate knowledge about Web technologies, but that are never exposed to our tool neither to the sentiment analysis domain.

Table 8.1 describes the possible categories, which the users can be aggregated in, the metrics and the volumes of users (absolute and relative) for each category.

Table 8.1: Dataset descriptive statistics

Metric's category	Metric	Number	Percent
Subjects	cardinality	35	-
Gender	male	29	83.0
	female	6	17.0
Age	age ≤ 25	27	77.1
	$25 <$ age ≤ 30	7	20.0
	age > 30	1	2.9
Technology expert	expert	13	38.1
	non expert	22	62.9
Domain expert	expert	6	17.6
	non expert	28	82.4
Application designer	yes	27	77.1
	no	8	22.9
Time spent on the Internet	hours ≤ 4	8	22.9
	$4 <$ hours ≤ 8	20	57.1
	hours > 8	7	20.0

8.2 Procedure

For novice users, the completion of the experimental tasks was preceded by a 5-minute explanation about the domain and about the basic composition actions supported by the tool. Expert users were instead introduced to the set of available components and the basic composition mechanisms. All users were first asked to fill in a pre-test questionnaire – see Appendix A.2, to gather data on their knowledge about services and mashups. All users were then asked to perform two composition tasks. After the completion of the two experimental tasks, users were then asked to fill in a satisfaction questionnaire. In Appendix A.2 and A.2 are reported the pre-test and the post-test questionnaires.

About two thirds of the subjects (13) were skilled on mashups. Dataset descriptive statistics are summarized in Table 8.1.

After the completion of the two experimental tasks, the users were then asked to fill in a satisfaction questionnaire, which combined several items to measure two main dimensions:

1. *Ease of use*: we expected the paradigm facilitate user retention across tasks.

2. *Satisfaction*: we expected a more positive attitude (in terms of perceived usefulness, acceptability and confidence with the tool) by expert users. Their knowledge of the domain could indeed facilitate the comprehension of the experimental tasks, and increase the perception of the control over the composition method.

The third dimension, *performances*, was measured by clocking the tasks execution.

8.3 Experimental tasks

The two tasks were comparable in terms of number of components to be integrated and composition steps. Task2, however, required a less trivial definition of filters, to sift the involved data sources, and a more articulated definition of bindings. Also, while the formulation of Task1 was more procedural (i.e., it explicitly illustrated the required steps), Task2 just described the final results to be achieved, without revealing any details about the procedure required – see Appendix. A.1.

8.3.1 Task1

In the first task the user had to drag and drop the pie chart called, polarity pies, in order to visualize the percentage of volume of the positive and the negative sentiment. After that he/she had to filter the data with a tag filter on the tags *arts & culture*, *events & sports*, *fashion & shopping* and *food & drink*.
Than the user had to add, dragging and dropping, a second chart to visualize the sentiment distribution through a scatter chart, and then had to create a binding between the pie chart and the scatter chart, selecting from different options the publisher, the subscriber, the event that rises the binding and the operation involved. After that, the user was invited to click on a pie slice in order to enjoy the experience of the intra-components synchronization.
The last point in the task asked the user to compare the volume time-line distribution for the three city brands (Milan, London and Madrid) adding a line chart that shows this visualization and synchronizing the line chart with the pie chart inserted before.

8.3.2 Task2

The second task was articulated along only two point, as previously said, our intent was indeed it to increase the difficulty with respect to Task1 also giving less information about the composition procedure, does leaving the user free to chose how to proceed.

The first point of the task asked to add a line chart to show the comparison of the volumes filtered by the positive sentiment polarity.
The second point asked the user to add a tree map – to show the comparison among the volumes aggregated for brand and tag – and a list viewer – to show the opinion flow. Both these visualizers have to be synchronized with the line chart inserted in the first point of this task, with the aim of obtaining that when clicking on a point of the line chart the tree map and the list viewer update their visualization.

8.4 Results analysis

In the following sections we describe the results on related to taking into account the user performance in task completion, the time spent by users to perform the two tasks,

the perceived ease of use and the users satisfaction.

8.4.1 Performance

All the participants were able to complete both tasks without particular difficulties. Details about the time spent by users in the two groups are reported in Figure 8.1 . No differences in task completion time were found between experts and novices. In particular, domain expertise was not discriminant for task 1 ($p = .085$) and for task 2 ($p = .165$). Similarly, technology expertise was not discriminant for task 1 ($p = .161$) and for task 2 ($p = .156$). The lack of significant differences between the two groups does not necessarily mean that expert users performed bad. However, it indicates that the tool enables even inexperienced users to complete a task in a limited time. The average time to complete task 1 was about 2.5 minutes, while for task 2 it was about 2 minutes. This positive result is not surprising since novices can perform as good as experts even in the case of Web searches [38, 53].

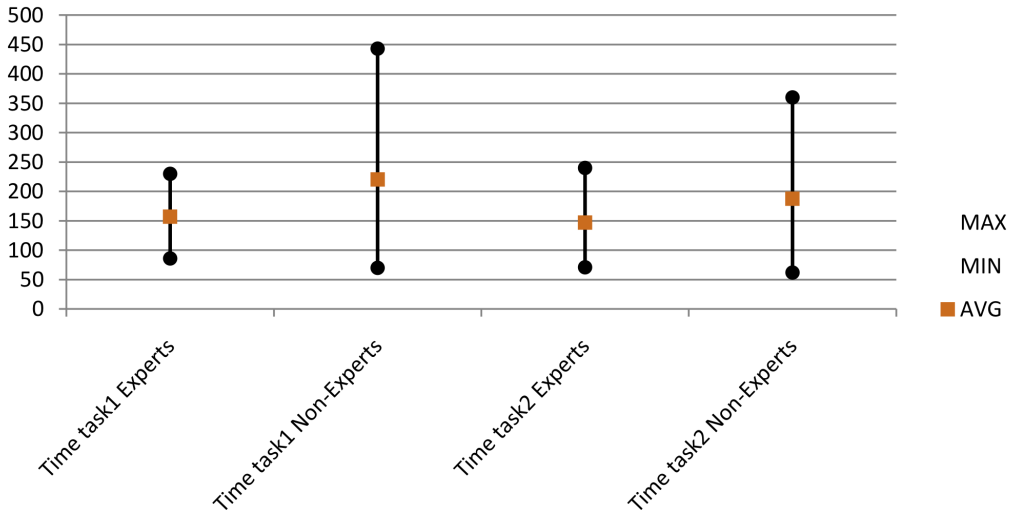
The difference in completion times for the two tasks can be also used as a measure of learning [33]. This difference is about half a minute ($t = 28.2, p = .017$), i.e., a reduction of about 15%. This result highlights the learnability of the tool: although the second task was more critical compared to the first one, subjects were able to accomplish it in a shorter time.

8.4.2 Ease of use

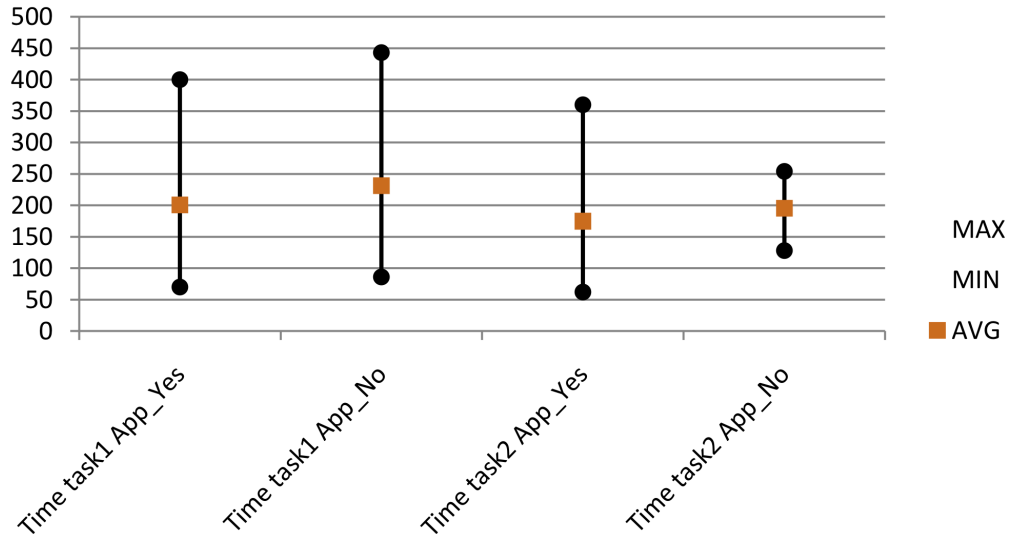
The ease of use was further confirmed by the data collected through four questions in the post-questionnaire, asking users to judge whether they found it easy to identify and include services in the composition, to define service bindings between services, and to monitor and modify the status of the mashups. We also asked users to score the general ease of use of the tool. Users could modulate their evaluation on a 7-point scale. The reliability of the ease of use questions is satisfying ($\alpha = .75$). The correlation between the four detailed questions and the global score is also satisfying ($\rho = .58, p < .001$). This highlights the high external reliability of the measures. On average, users gave the ease of use a mark of 1.77 (the scale was from 1 very positive to 7 very negative). The distribution ranged from 1 to 4 ($mean = 1.77, meanS.E. = .12$). We did not found differences between novice and expert users. This was especially true for perceived usefulness ($p = .51$). The detailed ease of use score collected for the two users groups are reported in Figure 8.2.

8.4.3 Satisfaction

The user satisfaction with the composition paradigm was assessed using two complementary techniques. A semantic-differential scale required users to judge the method on 12 items. Users could modulate their evaluation on a 7-point scale (1 very positive - 7 very negative). Moreover, a question asked users to globally score the method on a 10-point

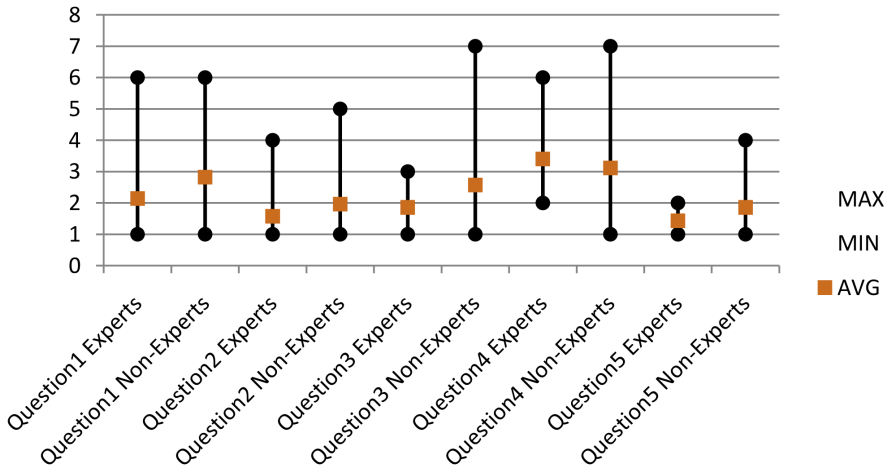


(a) Domain experts

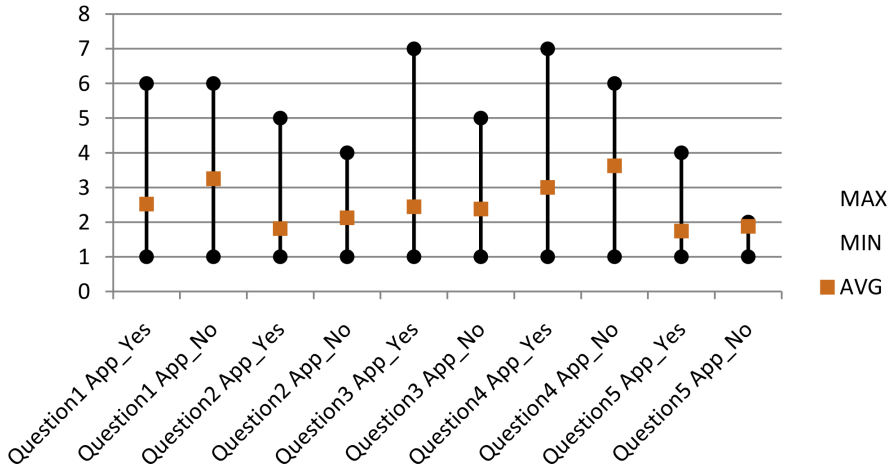


(b) Application developers

Figure 8.1: Task completion times for the two user groups



(a) Domain experts

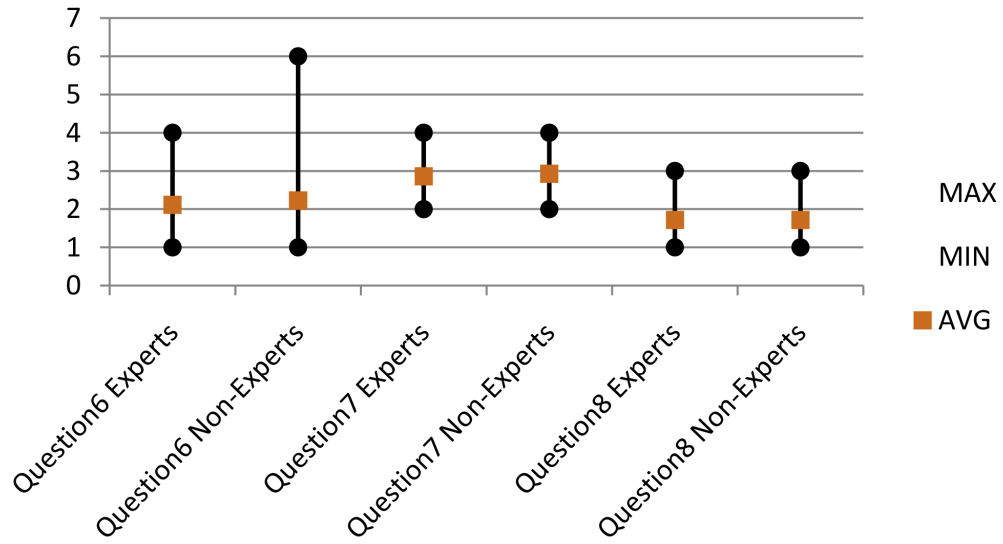


(b) Application developers

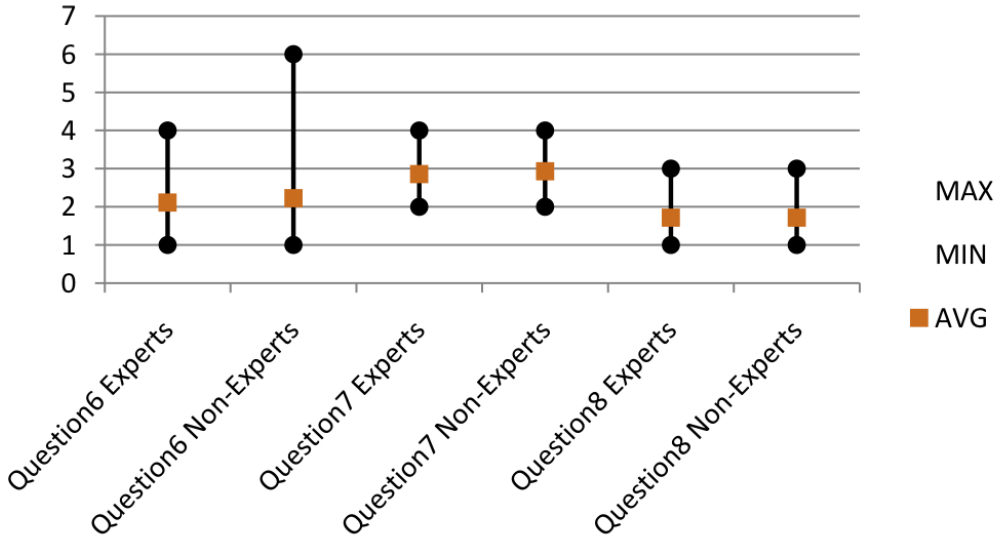
Figure 8.2: Ease of use scores for the two user groups

scale (1 very positive - 10 very negative). The reliability of the satisfaction scale is satisfying ($\alpha = 0.76$). Therefore, a user-satisfaction index was computed as the mean value of the score across all the 12 items. The average scores given by the two user groups for the satisfaction question are reported in Figure 8.3.
The average satisfaction value is very good ($min = 1.3, max = 3.5, mean = 2.2, meanS.E. = .09$). The correlation between the average satisfaction value and the global satisfaction score is satisfying ($\rho = .41, p < .015$). On average, users gave the composition method a mark of 2.9, with a distribution ranging from 2 to 4. We did not find differences between experts and novices. Despite our initial assumption, we found that the ease of use of the tool is perceived in the same way by novice and expert users, although the latter have greater domain knowledge. The moderate correlation between the satisfaction index and the ease of use index ($\rho = .55, p = .011$) also reveals that who perceived the method as easy also tended to evaluate it as more satisfying. This confirms that ease of use is perceived.

The last two questions asked users to judge their performance as mashup developers and to indicate the percentage of requirements they believed to have satisfied with their composition. This metric can be considered as a proxy of confidence [33]. On average, users indicated to be able to cover the 91% of requirements specified by the two experimental tasks ($min = 60\%, max = 100\%, meanS.E. = 1.7\%$). They also felt very satisfied about their performance as composers ($mean = 1.8, meanS.E. = .13$; 1 - very positive, 4 - very negative). We also found a direct correlation between the users perception of their performance as mashup developers and the global ease of use ($\rho = .57, p < .001$), meaning that the tool's ease of use improves user confidence.



(a) Domain experts



(b) Application developers

Figure 8.3: Results about satisfaction

Part V

Conclusions

Chapter 9

Summary and future works

This thesis has focused the design of a mashup platform with an innovative composition paradigm, specifically conceived to encourage end-user development through a lightweight mashup development process. Starting from some high-level abstractions, expressed by suitable components and composition models, we have shown how to ease the mashup definition by end users. In particular, we have introduced a Web-based architecture that enables the automatic, system-guided definition of service couplings for data flow and service synchronization control, while the users are still able to define their custom bindings.

In order to allow also non-programming skilled users to easier accomplish their composition aims, our work provides:

- An execution engine that merges the composition and the execution phases, showing instantaneously the mashup resulting from any incremental composition action.
- The automatic generations of the composition descriptors (SMDL and XPIL);
- Support for compatibility and quality check through the generation of recommendations during the definition of custom bindings.

We have also specialized the DashMash platform to cope with the requirements posed by a project funded by the Milan Municipality. This has allowed us to identify further requirements posed by enterprise mashups, where *data services* are needed to make local data warehouse adhere to the *data-as-a-service* paradigm.

In order to validate our hypothesis about the ease-of-use of the composition paradigm and its adequateness with respect to the end user development requirements, we conducted a user testing involving a sample of 36 users. The results are very positive, since they demonstrate how our solution allows the users to perform composition tasks effectively, also increasing the user perception of the platform ease of use and satisfaction.

As a final observation, it is worth noting that part of our work investigated how DashMash could be specialized for enterprise mashups, where the application domain and the component characterization are easier to identify and represent in form of rules for the automatic composition. However, the proposed approach remains valid for the compo-

sition of generic mashups. Of course, the lower the availability of domain descriptions, the greater the need for users to explicitly define service couplings. Nevertheless, as confirmed by the user-based study, the coupling definition mechanism is still intuitive and further facilitated by quality-based recommendations.

9.1 Future works

During the experiment, users provided suggestions about possible improvements of the platform. One comment suggested the need for easy mechanisms for component creation starting from generic services. While developing the components for sentiment analysis, we realized that the "componentization" process is still a demanding task. Although we believe that the preparation of components by expert developers is acceptable in an enterprise context, we are currently working at a wizard through which the user can create the presentation layer for services not equipped with a user interface. The service description and wrappers are then automatically created. The first results, concerning described services (i.e., WSDL and Linked Data), are encouraging. Our current work is devoted to improving this feature.

As future work, we also aim at exploring different composition solutions, to address, for example, the cooperative definition of mashups (a feature that can greatly enhance team-based cooperation), as well as an extension of the recommendations mechanisms based on the emergence of composition patterns from the community's mashups [48]. We also aim at investigating mashup interoperability, for example making DashMash mashups compatible with emergent standards, such as Enterprise Mashup Markup Language (EMML) [45].

9.2 Achievements

The results of this thesis related to the production of recommendations have been presented at the International Workshop *Composable Web 2010*, held at Vienna in July 2010, and published in the book *Current Trends in Web Engineering* published by Springer Verlag, LNCS series.

A paper, describing the new composition paradigm and the overall architecture is under revision for the conference WWW 2011.

In recognition of the innovative aspects promoted by the specialization of DashMash for sentiment analysis, the City of Milan has been awarded with the **Smau Innovation in Enterprise 2.0** price[1].

[1]Major details about the price are available at `http://www.dei.polimi.it/news/dettaglio.php?&id_elemento=151&idlang=eng`

Part VI

Appendix

Appendix A

Test materials

A.1 Experimental Tasks

During the experiment, user were asked to complete the following two tasks.

1. Create a first workspace in which:

   - A pie chart visualizes the percentage volume of the positive and negative sentiment. The visualization must show data relative to the tags: “arts & culture”, “events & sport”, “fashion & shopping”, “food & drink”.

   - A scatter visualizes the sentiment distribution, and is synchronized with a slice selection of the pie chart (one you think is interesting).

   - A line chart to compare the volume distribution in time for the three city brands (Milan, London, Madrid), synchronized with the selection of a slice of the pie chart (the same selected in the scatter).

2. Create a second workspace in which:

   - A line chart shows only the positive sentiment.

   - A tree map and a list viewer are synchronized with the line chart, so that a click on a point of a series of the line chart causes the synchronization of the tree map with the list viewer.

A.2 Pre-questionnaire

CODE _____

1. Do you think that the usage of a social application (like Facebook) can cause privacy problems on your personal data?

   Yes ☐ No ☐

   If yes, how much?

	very	quite	a little	neither of both	a little	quite	very	
Much	**1**	**2**	**3**	**4**	**5**	**6**	**7**	None

2. How many hours do you use the computer a day?
   |_____|

3. Are you a regular Web user? Yes ☐ No ☐

4. Have you ever tried to create Web applications? Yes ☐ No ☐

5. Have you ever used a tool to aggregate different services (e.g, I-google, Yahoo!Pipes)?
   If "yes", which ones?

_____
_____
_____

6. Do you know what is a mashup?
If yes, try to define it:

_____
_____
_____
_____

Social-personal data

1. Age |_____|

2. Sex Male ☐ Female ☐

A.3 Post-questionnaire

CODE _____

Questionnaire

We thank you for your collaboration. We ask you to express your impressions and opinions about this tool and on the Web application creation process you have just used. The questionnaire is strictly anonymous and it will not influence at all your academic career. We will consider every information you provide strictly confidential. We invite you to be as much sincere and detailed as possible. Feel yourself free to ask for explanations if any question is not completely clear.

1. To identify the necessaries components to create the Web application has been:
*(Please, mark **one and only one** number corresponding to your opinion)*

	very	quite	a little	neither of both	a little	quite	very	
Easy	**1**	**2**	**3**	**4**	**5**	**6**	**7**	Difficult

2. To define a binding logic between components has been:

	very	quite	a little	neither of both	a little	quite	very	
Easy	**1**	**2**	**3**	**4**	**5**	**6**	**7**	Difficult

3. To monitor the composition state (e.g., services, filters, bindings in a workspace) has been:

	very	quite	a little	neither of both	a little	quite	very	
Easy	**1**	**2**	**3**	**4**	**5**	**6**	**7**	Difficult

4. In case of errors, how it is easy to undo the previous actions?

	very	quite	a little	neither of both	a little	quite	very	
Easy	**1**	**2**	**3**	**4**	**5**	**6**	**7**	Difficult

5. In general, to learn to use the tool has been**:**

	very	quite	a little	neither of both	a little	quite	very	
Easy	**1**	**2**	**3**	**4**	**5**	**6**	**7**	Difficult

6. The development method that you have just used seems to you:
*(For every adjectives pairs, please mark **one and only one** number corresponding to your opinion)*

	very	quite	a little	neither of both	un po'	a little	very	
easy to use	**1**	**2**	**3**	**4**	**5**	**6**	**7**	difficult to use
Hard	**1**	**2**	**3**	**4**	**5**	**6**	**7**	relaxing
Clear	**1**	**2**	**3**	**4**	**5**	**6**	**7**	ambiguous
Useful	**1**	**2**	**3**	**4**	**5**	**6**	**7**	useless
Challenging	**1**	**2**	**3**	**4**	**5**	**6**	**7**	easy
Inefficient	**1**	**2**	**3**	**4**	**5**	**6**	**7**	efficent
Funny	**1**	**2**	**3**	**4**	**5**	**6**	**7**	boring
unreliable	**1**	**2**	**3**	**4**	**5**	**6**	**7**	reliable
effective	**1**	**2**	**3**	**4**	**5**	**6**	**7**	inefficacious
nice	**1**	**2**	**3**	**4**	**5**	**6**	**7**	unpleasant
flexible	**1**	**2**	**3**	**4**	**5**	**6**	**7**	rigid
satisfying	**1**	**2**	**3**	**4**	**5**	**6**	**7**	unsatisfying

7. Rate the tool and the composition mechanism, using a range from 1 (*very negative*) to 10 (*very positive*)

(Write your rate in the apposite space) |_____|

8. How are you satisfied about your performance as "Web application composer"?
*(Please, mark **one and only one** number corresponding to your opinion)*

Very	**Quite**	**A bit**	**None**
1	**2**	**3**	**4**

9. In percentage, how many requirements of the task you have performed with our tool do you think are covered by the application you developed?

0%	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**

10. In your opinion, what is the main advantage of our tool?

_____
_____
_____
_____
_____

11. And what is its main limit?

_____
_____
_____
_____
_____

12. Looking at the tool functionalities, do you think it is complete?

	very	quite	a little	neither of both	un po'	a little	very	
Complete	**1**	**2**	**3**	**4**	**5**	**6**	**7**	Incomplete

If incomplete, what do you want to suggest?

_____
_____
_____
_____

13. How much the knowledge on mashup and integration of services can facilitate in tool usage?

A lot	Quite	A little	None
1	**2**	**3**	**4**

Please check if you have answered every question. Thanks for your collaboration!

DashMash: A Mashup Environment for End User Development

Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Gabriele Sprega, Donato Barbagallo, and Chiara Francalanci

Politecnico di Milano - DEI
P.zza Leonardo da Vinci, 32 - 20133 - Milano - Italy
{name.surname}@polimi.it

Abstract. Web mashups are a new generation of applications based on the "composition" of ready-to-use services. In different contexts, ranging from the consumer Web to Enterprise systems, the potential of this new technology is to make users evolve from passive receivers of applications to actors actively involved in the "creation of innovation". Enabling end users to self-define applications that satisfy their situational needs is emerging as an important new requirement. In this paper, we address the current lack of lightweight development processes and environments and discuss models, methods, and technologies that can make mashups a technology for end user development.

Keywords: mashups, service-based dashboards, end user development.

1 Introduction

Web mashups are a new generation of tools that support the "composition" of applications starting from services and contents oftentimes provided by third parties and made available on the Web. Mashups were initially conceived in the context of the consumer Web, as a means for users to create their own applications starting from public programmable APIs or contents taken from Web pages. However, the vision is towards the development of more critical applications, for example the so-called *enterprise mashups* [10], a porting of current mashup approaches to company intranets. The potential flexibility of mashup environments can help people help themselves [20], by enabling the on-demand composition of the functionalities that they need. Mashups are therefore emerging as a technology for the creation of innovative solutions, able to respond to the different problems that arise daily in the enterprise context, as well as in any other context where flexibility and task variability are dominant requirements.

Given the previous premises, the need arises to provide mashup environments where the end users (i.e., the main actors of this new development process) can easily and quickly self-construct their applications without necessarily mastering the technical features related to service invocation and integration. This is true in every context - not only in the Enterprise: a "culture of participation" [8], in which users evolve from passive consumers of applications to active co-creators of new ideas, knowledge, and products, is indeed more and more gaining momentum

S. Auer, O. Diaz, and G.A. Papadopoulos (Eds.): ICWE 2011, LNCS 6757, pp. 152–166, 2011.

1.1 Contributions and Paper Outline

What makes mashups different from plain Web service compositions is their potential as tools through which end users are empowered to develop their own applications. However, this potential is rarely exploited. So far the research on mashups has focused on enabling technologies and standards, with little attention on easing the mashup development process - in many cases mashup creation still involves the manual programming of the service integration. Some recent user-centric studies [14] also found that, although the most prominent mashup platforms (e.g., Yahoo!Pipes, Dapper or Intel Mash Maker) simplify the mashup development, they are still difficult to use by non technical users. In this paper, we try to respond to the need of easying the mashup development, and propose a Web platform, *DashMash*, that allows end users to develop their own mashups making use of an intelligible paradigm that abstracts from technical variables. In particular:

1. Through a case study, we provide a scenario in which general mashup composition principles can be applied to the construction of applications targeting the experts (e.g., analysts and decision makers) of a given domain (Section 2). Based on this scenario, we outline relevant factors supporting end user development. First of all, the importance of a *lightweight development process*, in which mashup composition paradigms are embedded in usable visual environments hiding the complexity of the composition languages actually managing the execution of the mashup.
2. We present a runtime architecture supporting the lightweight development processes, which increases the user's control over the mashup composition process (Section 3). This is possible thanks to (i) an instant execution support, based on the "on the fly" interpretation of the user composition actions and the immediate execution of the mashup composition in a WYSIWYG (What You See Is What You Get) manner (Section 3.2), and (ii) the automatic generation of descriptive models for the results of the users composition actions (Section 3.3), which then drive the execution of the mashup.
3. We promote the adoption of *recommendation mechanisms* that take into account quality variables to help end users select data sources and mashup components and composition patterns (Section 3.3). This is enabled by the enrichment of descriptive models with annotations specifying also non-functional user requirements.
4. Based on the results of a usability experiment, we discuss the effectiveness of our approach from an end user perspective (Section 4).

2 Case Study: Mashups for Sentiment Analysis

In the context of a project funded by the Comune di Milano (Milan Municipality), we have worked on the construction of a Web platform through which end users can construct their dashboards for *sentiment analysis*[1]. Sentiment analysis

[1] A demo is available at
http://home.dei.polimi.it/cappiell/demo/DemoDashMash.mov

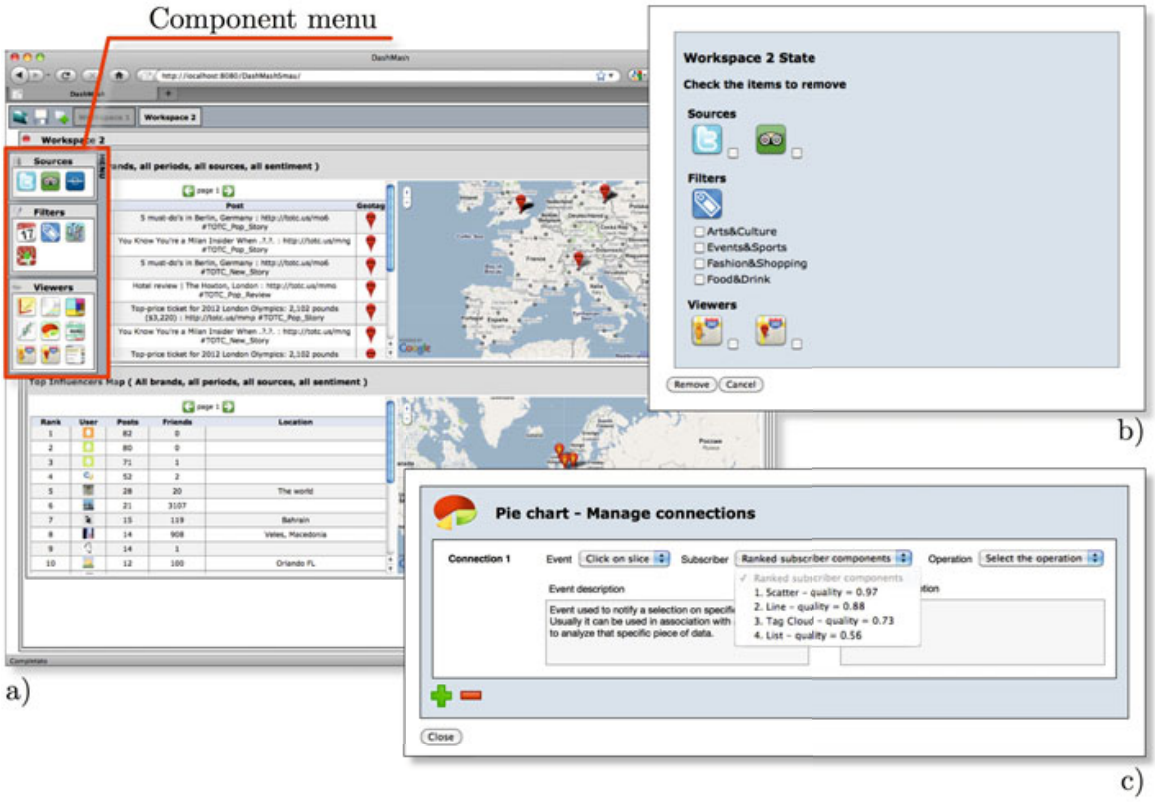


Fig. 1. Example of mashup composition for a sentiment analysis dashboard

focuses on understanding market trends starting from the unsolicited feedback provided by users comments published on the Web. Our project focuses on the design of an engine in charge of the automatic extraction of sentiment indicators summarizing the opinions contained in user generated contents [1], and on the provision of a Web environment where analysts can self-construct their analyses.

After preliminary attempts with a traditional static dashboard, we realized that end users could benefit from the ability to compose their analysis flexibly, playing in variable ways with sentiment indicators, and also complementing such indicators with "generic" external Web resources. This latter feature would indeed help them to improve their analyses, by interpreting sentiment indicators with a view on the events that cause trends and behaviors.

Figure 1 shows an example of use of the Web front-end of our platform, DashMash. A left hand menu presents the list of components, which for the sentiment analysis domain are *data sources* that materialize contents extracted from community sites, several types of *filters*, a multiplicity of *viewers* to visualize data, which are both open APIs, e.g, the Google APIs for maps and charts, and ad-hoc developed services[2], and utility open API/services, such as RSS feeds and calendars. Components can be mashed up by moving their corresponding icons into the so-called *workspaces*. Each workspace is associated with a data set

[2] Several charts offering advanced data visualizations have been developed using the Highcharts JS library (http://www.highcharts.com/).

resulting from the integration of data sources and filters, and renders this data set according to the visualizations offered by the selected viewers.

In the mashup shown in Figure 1a) the user has selected two data sources storing users comments extracted from two well-known social applications, Twitter and TripAdvisor. A filter is applied to select the only comments from users that are considered opinion leaders, so-called *influencers*. Influencers data are visualized through a *list viewer*, which is integrated with *Google Maps* to show the influencers locations. A further synchronization with another map and another list viewer allows one to see the original posts of each influencer, as well as the geo-localization of their posts, if available.

Users can iteratively modify the composition, by adding or dropping components. Changes are enacted at real time and the effect are immediately shown. They can also access a visual description of the status of the current composition (see Figure 1b), and easily modify sources, filters, viewers or even configuration properties of each single component.

Although the system automatically includes some default bindings to ensure basic inter-component synchronization, through simple dialog boxes the users can create new service combinations resulting into synchronized behaviors. For example, starting from the mashup shown in Figure 1a, the dialog box presented in Figure 1c allows the user to add a pie-chart viewer to show the distribution of influencers comments along different topics, and set a coupling so that a click on a pie slice contextualizes the analysis offered by the map viewer to that selected portion of data. Also, based on compatibility rules and quality criteria, the system provides suggestions about other candidate components to extend the mashup or replace existing components.

As shown by the previous example, the DashMash environment is characterized by factors that try to alleviate as much as possible the end users during the mashup composition task [14,8]:

- *Abstraction from technical details*: the representation of services as visual objects that abstract from technical details (e.g., their programmatic interface), the immediate feedback on composition action, and the immediate execution of the resulting mashup to reveal the service look&feel, help users realize the service functionality and the effect that the service has on the overall composition. In the end, users are asked to manipulate (e.g., add, remove or modify) visual objects focusing on the service visualization properties rather than technical details of service and composition logics. As also confirmed by a user-based experimentation (see Section 4), this increases user satisfaction and the user-perceived control over the composition process.
- *Composition support*: the composition task is guided in multiple ways. On the one hand, starting from components' descriptive models, the composition engine is able to infer and automatically create *default bindings* between the services that the users add to the composition. Compatibility rules and quality criteria are then adopted to "rank" available components and provide users with suggestions about *additional components* and *custom bindings*.

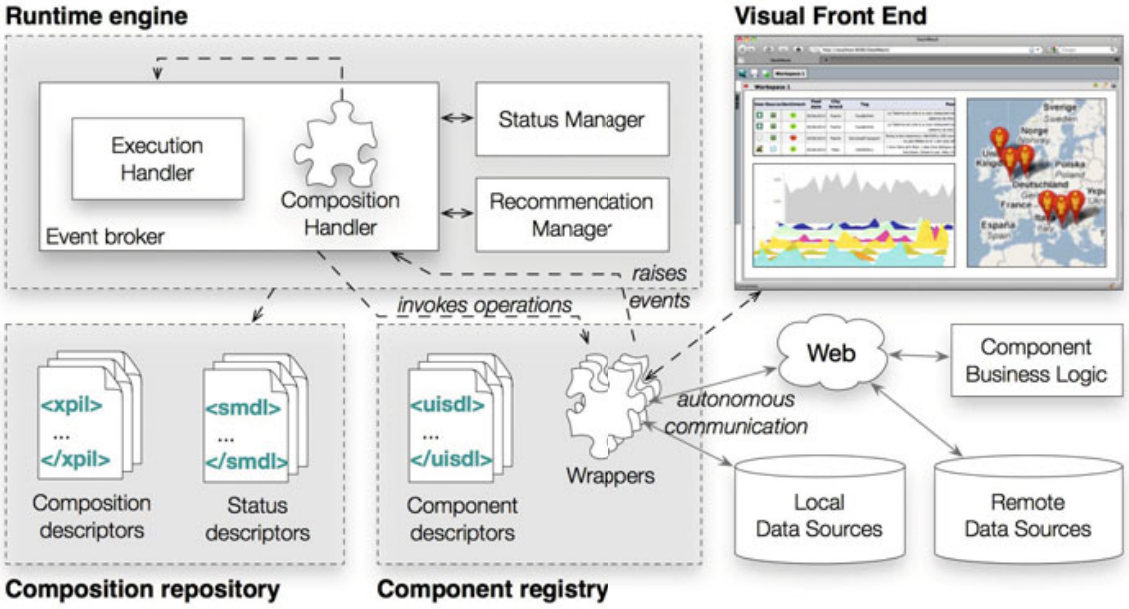


Fig. 2. Organization of the DashMash architecture

– *Continuous monitoring*: users are provided with mechanisms that allow them to understand the current state of the composition and to explore options about how to complete or extend the current composition.

The following sections are devoted to clarifying the modelling abstractions and the architectural features that allowed us to implement the previous requirements in DashMash.

3 The DashMash Platform

As represented in Figure 2, the organization of DashMash is centered around a lightweight paradigm in which the orchestration of registered services, the so called *Components*, is handled by an intermediary framework, in charge of managing both the *definition* of the mashup composition and the *execution* of the composition itself. Different from the majority of mashup platforms, where mashup design is separate from mashup execution, in DashMash the two phases strictly interweave. The result is that composition actions are automatically translated into models describing the composition; these models are immediately executed. Users are therefore able to *interactively* and *iteratively* define and try their composition, without being forced to manage complicated languages or even ad-hoc visual notations.

The current implementation of the DashMash runtime engine consists of a client-side (JavaScript) application that supports an *event-driven* execution paradigm. As represented in Figure 2, an *Event Broker* intercepts events, which can refer to users and system actions occurring during mashup execution (e.g., the click on a pie-chart slice which cause a change in a map), and to the dynamic definition of the composition (e.g., the drag&dop of a component icon into a workspace). The Event Broker then dispatches the events to the modules

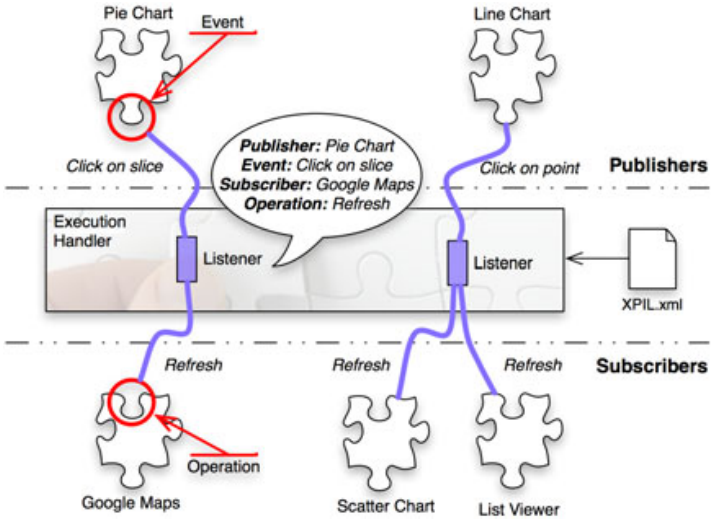


Fig. 3. Event-driven paradigm for service coupling definition and mashup execution

in charge of their handling, based on models and mechanisms that we explain in the rest of this section.

3.1 Event-Driven Execution

Events occurring during mashup execution are managed by an *Execution Handler* based on a *publish-subscribe model* addressing component integration at presentation level [21]: *events* generated from the user interaction with one mashup component (e.g., the selection of a slice in a pie chart) can be mapped to *operations* of one or more components that subscribe to such events (e.g., the visualization of details of the selected data in a scatter plot).

As represented in Figure 3, service couplings are expressed through the definition of the so-called *listeners* in a *composition model* expressed according to an XML-based language, *XPIL* [21]. During mashup execution, each component keeps running according to its own application logic, within the scope defined by an HTML `<div>`. As illustrated in Figure 3, as soon as events occur, the involved components publish them. Based on the defined listeners, the Execution Handler then notifies the subscribed components, if any, and triggers the execution of their corresponding operations.

This composition and execution logics requires each component to be characterized by a high-level model expressing the *events* that the component can generate, the *operations* that enable other components to modify its internal state, and the binding with the actual service/API, so that operations can be invoked. Therefore, as illustrated in Figure 2, for each registered component the component registry stores a *descriptor*, expressed according to the *UISDL* language [21], and a *wrapper*, in charge of rasing events and invoking operations. This component model provides a uniform paradigm to coordinate the mashup composition and execution, which obviates the heterogeneity of service standards and formats, also hiding the intrinsic complexity of services and composition.

3.2 Managing Composition

The *Composition Handler* manages composition events. In particular, as better explained in Section 3.3, it automatically translates the addition of a component into new *default listeners* and creates or updates (if already existing) the current composition model accordingly. The Composition Handler in turns dispatches composition events to the *Status Manager*, a module in charge of maintaining a *mashup state* representation. Combined with the composition model, the mashup state is useful to recover a previously defined mashup for a later execution, but especially to let users monitor their composition and modify it on the fly. State variables relate to default or specific parameter values (e.g., the value of a parameter for querying a data source), to layout properties (e.g., the colors used to show values on a chart) or to any other property that the user can set to control the component data and appearance. Composition events are also dispatched to the *Recommendation Manager*, a module in charge of providing suggestions about further components to select for extending/completing a composition (see Section 3.3). After terminating the handling of such events, the mashup composition is reloaded and immediately rendered into the workspace. The mashup is then executed according to the event-driven, publish-subscribe logic that characterizes the Execution Handler.

It is worth noting that the Composition Handler itself is a mashup component: any user composition action generates a Composition Handler's event, which is notified to and managed by the Execution Handler. An interesting side effect of this architectural choice is that the logic behind the automatic component coupling is "programmable" and, therefore, flexible: it depends on a set of predefined listener templates configured inside the Composition Handler which, being the Composition Handler a mashup component, can in turn be easily unplugged and/or replaced.

3.3 Definition of Listeners

DashMash supports the definition of *default* and *custom* listeners. The former are automatically defined by the Composition Handler when a composition action is intercepted. This ensures a minimum level of inter-component synchronization that does not require users to define service coupling. Custom bindings are instead user-defined. Nevertheless, the Composition Handler offers also support by generating compatibility– and quality– based recommendations. To this aim, it dispatches the composition events to the *Recommendation Manager* that is in charge of evaluating the quality of the current composition and provides suggestions about the selection of possible components to add or substitute to the existing ones in order to achieve or improve the mashup quality.

Default Listeners. To enable the automatic definition of default listeners, we start from a classification of components. For example, in order to facilitate the construction of dashboards, it is possible to identify the following classes of components (Figure 4):

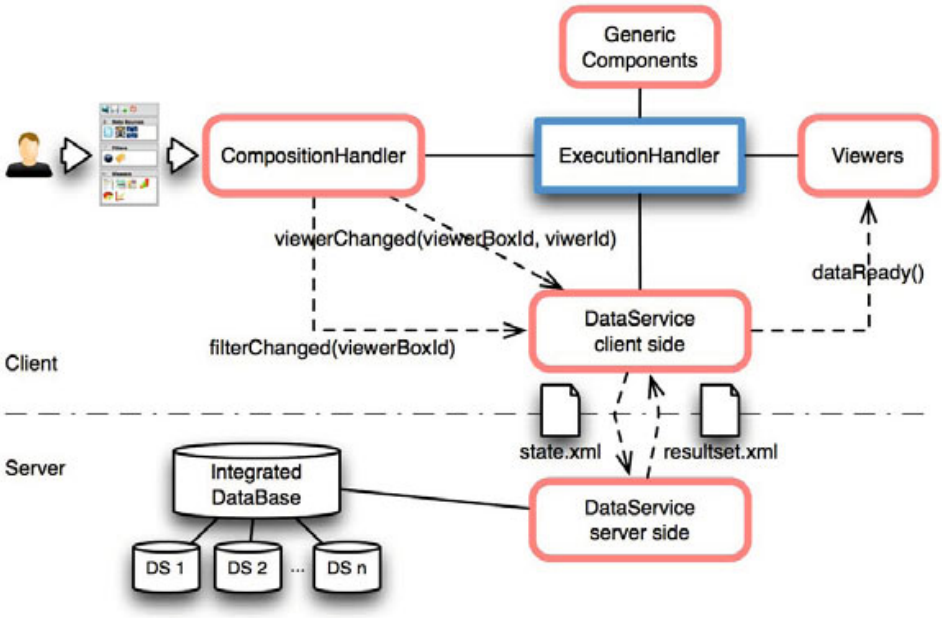


Fig. 4. Classification of components and their mutual interactions

- *Data services* are in charge of retrieving data from data sources. They are especially meant to provide access to internal (relational) data sources/ warehouses. For example, in the sentiment analysis domain internal data sources store data extracted from different social applications where users post their comments. A server-side module is in charge of accessing the data sources[3]. A client-side module provides the actual component that synchronizes with the other mashup components to intercept changes of the composition state and send pertinent information to the server-side module so that queries for data extraction can be constructed.
- *Filters* add selection conditions over the context defined by a workspace (e.g., an interesting keyword or a time interval specified through a calendar component), thus filtering the mashup result set.
- *Viewers* support the visualization of result sets, which can be extracted by data services from internal sources or from generic external resources. Since data visualization usually involves some form of aggregation, most viewers embed a transformation logic. Therefore viewers can be simple tables, any kind of graph (as for example, those provided by Google Charts), and any visualization/aggregation service that is useful for the specific domain (e.g., a tag cloud or a map).
- *Generic components* can be also integrated to make the analysis process more effective. They can provide a variety of functionalities, such as video or image retrieval through the most common APIs, or further data sources (e.g., RSS Feeds), which can complement the information extracted from the corporate data sources.

[3] When the integration of multiple sources is required, this module could embed the needed integration logics or alternatively make use of a dedicated integration layer.

The previous component classification can be adopted in several mashup contexts. For example JackBe Presto also exploits a similar classification of *mashables* [7]. What is new in our approach is that this classification of services allows us to codify default data flows that the platform exploits to automatically include ad-hoc listeners in the composition model. For example, viewers always "consume" data, as they elaborate and visualize data extracted by other components, for example data services. The addition of a viewer into the composition therefore requires at least the inclusion of a listener to subscribe the viewer to a *dataReady* event exposed by the component that produces data.

Figure 4 highlights the synchronization managed through the main default listeners. Any time a viewer is added into the workspace, the Composition Handler publishes an event that triggers a Data Service Client operation that sends a pertinent portion of the updated state to the Data Service Server. Based on the exchanged state information, the Data Service Server queries the workspace data set and sends the result set back to the Data Service Client. Based on another default listener, the Data Service Client raises an event so that subscribed viewers know about the new result set and, thus, refresh their state.

The "knowledge" about possible default mappings is coded inside the Composition Handler and can be easily configured, with the advantage that DashMash can be easily adapted to domains with possibly different services and service classifications. DashMash can also work without classification, allowing users to couple components by means of custom listeners definition.

Custom Listeners and Quality-based Recommendations. One peculiarity of DashMash with respect to other mashup platforms is the emphasis on quality aspects to support the users in the mashup construction. In particular, we have identified two phases where quality issues must be taken into account: (i) the registration into the platform of new services and (ii) the generation of recommendations during the mashup composition.

Registration of new services. As highlighted in Figure 2, the set of components C available in the DashMash platform is composed of services that can be created ad-hoc or can be public. Independently on the component nature, when a large amount of functionally equivalent data sources and services are available, quality can be one relevant driver for the selection of the most dependable components. We therefore adopt a quality evaluation approach for estimating the quality of components, which focuses on both the quality properties of mashup components and the reliability (i.e., reputation) of the Web information sources accessed to retrieve the mashup data:

– The *component quality* refers to the quality model proposed in [4], which specializes traditional quality dimensions (such as functional efficiency, reliability and usability of the APIs, and data and presentation quality criteria) to the peculiar context of mashup composition.

– The *reputation of data sources* refers to the model presented in [2], which specifically addresses the trustworthiness of Web 2.0 contents, and in particular those deriving from blogs and community sites.

When a new component is added to the DashMash component registry, the quality data needed to determine the quality indices prescribed by the two adopted quality models are documented as annotations to the component descriptors. Each component $c_i \in C$ is therefore associated with a *component descriptor* and a *quality data vector*. The component descriptor lists all the operations and the events, plus the technical details needed to compute quality indices. The quality data vector, $QD_i = [qd_{i1}, qd_{i2}, \cdots, qd_{in}]$, contains the list of measures for the component quality CQ_i and source quality SQ_i as aggregated quality indexes for the i-th component and its associated data source, respectively.

Component selection during mashup composition. When a component is added into a mashup under construction, recommendations about further components to be added to the composition are generated. In particular, the Composition Handler dispatches composition events to the *Recommendation Manager* that, starting from the quality indexes stored in the component registry, suggests actions to achieve or improve the quality of the mashup. To do so, we adopt the approach described in [16], in which components are ranked on the basis of their *mashability* [16]. Mashability is defined as the capability of a component (i) to be combined with previously selected components and (ii) to maximize the quality of the overall mashup. Thus, it can be seen as a combination of two dimensions: component compatibility and aggregated quality.

Component compatibility estimates whether a component can be coupled with those already included in a composition and distinguishes between *syntactic compatibility*, checking the compatibility between input/output parameters exposed by the components, and *semantic compatibility*, checking whether input/output parameters and operations belong to the same or similar semantic categories, assuming that syntactic compatibility is satisfied. The compatibility index $comp_i$ provides a preliminary measure of the compatibility of a service c_i to be added to a given mashup: a value equal to zero indicates the incompatibility from a syntactic point of view while a positive value provides a measure of semantic compatibility. This index is stored in form of a matrix, where events and operations of the available components are related to each other and their compatibility is scored. The matrix is updated every time a new component is registered to the component registry. During mashup composition, the Recommendation Manager accesses this matrix to determine the list of components that can be suggested to the user as they can be correctly coupled with the components already in the composition. For example, in Figure 1c, only the compatible components are shown in the drop-down list where the user can select a new component to add.

While compatibility ensures the construction of "correct" mashups, *aggregated quality* drives the user in the choice of components that can lead to quality mashups. It estimates the quality of the mashup under construction as a composition of the quality of individual components. The mashup quality QM_k cannot

be however quantified as a simple sum of the quality of individual components, CQ_i, but it is necessary to weigh quality indices by taking into account the role and the importance of each component [5]. Therefore, starting from the composition model, we take into account the in- and out-degree of each component in the composition graph, and use them to determine the component impact on the overall composition, so weighing the aggregated quality. As shown in Figure 1c, aggregated quality values are visualized for each compatible components suggested in the drop-down list.

4 User-Based Validation

In order to validate the composition paradigm of DashMash with respect to end user development requirements, we conducted a user based study. We observed domain experts and naive users completing a set of tasks through our platform. Our goal was to assess how easily the users would be able to develop a composite application. The experiment specifically focused on the efficiency and intuitiveness of the composition paradigm, trying to measure such factors in terms of user performance, ease of use and user satisfaction. In particular, we expected all users to be able to complete the experimental tasks. However, we expected a greater efficiency (e.g., reduced completion task times) and a more positive attitude (in terms of perceived usefulness, acceptability and confidence with the tool) by expert users. Their domain knowledge and background could indeed facilitate the comprehension of the experimental tasks, and improve the perception of the control over the composition method, and thus, their general satisfaction.

The study involved 35 participants. Six of them were real end users, i.e., analysts and decision makers that are supposed to actually use DashMash for their analyses in the sentiment analysis domain. In order to prove to which extent the tool was intuitive even for naive users, we also involved undergrad students of the Computer Engineering Programme at Politecnico di Milano, with a moderate knowledge about Web technologies, but never exposed neither to our tool nor to the sentiment analysis domain.

For novice users, the completion of the experimental tasks was preceded by a 5-minute explanation about the domain and about the basic composition actions supported by the tool. Expert users were instead introduced to the set of available components and the basic composition mechanisms. All users were first asked to fill in a pre-test questionnaire, to gather data on their knowledge about services and mashups. All users were then asked to perform two composition tasks. The two tasks were comparable in terms of number of components to be integrated and composition steps. Task 2, however, required a less trivial definition of filters, to sift the involved data sources, and a more articulated definition of bindings. Also, while the formulation of task 1 was more procedural, i.e., it explicitly illustrated the required steps, task 2 just described the final results to be achieved, without revealing any details about the procedure required.

After the completion of the two experimental tasks, users were then asked to fill in a satisfaction questionnaire.

4.1 Results

All the participants were able to complete both tasks without particular difficulties. No differences in task completion time were found between experts and novices. In particular, domain expertise was not discriminant for task 1 ($p = .085$) and for task 2 ($p = .165$). Similarly, technology expertise was not discriminant for task 1 ($p = .161$) and for task 2 ($p = .156$). The lack of significant differences between the two groups does not necessarily mean that expert users performed bad. However, it indicates that the tool enables even inexperienced users to complete a task in a limited time. The average time to complete task 1 was about 2.5 minutes, while for task 2 it was less then 2 minutes.

The difference in completion times for the two tasks can be also used as a measure of learning [9]. This difference is about half a minute ($t = 28.2, p = .017$), i.e., a reduction of about 15%. This result highlights the learnability of the tool: although the second task was more critical compared to the first one, subjects were able to accomplish it in a shorter time.

The ease of use was also confirmed by the data collected through four questions in the post-questionnaire, asking users to judge whether they found it easy to identify and include services in the composition, to define service bindings between services, and to monitor and modify the status of the mashups. We also asked users to score the general ease of use of the tool. Users could modulate their evaluation on a 7-point scale. The reliability of the ease of use questions is satisfying ($\alpha = .75$). The correlation between the four detailed questions and the global score is also satisfying ($\rho = .58, p < .001$). This highlights the high external reliability of the measures. On average, users gave the ease of use a mark of 1.77 (the scale was from 1 very positive to 7 very negative). The distribution ranged from 1 to 4 ($mean = 1.77, meanS.E. = .12$). We did not found differences between novice and expert users. This was especially true for perceived usefulness ($p = .51$).

The user satisfaction with the composition paradigm was assessed using two complementary techniques. A semantic-differential scale required users to judge the method on 12 items. Users could modulate their evaluation on a 7-point scale (1 very positive - 7 very negative). Moreover, a question asked users to globally score the method on a 10-point scale (1 very positive - 10 very negative). The reliability of the satisfaction scale is satisfying ($\alpha = 0.76$). Therefore, a user-satisfaction index was computed as the mean value of the score across all the 12 items. The average satisfaction value is very good ($min = 1.3, max = 3.5, mean = 2.2, meanS.E. = .09$). The correlation between the average satisfaction value and the global satisfaction score is satisfying ($\rho = .41, p < .015$). On average, users gave the composition method a mark of 2.9, with a distribution ranging from 2 to 4. We did not find differences between experts and novices. Despite our initial assumption, we found that the ease of use of the tool is perceived in the same way by novice and expert users, although the latter have greater domain knowledge. The moderate correlation between the satisfaction index and the ease of use index ($\rho = .55, p = .011$) also reveals that who

perceived the method as easy also tended to evaluate it as more satisfying. This confirms that ease of use is perceived.

The last two questions asked users to judge their performance as mashup developers and to indicate the percentage of requirements they believed to have satisfied with their composition. This metric can be considered as a proxy of confidence [9]. On average, users indicated to be able to cover the 91% of requirements specified by the two experimental tasks ($min = 60\%, max = 100\%, meanS.E. = 1.7\%$). They also felt very satisfied about their performance as composers ($mean = 1.8, meanS.E. = .13$; 1 - very positive, 4 - very negative).

5 Related Works

Service composition has been traditionally covered by powerful standards and technologies (such as BPEL, WSCDL, etc.), which however can be mastered by IT experts only [17]. According to the End User Development vision, enabling a larger class of users to create their own applications requires the availability of intuitive abstractions and easy development tools, and a high level of assistance [3,12]. Some projects (e.g., Dynvoker [19], SOA4All [11]) have focused on easying the creation of effective presentations on top of services, to provide a direct channel between the user and the service. However, very often such approaches do not allow the composition of multiple services into an integrated application.

Mashups are emerging as an alternative solution to service composition that can help realize the dream of a *programmable Web* [13], approachable even by non-programmer users. There is a considerable body of research on mashup tools, the so-called *mashup makers*, which provide graphical user interfaces for combining mashup services, without requiring users to write code. Among the most prominent platforms, Yahoo!Pipes (`http://pipes.yahoo.com`) focuses on data integration via RSS or Atom feeds, and offers a data-flow composition language. JackBe Presto (`http://jackbe.com`) also adopts a pipes-like approach for data mashups, and allows a portal-like aggregation of UI widgets (mashlets). IBM DAMIA [18] offers support to quickly assemble data feeds from the Internet and a variety of enterprise data sources. Mashart [6] focuses on the integration of heterogeneous components, offering a mashup design paradigm through which users create graph-based models representing the mashup composition. Marmite [20] is specifically tailored for accessing information sources: it offers a more intuitive paradigm to easily program and chain a set of operators for filtering sources. Operators provide an intelligible mechanism for extracting contents from data sources (especially from Web pages); however, the operator composition is still constrained to a parameter coupling logic based on manual type-matching.

With respect to manual programming, the previous platforms certainly alleviate the mashup composition tasks. However, to some extent they still require an understanding of the integration logic (e.g., parameter coupling). In some cases, building a complete Web application equipped with a user interface requires the adoption of additional tools or technologies. Even when they offer an easy composition paradigm, as it happens for example for Intel Mash Maker

(`http://mashmaker.intel.com`), they do not guide at all the composition process, as we do in DashMash through quality-based recommendations. A study about users' expectations and usability problems of a composition environment for the the ServFace tool [14] shows that there is evidence of a fundamental issue concerning conceptual understanding of service composition (i.e., end users do not think about connecting services). Also, in given contexts, the openness towards any kind of services and the full set of functions these tools are able to provide are not actually required. *Sandbox* environments, like the one presented in this paper, where ready-to-use services are composed according to a "constrained" integration logic, can help achieve a lightweight, easy to use composition [15], which is especially effective to serve well-defined situational purpose [10].

6 Conclusions

This paper has presented a mashup platform that leverages a composition paradigm aiding end-user development. Our platform is particularly effective for enterprise mashups, where the application domain and the component characterization are easier to identify and represent in form of rules for the automatic definition of service coupling. However, the approach remains valid for the composition of *generic* mashups. Of course, the lower the availability of domain descriptions, the greater the need for users to explicitly define service couplings. Nevertheless, as confirmed by the user-based study, the coupling definition mechanism is still intuitive and further facilitated by quality-based recommendations.

As future work, we aim at exploring different composition solutions, to address, for example, the cooperative definition of mashups (a feature that can greatly enhance team-based cooperation), also enabling mashup creation and fruition on mobile device, as well as an extension of the recommendations mechanisms based on the emergence of composition patterns from the community's mashups [16]. We also aim at easing the creation by users of DashMash components. First preliminary results concerning described services (i.e., WSDL and Linked Data) are encouraging. Our future will be also devoted to improving this feature.

References

1. Barbagallo, D., Cappiello, C., Francalanci, C., Matera, M.: A reputation-based dss: the interest approach. In: Proceedings of ENTER 2010 (2010)
2. Barbagallo, D., Cappiello, C., Francalanci, C., Matera, M.: Reputation-based selection of information sources. In: Proceedings of ICEIS 2010 (2010)
3. Burnett, M.M., Cook, C.R., Rothermel, G.: End-user software engineering. Commun. ACM 47(9), 53–58 (2004)
4. Cappiello, C., Daniel, F., Matera, M.: A quality model for mashup components. In: Gaedke, M., Grossniklaus, M., Díaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 236–250. Springer, Heidelberg (2009)

5. Cappiello, C., Daniel, F., Matera, M., Pautasso, C.: Information quality in mashups. IEEE Internet Computing 14(4), 14–22 (2010)
6. Daniel, F., Casati, F., Benatallah, B., Shan, M.-C.: Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In: Laender, A.H.F., Castano, S., Dayal, U., Casati, F., de Oliveira, J.P.M. (eds.) ER 2009. LNCS, vol. 5829, pp. 428–443. Springer, Heidelberg (2009)
7. Derechin, L., Perry, R.: Presto Enterprise Mashup Platform. Technical report, JackBe (2010)
8. Fischer, G.: End-User Development and Meta-design: Foundations for Cultures of Participation. In: Pipek, V., Rosson, M.B., de Ruyter, B., Wulf, V. (eds.) IS-EUD 2009. LNCS, vol. 5435, pp. 3–14. Springer, Heidelberg (2009)
9. Hornbk, K.: Current practice in measuring usability: Challenges to usability studies and research. Int. Journal of Human-Computer Studies 64(2), 79–102 (2006)
10. Jhingran, A.: Enterprise information mashups: Integrating information, simply. In: VLDB, pp. 3–4 (2006)
11. Krummenacher, R., Norton, B., Simperl, E.P.B., Pedrinaci, C.: Soa4all: Enabling web-scale service economies. In: Proceedings of ICSC 2009, pp. 535–542. IEEE Computer Society, Los Alamitos (2009)
12. Liu, X., Huang, G., Mei, H.: Towards end user service composition. In: COMPSAC, vol. (1), pp. 676–678. IEEE Computer Society, Los Alamitos (2007)
13. Maximilien, E.M., Wilkinson, H., Desai, N., Tai, S.: A domain-specific language for web aPIs and services mashups. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 13–26. Springer, Heidelberg (2007)
14. Namoun, A., Nestler, T., De Angeli, A.: Conceptual and usability issues in the composable web of software services. In: Daniel, F., Facca, F.M. (eds.) ICWE 2010. LNCS, vol. 6385, pp. 396–407. Springer, Heidelberg (2010)
15. Ogrinz, M.: Mashup Patterns: Designs and Examples for the Modern Enteprise. AddisonWesley, Reading (2009)
16. Picozzi, M., Rodolfi, M., Cappiello, C., Matera, M.: Quality-based recommendations for mashup composition. In: Daniel, F., Facca, F.M. (eds.) ICWE 2010. LNCS, vol. 6385, pp. 360–371. Springer, Heidelberg (2010)
17. Ro, A., Xia, L.S.-Y., Paik, H.-Y., Chon, C.H.: Bill organiser portal: A case study on end-user composition. In: Hartmann, S., Zhou, X., Kirchberg, M. (eds.) WISE 2008. LNCS, vol. 5176, pp. 152–161. Springer, Heidelberg (2008)
18. Simmen, D.E., Altinel, M., Markl, V., Padmanabhan, S., Singh, A.: Damia: data mashups for intranet applications. In: Wang, J.T.-L. (ed.) Proceedings of SIGMOD 2008, pp. 1171–1182. ACM, New York (2008)
19. Spillner, J., Feldmann, M., Braun, I., Springer, T., Schill, A.: Ad-hoc usage of web services with dynvoker. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 208–219. Springer, Heidelberg (2008)
20. Wong, J., Hong, J.I.: Making mashups with marmite: towards end-user programming for the web. In: Proceedings of CHI 2007, pp. 1435–1444 (2007)
21. Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F., Matera, M.: A framework for rapid integration of presentation components. In: Proceedings of WWW 2007, pp. 923–932 (2007)

Bibliography

[1] Eclipse rich client platform, http://wiki.eclipse.org/index.php/.

[2] http://download.oracle.com/.

[3] http://en.wikipedia.org/wiki/xaml.

[4] http://en.wikipedia.org/wiki/xul.

[5] http://giove.isti.cnr.it/tools/mariae/.

[6] http://www.etltool.com/.

[7] http://www.highcharts.com.

[8] Microsoft .net framework, http://www.microsoft.com/net/.

[9] Smart client-composite ui application block, http://msdn.microsoft. com/library/.

[10] S. Agnoletto and A. Barrese. Progettazione dei mashup orientata alla qualità: modelli, metodi e strumenti di sviluppo. Master's thesis, Politecnico di Milano, 2009.

[11] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. Technical report, BEA, IBM, Microsoft, SAP, Siebel, 2003.

[12] A. D. Angeli, A. Namoun, and T. Nestler. End user requirements for the composable web. *ComposableWeb 2010*, 2010.

[13] A. D. Angeli, A. Namoun, and T. Nestler. End user requirements for the composable web. In *Proc. of ComposableWeb 2010, in print*, 2010.

[14] S. Balasubramaniam, G. A. Lewis, S. Simanta, and D. B. Smith. Situated software: Concepts, motivation, technology, and the future. *IEEE Software*, pages 50–55, Nov-Dec 2008.

[15] D. Barbagallo, C. Cappiello, C. Francalanci, , and M. Matera. A reputation-based dss: the interest approach. *ENTER*, 2010.

[16] D. Barbagallo, C. Cappiello, C. Francalanci, and M. Matera. *Applied Semantic Technologies: Using Semantics in Intelligent Information Processing*, chapter Semantic sentiment analyses based on the reputation of Web information sources. Taylor and Francis, 2010.

[17] D. Barbagallo, C. Cappiello, C. Francalanci, and M. Matera. A reputation-based dss: the interest approach. In *Proc. of ENTER 2010*, 2010.

[18] D. Barbagallo, C. Cappiello, C. Francalanci, and M. Matera. Reputation-based selection of information sources. In *Proc. of ICEIS 2010*, 2010.

[19] M. Burnett, C. Cook, and G. Rothermel. End-user software engineering. *Communications of the ACM*, 47(9):53–58, 2004.

[20] C. Cappiello, F. Daniel, and M. Matera. A quality model for mashup components. In *ICWE*, pages 236–250, 2009.

[21] C. Cappiello, F. Daniel, M. Matera, and C. Pautasso. Information quality in mashups. *IEEE Internet Computing*, 14(4):14–22, 2010.

[22] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. W3c note, World Wide Web Consortium, March 2001.

[23] F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan. Hosted universal composition: Models, languages and infrastructure in mashart. *LNCS 5829*, pages 428 – 443, 2009.

[24] F. Daniel and M. Matera. *Quando l'utente guida l'innovazione: Il web mashup*, volume 34, pages 29–38. AICA - Associazione Italiana per l'Informatica e il Calcolo Automatico, June 2010.

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

[26] F. Daniel, J. Yu, B. Benatallah, F. Casati, M. Matera, and R. Saint-Paul. Understanding ui integration: A survey of problems, technologies, and opportunities. *IEEE Internet Computing*, 11:59–66, May 2007.

[27] M. deHaaff. Sentiment analysis, hard but worth it! *Customer Think*, Mar 2010.

[28] L. Derechin and R. Perry. Presto enterprise mashup platform. Technical report, JackBe, 2010.

[29] P. E. et al. *The Many Faces of Publish/Subscribe*. Taylor and Francis, 2003.

[30] G. Fischer. End-user development and meta-design: Foundations for cultures of participation. *Journal of Organizational and End User Computing (JOEUC)*, 2009.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[32] H. Haas and A. Brown. Web services glossary. W3C note, W3C, Feb. 2004. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

[33] K. Hornbaek. Current practice in measuring usability: Challenges to usability studies and research. *International Journal of Human-Computer Studies*, 64(2):79–102, 2006.

[34] IBM. Qedwiki, http://services.alphaworks.ibm.com/graduated/qedwiki.html.

[35] Intel. Intel mash maker. Technical report, 2010.

[36] JackBe. Jackbe presto. Technical report, http://www.jackbe.com/, 2010.

[37] A. Jhingram. Enterprise information mashups: Integrating information. *VLDB*, pages 3 – 4, 2006.

[38] K. Khan and C. Locatis. Searching through cyberspace: The effects of link display and link density on information retrieval from hypertext on the world wide web. *Journal of the American Society for Information Science*, 49(2):176–182, 1998.

[39] C. Larman and V. R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, 36(6):47–56, 2003.

[40] M. Lenzerini. Data integration: A theoretical perspective. In L. Popa, editor, *PODS*, pages 233–246. ACM, 2002.

[41] E. Meazzo and M. Mosconi. My personal trainer. complementary-holter metabolico virtuale. Technical report, Bachelor Thesis, Politecnico di Milano, 2010.

[42] R. Mihalcea, C. Banea, and J. Wiebe. Learning multilingual subjective language via cross-lingual projections. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 976–983, Prague, Czech Republic, June 2007.

[43] Z. Obrenovic and D. Gasevic. Mashing up oil and water: Combining heterogeneous service for diverse users. *IEEE Internet Computing*, pages 56–64, Nov/Dec 2009.

[44] M. Ogrinz. Mashup patterns: Designs and examples for the modern enterprise. *Addison-Wesley*, 2009.

[45] OMA. Emml documentation. Technical report, Open Mashup Alliance, http://www.openmashup.org/omadocs/v1.0/index.html, 2010.

[46] B. Pang and L. Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *In Proceedings of the ACL*, pages 271–278, 2004.

[47] B. Pang and L. Lee. Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval*, 2(1-2):1–135, Jan. 2008.

[48] M. Picozzi, M. Rodolfi, C. Cappiello, and M. Matera. Quality-based recommendations for mashup composition. In *Proc. of ComposableWeb 2010, in print*, 2010.

[49] A. Puerta and J. Eisenstein. Ximl: A universal language for user interfaces. *RedWhale Software, Palo Alto, CA USA*.

[50] D. Schawbel. Top 10 reputation tracking tools worth paying, 2008.

[51] ServFace. http://141.76.40.158/servface/.

[52] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh. Damia: data mashups for intranet applications. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 1171–1182. ACM, 2008.

[53] A. G. Sutcliffe, M. Ennis, and S. J. Watkinson. Empirical studies of end-user information searching. *J. Am. Soc. Inf. Sci.*, 51(13):1211–1231, 2000.

[54] E. von Hippel. Democratizing innovation. *MIT Press*, 2005.

[55] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. *CHI*, pages 1435–1444, 2007.

[56] Yahoo. http://pipes.yahoo.com/pipes/.