Using Architecture Integration Patterns to Compose Enterprise Mashups

Yan Liu\textsuperscript{a,b}, Xin Liang\textsuperscript{b}, Lingzhi Xu\textsuperscript{c}, Mark Staples\textsuperscript{b,c}, Liming Zhu\textsuperscript{b,c}

\textsuperscript{a}Pacific Northwest National Laboratory, WA, USA\textsuperscript{b}University of New South Wales, NSW 2052, Australia\textsuperscript{c}NICTA, 13 Garden St, Eveleigh NSW 2015, Australia

Abstract

Enterprise mashups leverage various source of information to compose new situational applications. The architecture of such applications must address integration issues: it needs to deal with heterogeneous local and/or public data sources, and build value-added applications on existing corporate IT systems. In this paper, we leverage enterprise architecture integration patterns to compose reusable mashup components. We present a service oriented architecture that addresses reusability and integration needs for building enterprise mashup applications. Key techniques to customize this architecture are developed for mashups with themed data on location maps. The usage of this architecture is illustrated by a property valuation application derived from a real-world scenario. We demonstrate and discuss how this state-of-the-art architecture design method can be applied to enhance the design and development of emerging enterprise mashups.

Keywords: software architecture, design, patterns, mashup

1. Introduction

The concept of enterprise mashup stems from the mashup development of Web 2.0. Mashup enables individual users to retrieve content from several Internet sources to compose highly personalized and convenient widgets or applications on the Web. Mashup development has now been extended to the enterprise domain whereby a broad range of local or public information\footnote{http://mashapaustralia.org/} can be accessed through various mechanisms such as REST APIs or Web feed formats such as RSS, ATOM, and JSON. Leveraging this kind of information brings opportunities to develop new situational applications to improve business productivity or repurpose existing applications with value-added features. An example scenario in this context has been presented in our previous work [1]: a property valuation
service (PVS) company which uses an internal service to retrieve historical data about the valuation results of individual areas. Such data informs the house price valuation process when combined with other public information such as traffic and crime statistics. Mashup of both local and public data sources on a location map enriches the information set for a valuer\(^2\) to make an accurate decision on-site.

The development of enterprise mashups brings more architectural challenges than the development of individual "throw-away" end-user widgets. These challenges arise from the need to deal with integration issues such as organizing flows of public and private data, and transforming data into consistent formats for mashup manipulation. In addition, the architecture of enterprise mashups should consist of reusable architectural components so that data sources or new logic for mashups can be easily extended.

The current development of enterprise mashups is ad-hoc and mostly from scratch. A developer usually chooses a preferred development environment such as Yahoo Pipes, Google Mashup Editor, Microsoft Popfly or QedWiki, and manually combines data at the syntactic level. Many similar mashups use the same tools and APIs but do not systematically share common pattern. This implies that less reusability is achieved. As mashups are emerging applications, there has been little research dedicated to proposing a systematic development methodology, and few best practises are well generalized from industrial practices.

Among software architecture principles, architecture patterns (or styles) offer the promise to help architects to identify combinations of architecture and solution building blocks that have been proven to deliver effective solutions in the past, which may provide the basis for effective solutions in the future. Reusing established architecture patterns for mashups and building solutions can provide deep insights to both resolve the architecture design itself, and codify knowledge about mashup technologies. Our previous work in [1] identifies architecture integration patterns that are useful for composing mashup components.

In this paper, we introduce a REST architecture style that further addresses the integration of these architecture patterns in an extensible design. In this RESTful architecture, key components to realize these patterns can be customized and reused to meet specific mashup needs. We develop techniques to realize these patterns for a map with overlays, but the approach is generic to other types of mashup presentation such as markers and annotation on a map. We demonstrate our architectural approach and techniques using a scenario derived from a real-world property valuation service (PVS) used in the Australian lending industry.

In this paper, we first describe the business scenario of the property valuation service in Section 2 and identify the needs for developing enterprise mashups.

\(^2\)A valuer is a professional that assesses the price of a property, usually during an on-site inspection.
Three architecture patterns are discussed in Section 3 to address the mashup design and development. Key components of each pattern are presented in the context of mashup design. The integration of these patterns are guided by a RESTful architecture in section 4. The practical usage of this architecture using a Web-based PVS is demonstrated in Section 5. In the conclusion of this paper, we summarize the related work and lessons learnt.

2. Motivating Business Scenario

In this section, we introduce a business scenario for enterprise mashups — a Property Valuation Service (PVS) derived from our collaboration with an organization in the Australian lending industry [2].

PVS is one of the key services in the mortgage lending process. It provides a lender or a broker with an accurate valuation of the real estate in a loan application. PVS is usually provided by an independent valuation firm. When a valuation firm receives a request from lenders or brokers, the valuation process starts, and requires a number of tasks to be completed. The business goal of the whole valuation process is to reply to customers with an accurate property valuation and with minimal response delays.

In the valuation process, three roles cooperate to accomplish this the valuation task, namely administrator, valuer and manager. The administrator contacts the owner of a property, schedules the inspection time and assigns a valuer to do the valuation on site. Once the valuer accepts the job, the valuer conducts an on-site inspection, completes the valuation form and sends it back to the valuation firm office. A manager receives the valuation form from the valuer and approves it before the valuation result is sent to the customer.

Although the whole business process is well-defined and the responsibility of each role is clear, the actual interactions between different roles are subtle, and achieving the business goal of guaranteed responsiveness is non-trivial. The outcome depends on the availability of the selected valuer and their working experience in the target suburb. Assigning the job to the wrong valuer (either unavailable or unfamiliar with the market in the area) is likely to incur delays caused by switching valuers and back and forth communication through phone calls, emails or faxes. It could also result in less accurate valuations due to a lack of expertise for special purpose valuations such as commercial property valuations. To avoid these risks, the administrator needs to take into account the valuers’ record in terms of the valuation quality and quantity. This information is available in the cooperate system, but administrators need central IT support to access such information.

The lack of flexible and customizable software tools is one major barrier to improve the business process of PVS. One system used in the valuation industry is called Valuation Management System (VMS). Currently VMS only satisfies basic business requirements and does not provide customizable functions to the internal business process of individual valuation firms. For example, there is no function to allow administrators to quickly access valuers’ historical records for particular suburbs.
Enterprise mashups with customizable information overlays provide a feasible solution to address this problem. The mashup architecture is motivated by presenting information to facilitate the tasks in the PVS process. It consists of data or information from three sources: public information on suburb boundaries, corporate records about valuers, and public information on housing prices. The architecture supports a mashup with overlays for administrators to easily view the available valuers for an area. From the valuer’s point of view, the mashup provides a Web-based valuation form so that the valuer can complete the form while simultaneously accessing information related to the property value, such as the traffic statistics.

3. Mashup Architecture Using Integration Patterns

The mashup architecture is driven by the need to combine data or functions from more than one source. The data and the output from any function are of myriad forms and formats. They can be stored at cooperate information systems, available from public domains using RSS (Really Simple Syndication) feeds or Web services calls. They can be accessed by various protocols, such as DB drivers, RSS (Really Simple Syndication), REST (REpresentational State Transfer), HTTP (HyperText Transfer Protocol) or SOAP (Simple Object Access Protocol).

The architectural nature of mashup includes three core aspects: the dataflow procedure, the composition of heterogenous data and interfaces, and different views on the same sets of data. Such demands motivate reusing architecture integration patterns to build mashup design solutions [3].

We develop an architectural solution for composing mashups guided by integration patterns. Figure 1 depicts the conceptual level architecture. This approach initially exploits three patterns, namely Pipes and Filters pattern, Data Federation pattern and Model-View-Controller pattern. These patterns by nature are suitable for realizing the core aspects of the mashup architecture respectively: composing data flow, transforming heterogenous data formats, and enabling multiple views on data. The composition of these patterns adopts the RESTful architecture style addressed in the next section.

To apply these generic patterns to address specific mashup needs, our architecture approach follows three steps:

1. identify key components for mashup development
2. map the key mashup components to individual pattern elements and structure
3. carefully design the intersecting points of patterns to reduce the risk of cross-cutting concerns.

We demonstrate our approach for each integration pattern as follows.

3.1. Pipes and Filters

The dataflow in the conceptual architecture (see Figure 1) has a procedural nature. Many mashups leverage the Pipes and Filters integration pattern to
create the logic for data combination. The implementation of Pipes and Filters pattern consists of a series of interconnected components. Each component performs a specific function such as filtering unnecessary data, joining records, or forking streams. The connection between components denotes that the output from one component is the input to the other component.

In the context of our mashup architecture, the data retrieval service consists of pipes of composite components that produce data as inputs to the mashup generation part. The data retrieval service devises a number of Filter components that are attached to each data source. For example, domain.com.au provides comprehensive housing prices for cities and suburbs in Australia, however the mashup may only consume data of the City of Sydney covering several suburbs. Therefore a Filter connecting to the data source in URL extracts data just for those suburbs and aggregate them for the area of the City of Sydney.

A Filter component can be further decomposed into several Pipes and Filters as shown in Figure 2. It contains a data loader to load in the data, connects to a couple of components to trim unnecessary data of various kinds, and finally converts the data to the required format by the mashup component. For example, Australian Bureau of Statistics (ABS)\(^3\) provides comprehensive information of the geographic data of Australian territories, which can be used to produce the area boundary on Google Maps. At the moment only a zip file is available for downloading, and the format is in the GIS format. However Google Maps only renders map data in the KML (Keyhole Markup Language) format.

\(^3\)http://www.abs.gov.au/
which is used to display geographic data in an Earth browser such as Google Earth, Google Maps, and Google Maps for mobile. Hence the format converter is attached at the end of the filter component.

![Figure 2: Pipes and Filters Pattern](image)

Similar to the Filter component, the Data Gen component also consists of several Pipes and Filters shown in Figure 3. It transforms the data into a different scale or measure for the presentation purpose. Statistical data such as the median property prices of suburbs are normally presented by different color or density. The Mapping Function in Figure 2, for example, maps the median property price to color units in order to overlay the price information on the map. This Mapping Function is different from the Format Converter, which concerns with the structure and format of the data. The component of Scaling Function deals with perceptual scaling that can be necessary along the pipes. Since there can be different methods of presenting map overlays, such as symbols marked on location maps in 1D (height), 2D (area) and 3D (volume), the Scaling Function converts the symbol representation between dimensions.

![Figure 3: Mashup Data Generation](image)

The advantage of using the Pipes and Filters pattern is the simplicity in modeling data flows. Each pipe or filter in the flow is explicitly defined and their integration is driven by the data flow. Data sources are filtered, processed and combined to produce new meshed-up information. In addition, it is easier to maintain the mashups built using this pattern since the mashups execute exactly as they are designed following the flow paths at runtime.

### 3.2. Data Federation

The Data Federation pattern aims to efficiently federate both structured and unstructured data from multiple disparate sources. It helps to mitigate the burden of aggregating, correlating and correcting relevant yet heterogenous data.
This pattern supports data operations against a transient or virtual view. At
design time, views of the data sources are declared and the relationships between
such views are defined as operations including joins, unions, projections, selec-
tions and aggregations. Programs (such as stored procedures) or software tools
(such as CodeGlide Fusion\(^4\) and Denodo\(^5\)) can be applied to define operations
and produce views.

The relation between the Data Federation pattern and the Pipes and Filters
pattern is illustrated in Figure 1. The Data Federation pattern can be realized
by several Pipes and Filters, each providing the *Mashup Component* with an
interface to access the data. The data remains under the control of the source
systems and are retrieved on demand for federated access.

Data federation concepts and techniques were extensively researched in the
last two decades. Now this pattern has found yet another use in mashup archi-

tectures. In the context of mashup, one variation introduced to the Data
Federation pattern is data redundancy. The original Data Federation pattern
creates virtual views without introducing data redundancy. For mashups, how-
ever, data are often reused since the outputs from the Pipes and Filters produce
the data in a more suitable structure and format to be processed by mashup
components. Therefore, these data can be stored and reused later without going
through the data processing Pipes and Filters every time they are needed. Thus
introducing some level of redundancy can both simplify the design and improve
its responsiveness.

3.3. Model-View-Controller Pattern

Mashups follow the Model-View-Controller (MVC) pattern by nature as they
render data to present views according to user inputs. In the MVC pattern, the
Model has access to data from a storage or a service provider site, and organizes
the data in a structure to be used by the Controller. The Controller accepts the
browser input, figures out what to query the model for, and decides which view
to use and what data to send to the View. The View accepts input from the
controller and generates outputs in HTML, XML or JavaScripts to the user's
browser.

There are two ways to construct a mashup architecture following MVC:
server-side mashup \([4]\) and client-side mashup \([5]\). In the deployment of a client-
side mashup, the logic of composing map overlays is performed by Javascript
at the client browser. This architecture can use Google Mashup Editor, XML
databases and Javascript libraries such as jQuery. This simple deployment has
drawbacks due to the browser's limits to handle complex logic and large amounts
of data. Alternatively, a server-side mashup allows the integration of data con-
tents at the server side. The MVC pattern is constructed as the server-side
mashup in our architecture.

\(^4\)http://fusion.codeglide.com
\(^5\)http://www.denodo.com
To illustrate the MVC structure, we adopt the sever-side mashup diagram from [4] and annotate the diagram with the key artifacts involved in the MVC pattern. As shown in Figure 4, a View of the mashup is presented to the end-users through a Web page that contains GUIs. The user interacts with the GUI components and triggers Javascript invocations (through Ajax) to the server side such as loading a page with the mashup embedded. The logic of producing the mashup Web page is guided by the Controller. The Controller at server-side system uses Servlets to dispatch the requests to an appropriate service class that realize the functions of Mashup Generation service in the architecture presented in Figure 1. Mashup generation and data processing from other sites or services (as Models) are done by the service class as part of the Controller. The response is then sent back to the Servlet, which in turn sends the response back to the client browser. The browser’s JavaScript functions to update the Web page and represent the information obtained from the response as the View to the end user.

The server-side mashup is more scalable when making use of the Data Federation pattern and the Pipes and Filters pattern. As discussed above, the Data Federation pattern for mashups introducing data redundancy whereby data read-once/used-many-times can be stored at the sever side without accessing the other sites or services each time the data demanded.

4. Composing Patterns in RESTful Architecture

The conceptual architecture in Figure 1 consists of two core parts: the Data Retrieval Service (DRS) and the Mashup Generation Service (MGS). The function of DRS is to prepare the data from multiple sources in a format ready to be processed by the MGS. The MGS takes the inputs of DRS and generates data overlays that can be rendered and presented in a client’s browser. Each part of the service incorporates the integration patterns to structure its key
components. The integration of the two services requires the interface between DRS and MGS be designed to best suit the data source. If the thematic data is produced through existing enterprise legacy systems, the DRS needs to implement the interface exploiting traditional enterprise integration technologies such as Java Message Service (JMS) or Simple Object Access Protocol (SOAP) Web services. The thematic data can be delivered to the MGS through message queues or as an attachment to Web service calls. The technical details are out of the scope this paper. The best practises to utilize these technologies have already been well studied and documented such as enterprise integration patterns in [6] and [7].

In this section, we focus on consuming the thematic data from services in the Internet that provide access to data sources. These services provide open APIs that provide immediate and easy programming access to shared data in the public domain such as the Public Data Sets on Amazon Web Services[^5]. These open APIs have increasingly been implemented in a RESTful style following the software architecture design paradigm called Representational State Transfer (REST) [8].

The most important element of REST is a resource, which is the source of specific information. Resources can be made available in any format, such as a HTML page, an image, or an annotation file for DNA sequence. A resource is uniquely identified on the Web by a URI. To access a resource, REST uses the HTTP protocol to send URL requests and receive responses through HTTP. Common HTTP methods of GET, PUT, POST and DELETE provide operations on resources by defining respectively list/retrieve, replace/update, create and delete operations for resources.

The simple architecture style of REST brings a number of benefits to the design of highly flexible, low-overhead REST APIs to access collections of themed data:

1. Support for unified development of Web-browser based interfaces to data source. From the development point of view, REST APIs can be invoked using different programming languages such as C++, Python, Java, C# and so on. Thus, new features that further manipulate the data obtained through REST APIs can be developed in a compatible manner to existing applications.

2. Support for encapsulation of data sources. Access to data is well defined in REST APIs, which can be designed to produce different viewpoints of the data in the responses.

3. Lightweight. Compared to SOAP-based APIs in the complex SOAP message format, REST APIs are lightweight in terms of both communication and computing overhead. SOAP is well recognized for its poor performance when the SOAP messages contain large payloads [9].

4. Support for a layered architecture. REST allows the introduction of other computing layers between the Web browser and the server hosting the

[^5]: http://aws.amazon.com/publicdatasets/
data to help with more sophisticated processes such as applying the Data Federation pattern, or Pipeline and Filter pattern to further filter or manipulate the data.

REST is affinitive with Mashup to enable use and reuse of publicly available data. While Mashup produces rich context of thematic data overlay, REST facilitates accessing the thematic data underpinning the mashup. A good example is the Suburban Trends mashup. The sample HTTP request and response using REST APIs are illustrated in Figure 5.

REST API Examples

```

```

Returns application/json:

```
{
  ssc: "31565",
  name: 'WEST END',
  post_code: '4101',
  state: 'QLD',
  lgp_id: '31000',
  centre: { lat: "-27.48214186", lon: "153.0059111" }
  extents: {
    ne: { lat: "-27.4742019", lon: "152.9962674" },
    sw: { lat: "-27.4800819", lon: "153.0155549" }
  },
  surr_subc: { 1: "31265", 2: "31353", ... , 10: "31519" },
}
```

Figure 5: Sample REST API taken from the Suburban Trends

The Suburban Trends mashup combines publicly available online resources from the Australia Bureau of Statistics, Australian Institute of Criminology, the NSW Bureau of Crime Statistics and Research, and Google. One of Suburban Trends’ REST APIs returns comprehensive information associated with the State Suburb Code (SSC) including the suburb’s centre coordinates, the suburb’s extent coordinates (the two coordinates that form the suburb’s bounding box), the SSC for the ten closest suburbs as shown in Figure 5. These coordinates are the primary inputs for the MGS to produce the skeleton of the territory shape (more details will be presented in the next section).

Other REST APIs from Suburban Trends provide thematic data as valuable inputs to enrich the mashup for the Property Valuation Service, such as socio-economic rankings, safety ratings and yearly crime statistics for New South Wales state since 1995. By means of REST APIs, Data Retrieval Services such as Suburban Trends enable Data-as-a-Service by providing easy programming

---

access to applications that may produce diverse, application specific mashup views of the same set of data.

5. Architecture Deployment

5.1. Web-based Property Valuation

The architecture integration patterns for composing mashups are applied to the Property Valuation Service (PVS) process discussed in Section 2. In this section we describe our experience of developing the mashup for a Web-based PVS that utilizes the architecture integration patterns presented in this paper.

5.2. Web-based Valuation Form

The mashup of the PVS has three main GUI panels as shown in Figure 6. The left panel contains the property list that the valuer has inspected or will inspect. The right panel is a tabbed panel that contains the valuation form. This panel also contains a tab to show historical prices trend of the area that the property belongs to. The bottom panel is the map panel, which displays overlays on Google Maps such as housing prices and traffic status. The map highlights the areas that a valuer prefers to work on together with the performance ranking and the booking information. The coloring of the map is based on the median price level of the properties in the area.

Figure 6: Mashup: valuers distribution in suburbs

The Web-based evaluation form is consistent with the paper-based valuation form. The form can be converted to the PDF format and sent to the valuation firm on the fly, which saves paper and the cost of postage. This Web-based valuation form has several other advantages. First, it saves time spent by the
administrator on spotting the best available valuer in the target area. The administrator only needs to input the suburbs of the target areas and the map will be automatically produced with a marker to list available valuers. Valuers are displayed only in their working areas, together with their ranks and bookings information. Second, the valuer can leverage the historical prices of the valuation property easily. Moreover, different types of information can be chosen to highlight on Google Maps. For example, when a valuer clicks on the traffic entry in the valuation form, the map panel displays a map with traffic overlay.

5.3. Mashup Development

The core of this Web-based PVS is the embedded mashup display. The mashup development follows the architecture patterns discussed in Section 3. We use the scenario of the valuer distribution in the area of City of Sydney (including several suburbs) as an example to demonstrate this architecture approach.

The key components to produce the mashup is illustrated in Figure 7. There are four sources of data: the territory geographic data for producing shape boundary on Google Maps, the housing prices and traffic data from the public domain, and the valuer information from the corporate information systems. Thus four pipelines form the Data Federation pattern and produce the inputs to the mashup component. The overall mashup data flow follows the Pipes and Filters pattern. The interactions amongst the client browser, the Web server and the mashup component employ the MVC pattern and use a server-side mashup deployment.

![Figure 7: Using Patterns for Web-based Property Valuation](image-url)
5.4. Preparing Data for Mashup

Preparing data for mashup involves the tasks of accessing the shape files and transforming them to the KML format. KML is an XML format and uses a tag-based structure with nested elements and attributes to describe a shape in coordinates. The KML file does not contain any thematic data. Instead, it provides the base shape that thematic data can be presented as overlays on top of it. The list below shows a partial KML file describing a shape.

```xml
<kml xmlns="http://www.opengis.net/kml/2.2">
<Placemark id="1">
  ...
  <LinearRing>
    <coordinates>
      151.131627264,33.8516306946267,0
      151.131732096,33.8510865171267,0
      151.133179072,33.8499404976268,0
      151.133180896,33.8499313771268,0
      151.133204064,33.8498165106268,0
      151.133276064,33.8494426071268,0
      151.13322,33.8494599971268,0
      151.133136128,33.8494760181268,0
      ...
    </coordinates>
  </LinearRing>
  ...
</Placemark>
</kml>
```

In our case, the KML shape files are essential to produce overlaid information on location maps such as Google Maps. Google Maps APIs provide a way to create a customized overlay if a well-structured KML file is provided on a publically accessible server. The KML shape files are produced by the data retrieval service of the architecture, which consists of a number of data loaders devised to retrieve data from various data sources as shown in Figure 1. The original shape files to produce the KML files can be obtained in two ways as discussed in section 3: (1) downloading and processing the raw shape files from the Australian Bureau of Statistics (ABS), or (2) invoking REST APIs of Internet services that provide suburb coordinates. The former requires components that may use Geographical Information Software (GIS) such as MapWindow to process the shape files as illustrated in Figure 7. The latter can be implemented by toolkits that support client side HTTP standards. The sample code using HTTPClient to invoke the REST API is very straightforward as follows:

```java
```

To demonstrate the techniques of producing data for KML-based map overlays, we present the details of processing original shape files downloaded from ABS in a geographical spatial vector data format. The shape file specifies the coordinates of all suburb boundaries in Australia, including the fields for each suburb: (1) Name of the suburb; (2) State that the suburb belongs to; and (3) Coordinate, the list of longitude/latitude pair that represents the boundary
of the suburb. The coordinate data needs to be aggregated for the relevant suburbs and the remaining data can be filtered out. To do this, the shape file is loaded into a Shape Filter. We use a COTS GIS software MapWindow that supports some editing functions on a shape file. For example, MapWindow has built-in SQL-parsing functionality so that the filtering task can be accomplished by an SQL script similar to \texttt{Select \ldots from \ldots Where State = 'New South Wales'}. Then the data is converted to KML coordinates.

The \textit{GIS2KML Converter} shown at the top of Figure 7 leverages a third party \textit{MapWindow} Plug-in called \textit{Shp2KML} to convert the shape files to the KML files. The effect of the resulting KML file is depicted in Figure 8, which shows the base shape of the suburb ”City of Sydney”. The next step is to input the base KML file to the mashup component to merge overlays with map coordinates.

![Figure 8: Display of City of Sydney in base KML file](image)

One pipeline deals with extracting valuer working area distributions at the bottom of Figure 7. All completed historical valuations are stored in a central corporate database and the existing VMS is connected to access this database. One filter component contains SQL stored procedures to extract the data of interest. Another component \textit{Criteria Filter} is for valuer’s performance ranking. The ranking originates from the PVS manager with regard to individual valuation quality and quantity in the past. Such statistical results can be directly generated from an independent human resource system in a XML format. A XML Parser is used as the filter, which reads the XML file, processes the data and then matches the records of historical valuation results to discover valuers working in a certain area with a satisfactory record.

5.5. \textit{Producing Map Overlays}

Eventually the four pipelines converge in the mashup component to produce a single KML file with map overlays that the Google Maps server can render.
The overlay is visualized by colors. Different colors or color density can represent scales of the data. Hence the component of *Color Mapping Function* maps the data from one measure (price) to another (color). The mapping between the color and the housing prices is customized by the scale parameter of *interval*, shown in Equation 1.

\[
\text{ColorUnit} = \frac{\text{Max}\{\text{housingprice}\} - \text{Min}\{\text{housingprice}\}}{\text{Interval}}
\]  

The next step is attaching the color unit to each element (suburb) in the base KML file. It is straightforward to find the corresponding color level using the color unit, given the starting color is determined. Eventually, the mashup component composes a final KML file with map overlays built-in. The sample code of the final KML file is illustrated below.

```xml
<name>CityofSydney</name>
<placemark id="0">
  <name>Alexandria</name>
  ...
  <polygon>
  ...
  <outerBoundaries>
    <LinearRing>
      <coordinates>151.202134016, -33.9034390011261, 0 151.20181404...
    </LinearRing>
  </outerBoundaries>
</placemark>
</name>
```

The resulting KML files are transmitted to the Google Maps server from an accessible site for rendering. As the files are normally of hundreds of megabytes in our case, optimizing of the transmitting overhead is essential to make the mashup application responsive. Our solution is compressing the KML file using ZIP that helps to produce the size of the compressed file approximately a quarter of the original size. For example, the KML file containing all suburbs in New South Wales is 110.4MB whereas the compressed KMZ file is 30.9MB.
Using the Google Web Toolkit (GWT) based library called GWT-Ext\(^8\), the compressed KMZ file can be rendered following the template below. The effect of the final display is illustrated in Figure 6 and 9.

```javascript
MapPanel.addOverlay("http://www.kmz-serverhost.com/CityOfSydney_HouseValue.kmz");
```

5.6. Producing Different Views

Three pipelines at the top of Figure 7 deliver different views in the mashup following the MVC pattern. As shown in Figure 9, the mashup enables users to select the menu item of interest, and the mashup switches between views. Since the communication is using Ajax asynchronous calls (see Figure 10 for details), the end-user is not blocked for one operation and can continue the interaction with other functions of the Web page before the response returns. From our experiment, the user will experience some delay (no longer than loading a normal Google Maps page) the first time the map and overlays are loaded, and later refreshing of the content or switching the map between views is fairly prompt.

6. Related Work

There are large range of mashup technologies and development tools. One study \([10]\) categorizes them based on their role in the lifecycle of a mashup. In our previous paper \([1]\), we broadly categorize them into two groups, the mashup builders and the mashup enables.

Mashup builders are tools that help developers produce mashup presentations such as IBM QEDWiki and Yahoo! Pipes. These tools can even allow non-developers to compose simple mashups by connecting data sources and operations (such as filtering, sorting, selecting) widgets to create new composite applications. However, these mashup builders do not allow finer arrangement of the internal structure of the mashup application, making systematic reuse, performance tuning and loose coupled design difficult. These composite applications are often brittle and at the mercy of data source changes. Our approach uses the thematic notion and combines this with design patterns to allow a more modular design of the mashup composition beyond data manipulation.

On the other hand, mashup enablers provide functionalities to mashup builders by accessing unstructured data, and making internal and external resources available. For example, Kapow Mashup Server, Openkapow, Tarpipe and Feed43 belong to this category. The primary goal of these mashup enablers is to have pre-built data source adaptors (such as RSS, ATOM, RMDB, Messages and Application-specific APIs) and actuators (such as email/post to other data services’ update functionality). Although they do not impose a black-box internal design of the mashup composition like graphic mashup builders, they also leave the design completely to developers to build from scratch. Our approach fills

\(^8\)http://gwt-ext.com/
this gap by providing a new and reusable internal architecture from a thematic point of view leveraging existing patterns.

Other research has also put further structure on top of existing data. One major direction is the semantic annotation of data sources in the context of Web Services and RESTful Services [11]. Mashups pose new challenges as data is sometimes embedded in XHTML or other presentation structures, rather than being clearly separated. These researchers add semantics to data by annotating them with small domain ontology models. Such semantics could be based on themes. However, semantic web technologies do not provide architectural mechanisms that address how such annotated data should interact with the rest of
the control, adaption and presentation layers. We consider semantic annotation to be a formal way of organizing data into themes, which supports our method.

As mashup is a type of integration, existing enterprise integration and composition patterns have been adapted to support mashups [12]. Such patterns focus on the interfaces and interactions among different sources rather than the internal architecture. Liu at al. [13] described an architecture to build Web service composition using mashup techniques. Cetin et al. [14] proposed a service migration strategy in favor of mashup properties—composition of heterogeneous resources. Zou et al. [15] introduced accountable and reliable enterprise mashups. Other ways of designing mashups exist for specific domains and applications, such as searching [16] and business processes [17].

Our work is complementary to these methods, yet provides a more generic architecture solution of building themed mashups by means of architecture patterns. In our approach, the data retrieval and thematic mashup generation perform as individual services and are seamlessly integrated together using Web-based techniques in the RESTful style. Functions such as semantic filtering and service selection can be added to extend the features of this architecture without affecting the mashup generation. Our conceptual architecture still has space to address advanced issues of mashup for enterprise applications. It remains as future work to extend this architecture to manage security and accountability [15] when data are retrieved from third party services, data sources or applications.

7. Lessons Learnt and Conclusion

In this paper, we propose an architecture solution that demonstrates new usage of architecture integration patterns in mashup architecture design and development. This architecture incorporates the state-of-the-art RESTful style for integrating key components that access and transform data to produce mashup overlays. The design of this architecture focuses on the integration and reusability of mashup components so that the mashups can be composed subject to business processes. We also discuss techniques for implementing Google map-based mashups. A mashup application for the property valuation business service is devised to demonstrate the practical usage of this architecture. Through our experience, we observe that component-based development helps to achieve good encapsulation of the operations required by mashups, and thus enables architecture patterns to structure key components in mashups. We summarize three aspects of lessons learnt through our experience.

Mashup Deployment Architecture. The MVC pattern can be deployed as either a client-side mashup or a server-side mashup (see discussion in Section 3.3). However it is not straightforward to make the design decision about which deployment architecture is suitable for a specific mashup scenario. We initially adopted the client-side mashup architecture, attracted by its simplicity and comprehensive tool support (Google Mashup Editor). However, as the mashup gets complicated with new features added, different operations needs to be performed on different data sets. Eventually the client-side mashup was no longer suitable and we migrated to a server-side mashup.
Using RESTful Services. The affinity of REST and Mashup enables use and reuse of data sources for mashup development. Increasingly mashup applications publish processed thematic data for map overlays using REST APIs. Consuming these services using REST APIs and lightweight HTTP standards is widely supported by different languages and software tools. Mashup applications can benefit from reusable data already processed and so reduce the costs of development.

Using Other Integration Patterns. The three integration patterns presented are not exclusive in our architecture solution. Instead, they form the base structure to enable interactions with other integration patterns augmented for application specific mashup needs. For instance, the Data Federation pattern may contain a sub-scenario of locating a specific data service from multiple providers. This scenario may employ the JEE Service Locator pattern or Web Services UDDI service discovery pattern. In our envision, as the scale of the mashup application expands, augmenting the mashup architecture with more integration patterns is necessary. The design needs to focus on reducing the crosscutting concerns from multiple patterns.

The current architecture is very focused on structuring the mashup flows, and does not consider some advanced features required by enterprise application including security. We envisage that this architecture approach utilizing integration patterns serves as a basis for further extension to address more complicated enterprise mashups requirements.

8. Acknowledgement

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References


