Technical University of Crete, Greece
Department of Electronic and Computer Engineering

Dynamic Web Service Mashups

Apostolos K. Nydriotis

Thesis Committee:
Professor Minos Garofalakis
Professor Stavros Christodoulakis
Assistant Professor Antonios Deligiannakis

Chania, Monday 13th December, 2010

Abstract

In today's world, Web services represent a major technology for deploying interactions between heterogenous applications and for connecting business processes. In order to take advantage of the flexibility such services offer, their dynamic invocation has always been a primary concern. In parallel, the mashup programming paradigm has recently emerged in the context of the Web, giving end-users the opportunity to repurpose, combine and use diverse data sources in a "self-service" way to satisfy their unique needs. This thesis suggests a methodology for mashup platforms to support dynamic invocation of any Web service. Following the mashup concept, we propose on-the-fly creation of a widget to handle the invocation of the particular Web service, without the need for the user to write code.

Contents

List of Figures

LIST OF FIGURES

Chapter 1

Introduction

The extensive usage and development of the Internet and related technologies, have resulted in an interconnected world where we are able to exchange and process information easily, quickly and collaboratively in order to maximize efficiency and performance. In such a world, Web services are becoming the leading technology for automating interactions between heterogeneous and distributed applications and for connecting business processes, since they can provide a distributed computing infrastructure for both intra- and cross-enterprise application integration and collaboration [9].

On the other hand, the mashup paradigm has recently emerged, triggered by the vast amount of Web 2.0 applications created by developers and researchers. As the word "mashup" implies, the mashup concept suggests the combination of existing resources with data and Web APIs to create new Web applications, able to satisfy the continuously growing needs of Web users. The truly revolutionary characteristic of mashups is that they are meant to be composed quickly and with as little effort as possible by end-users who have very limited or even programming expertise.

By augmenting the mashup approach with Web services, we arrive at the Web service Mashups concept, which aims to design and develop a new generation of Web applications, based on the composition of Web services. So far, many different mashup platforms have been created, but they all provide a limited set of capabilities and, we believe, will soon be followed by more powerful and flexible successors.

1.1 Thesis Contribution

In this thesis, we move the flexibility that mashup platforms can achieve one step forward. We present a way for mashup platforms to integrate support for any SOAP Web service *on-the-fly*, without the need to write any code. This way, the platform can be dynamically extended by its user, in response to her needs, without the interference of experts. DYNAMO – currently integrated in Apatar mashup platform – analyzes the Web Service description document to acquire all the necessary information for dynamically creating a client matching the particular service. Then, using this information, it generates and compiles code to implement the service client as a platform widget. Hence, all the user has to do is provide the location of the Web service description document and wait for a plug-in to be created. After that, she is able to use and share the desired Web service in the same fashion as any other widget of the platform. We have also extended Apatar to support some basic geocoding functions which it lacked and which we consider useful (if not essential) in several mashup application domains. Finally, we used DYNAMO and our geocoding functions in a mashup application in order to demonstrate our work.

1.2 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we present the background and related work to our project. We introduce the mashup programming paradigm as well as its consumer-oriented and enterprise usage and we discuss the basics of the Web services technology. We also present other published projects related to our work. Then, Chapter 3, briefly introduces the most popular mashup platforms that are used today and explains the reasons behind our choice of Apatar for integrating our dynamic Web service client generator. In Chapter 4 we discuss the design and implementation of DYNAMO and in Chapter 5 we demonstrate it by using it in a mashup application. Finally in Chapter 6 we discuss the conclusion and the future work for our project.

Chapter 2

Background and Related Work

2.1 Mashups

2.1.1 What are Mashups?

As the Web develops rapidly and the number of its users keeps increasing, the amount of data and services that it provides continuously grows. As a result, there is an emerging need for users to combine multiple services and data sources to best serve their goals. Mashups are applications developed specifically for satisfying this need.

The purpose of mashups is to allow users to control data in a self-service way, without the interference of experts, so that the result would be perfectly suited to the individual needs of each user. Therefore, mashups must be easy to implement and reusable. The first prerequisite comes directly from the need to avoid the experts' interference, while the second is necessary for saving time and effort needed to develop the same mashup multiple times.

One of the dominant approaches for developing mashup applications relies on connecting and wiring widgets. Widgets – or mashlets – are application fragments, that provide intuitive graphical user interfaces (GUIs) and are responsible for limited computational tasks. Each widget has one or more inputs as well as one or more outputs, enabling data to flow into the widget, be processed and then flow out of it and into the next one. This design allows composing a mashup application as of a network of widgets, wired together to produce the desired output.

Examples of a widget and a simple mashup application are shown in Figure 2.1 and Figure 2.2, respectively. Once the mashup is assembled, its creator has the option to either save and reuse it, or share it so that other users may modify and use it. This, of course, contributes even more to the agility of data manipulation which is what the mashup concept is all about.

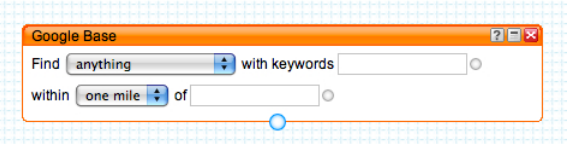

Figure 2.1: Google Base widget in the Yahoo! pipes mashup platform.

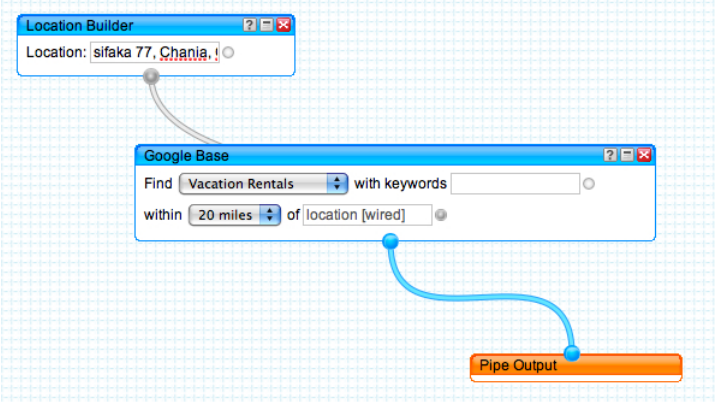

Figure 2.2: A simple mashup application built in the Yahoo! pipes mashup platform.

In order to demonstrate the effectiveness of mashups, we present their use in a holiday trip planning scenario. Assume that someone wants to visit Athens for some days. Normally she would spend a lot of time and effort visiting many web pages to decide which hotel best serves her needs and which airline will she use, to find information about the archaeological sites she will visit, and so on. On the other hand, she could use a mashup application to combine data from a web service providing information and reviews for hotels in the area, an RSS feed containning ticket rates to Athens airport, another web service to get information about historical sites in the city and finally a map service (e.g. Google

Maps) for a visual representation of the results. Using a mashup application drastically simplifies the whole procedure since all the user has to do is as simple as "describing" what data she needs and the mashup will deliver it to her. Moreover, since mashups are reusable, the same application can be used again by the same or another user when they arrive at a similar situation.

2.1.2 Enterprise Mashups

Although many of the early mashup applications were consumer-focused, recently the enterprise has started to both accept and be interested in the mashup paradigm. As mashup creation techniques mature, more and more organizations begin to mash their resources (services and data) together with other existing resources, internal or external to their organization, to provide new interesting ways of representing data.

As explained in [7], there is a rising trend for simple applications to be constructed on-the-fly to solve some evanescent day-to-day problems. Such *situational* applications often need data, such as spreadsheets, presentations and e-mails, that usually cannot be handled by Enterprise Information Integration (EII) architectures. Moreover, situational applications usually target only a small community of users and a specialized business need while, on the contrary, typical enterprise applications are developed by corporate IT staff for a large number of generic users and a general purpose. Therefore, situational applications represent the long-tail of enterprise application development, as shown in Figure 2.3.

Mashup applications are situational in nature, support self-service development to meet the user's unique needs, and have a short implementation cycle. Furthermore, since mashups are meant to be flexible and personalized, they can rather simply access and use data residing in local storage, such as files stored on a personal hard drive. On that ground, enterprise mashup technologies can help solve business and IT challenges and also can provide new insights through mashing functionality from different sources. Figure 2.4 illustrates how a mashup platform can provide an effective data-integration paradigm across several enterprise systems.

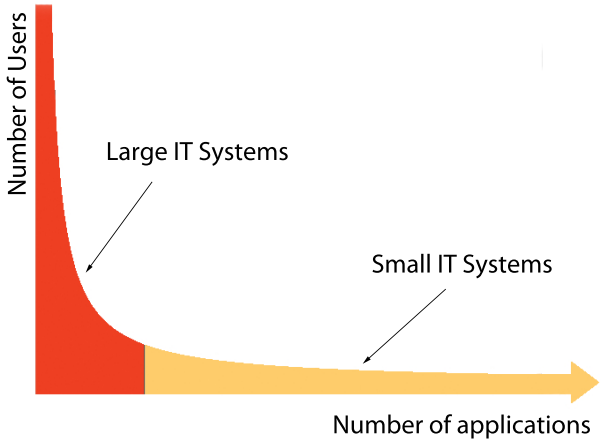

Figure 2.3: Enterprise application development.

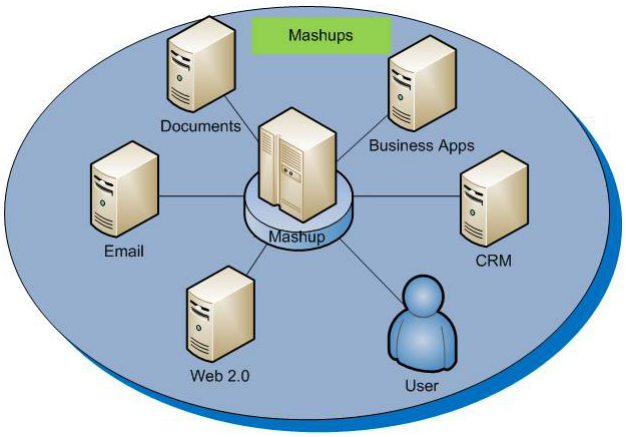

Figure 2.4: Enterprise mashups.

2.2 Web Service Basics

The World Wide Web Consortium (W3C), describes a web service as a software system that supports interoperable machine-to-machine interaction over a network [16]. The major stimulus for creating such services is to enable the automation of business-to-business interaction over the Web. Therefore, the problem that has to be addressed is the automated execution of distributed applications which have been developed independently to each other and possibly by different organisations.

This kind of process in the Web environment is not trivial and faces numerous challenges, most of which are related to the very nature of the Web. First of

all, each organization uses its own application semantics that are not necessarily known to other organizations. Secondly, communication protocols may differ from application to application while cooperation between organizations is probably spontaneous and may not be very frequent. Finally, firewalls do not allow tight coupling of applications. To overcome these difficulties and enable application integration in the Web, a number of standards and protocols such as SOAP and REST communication protocols have been developed, along with the WSDL language and UDDI registries. Each of them will be discussed in the following sections.

In the process of engaging a Web service, we can identify two participants, a *requester entity* and a *provider entity*. The requester entity may be any entity that wishes to make use of a Web service; for example, a person or an organization. The provider entity may be any entity that provides a Web service. Additionally, we identify the entity's *agent*, which is the piece of software that actually implements the Web service. As illustrated in Figure 2.5, the Web service engaging process is generally composed of four steps [14]. In the first step, the requester and the provider entities have to become known to each other, or at least one of the entities must become known to the other. In the next step, the two entities have to agree on the service description and semantics that the agents will use to interact with each other. The third step consists of the realization of the description and the semantics by the agents. In the final step, the agents interact by exchanging messages. The previously mentioned standards play a key role in successfully completing this process.

2.2.1 UDDI

As noted earlier, the first step in the process of engaging a Web service, involves the participating entities becoming known to each other, a step also known as *Web service discovery*. Of course, this is a very important part of the procedure since no Web service could be useful if no potential user is able to find sufficient information, to permit its execution. Web service discovery is meant to be achieved by UDDI.

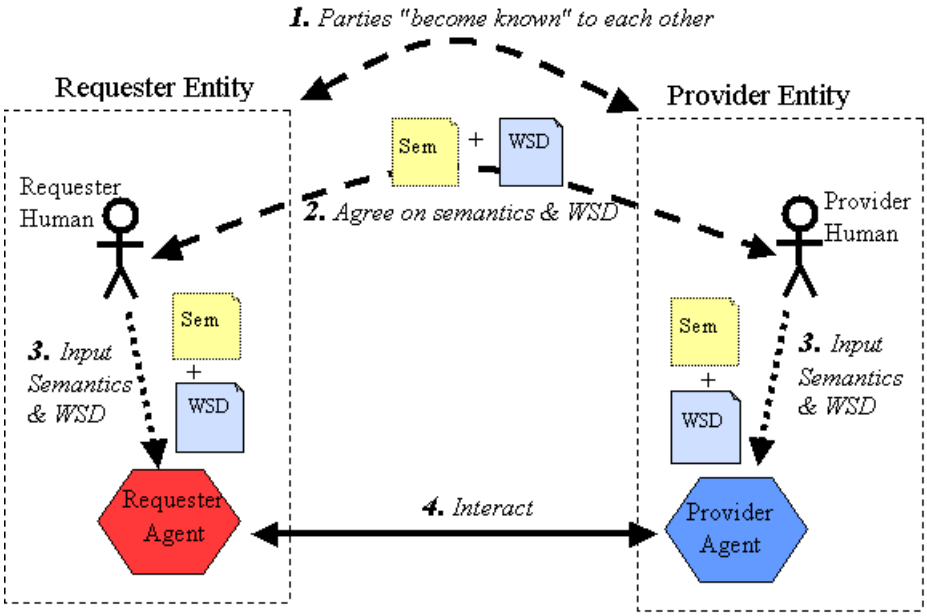

Figure 2.5: General process of Engaging a Web Service.

UDDI stands for Universal Description Discovery and Integration, and its purpose is to define a set of services for supporting discovery and description of businesses, organizations, and other Web service providers, the Web services they make available and the technical interfaces which may be used to access such services. UDDI is based on industry standards, such as HTTP, XML, XML Schema and SOAP to provide interoperable, foundational infrastructure for a Web service-based software environment for publicly available services, as well as services exposed only internally within an organization [10]. It is structured by nodes, which are the basic architectural units, registries, which comprised one or more nodes, and affiliations of registries which are formed by multiple registries.

A UDDI registry does not provide just technical specifications of Web services but also focuses on representing more extensive information about them such as data and metadata. It provides a standard mechanism to classify, catalogue and manage Web services so that queries can be issued to it – at design or run time – whose answers will aid in the discovery of the Web service or other relevant information on the service.

The representation of this information is structured in hierarchical structure consisting of three levels. The top level of this structure is called *businessEn-*

tity and it is used to represent businesses and providers within the UDDI. The information that is stored on this level describes the business or provider along with the service it provides. Such information would be names and descriptions, contact and classification information, and so on. However, this level does not provide any technical information about the service.

Next in the hierarchy is the *businessService* level which models logical groupings of Web services. The information stored here describes the purpose of the individual services found in each grouping but, similar to *businessEntity*, *businessService* does not provide technical information about Web services.

Finally, the last level is called *bindingTemplate* and it represents an individual Web service. This level provides the technical information needed by applications to bind to and interact with the described service. This level must store either the access point for the service or an indirection mechanism pointing to the access point.

The information model used by UDDI specifications is specially designed to be as flexible as possible. Therefore, the information that can be accommodated is not bound to any specific model or technology and, as a result, a UDDI registry can store information for a diverse set of services. For example, the information stored at the *bindingTemplate* level could be describing a Web service based on WSDL, XML, or other technologies.

2.2.2 WSDL

The second step that has to be followed when engaging a Web service is the involved parties' agreement on the semantics and the Web service description. Therefore, every Web service publishes an interface that is described using a machine processable format and, more specifically, WSDL. WSDL stands for Web Service Description Language, and is an XML-based language that describes Web services and how to access them. It defines an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages.

Every WSDL document is written in XML and consists of six basic elements [15].

- The `<types>` element defines the data types used by the Web service. WSDL is designed for maximum interoperability and platform neutrality and therefore is not tied to any specific data typing system. However, it uses as default the W3C XML Schema specification since this is currently the most widely used specification for data typing.

- The `<message>` element defines the messages used for interacting with the Web service. Every message consists of one or more logical parts, each of which is associated with a type defined in the `<types>` container element. The parts can refer to message parameters or message return values and can be compared to the parameters of a function call in a traditional programming language.

- The `<portType>` element defines the operations that are supported by the Web service.

- The `<binding>` element defines the message format and protocol details for operations and messages defined by each portType.

- The `<port>` element defines an individual service endpoint by specifying a single address for a binding.

- The `<service>` element groups a set of related ports together.

The WSDL document structure is presented in Figure 2.6.

2.2.3 SOAP

SOAP, formerly defined as Simple Access Object Protocol, is one of the most significant Web service technologies. It is a lightweight protocol intended to describe the exchange of structured information between peers in a decentralized, distributed environment [13]. Using XML technologies, it defines an extensible messaging framework to provide a message construct that can be exchanged over a variety of underlying protocols, such as Hypertext Transfer Protocol (HTTP). Although SOAP is fundamentally a stateless, one-way message exchange paradigm, applications may create more complex interaction patterns.

```
<definitions>
    <types>
        definition of types
    </types>
    <message>
        definition of a message
    </message>
    <portType>
        definition of a portType
    </portType>
    <binding>
        definition of a binding
    </binding>
    <service>
        <port>
            definition of a port
        </port>
    <service>
</definitions>
```

Figure 2.6: WSDL document structure.

A SOAP message is formally specified as an XML Information Set which provides an abstract description of its contents. Its structure consists of three major parts, or blocks. The *envelop*, the *header* and the *body*. The *envelope*, which is a required part, marks the beginning and the end of the message and constitutes the basic communication unit. The *header* part is optional and may contain one or more header blocks which enclose the message's attributes or define technical details for the message. Headers may contain application-defined information associated with the message such as security tokens and transaction identifiers, as well as information related to how the message will be handled by intermediaries and its final destination. Finally, the *body* part is required and consists of one or more body parts which comprise the actual message. Figure 2.7 illustrates an example of a SOAP message, identifying the above mentioned

structures and information.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
      <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
         <n:priority>1</n:priority>
         <n:expires>2001-06-22T14:00:00-05:00</n:expires>
      </n:alertcontrol>
   </env:Header>
   <env:Body>
      <m:alert xmlns:m="http://example.org/alert">
         <m:msg>Pick up Mary at school at 2pm</m:msg>
      </m:alert>
   </env:Body>
</env:Envelope>
```

Figure 2.7: SOAP message example.

2.3 Apache Axis2

Axis2 is an open source, XML-based Web service framework developed under the auspices of the Apache Software Foundation and it is basically a successor to Apache SOAP and the Apache Axis SOAP stack. It consists of a Java and a C++ implementation of the SOAP protocol, as well as several utilities and APIs for generating and deploying Web service applications.

The purpose of the Axis2 framework is to conceal details relevant to the SOAP protocol and WSDL documents from the programmer. When developing Web services, matters like the implementation of logic for encoding, decoding, sending or accepting requests and responses have to be considered. However, by using Axis2, a programmer avoids all this tedious work, and is free to dive into the really important part of the Web service implementation, which is the development of the actual service.

The main feature of Axis2 that we use in this thesis is a utility called WSDL2Code. This is a code-generating tool that is responsible for parsing a WSDL document and generating appropriate code to handle Web service requests and responses. Depending on the parameters passed to the WSDL2Code utility, it generates client and server stubs for calling or implementing a Web service matching the WSDL document. As a result, all the programmer needs to do in order to invoke a Web service, is to make the method calls on the Web service object as if it were a local object. We use this utility to generate the client proxy which is the collection of classes that work together to build and process SOAP messages on the client side.

2.4 Related Work

To our knowledge, there has not been much work on dynamically creating clients for Web services, especially in mashup environments. The only published project that we are aware of is the Dynamic RESTful Web Service Client (DRWSC) which is described in [17].

DRWSC – is a standalone application which, based on the ability of the WSDL 2.0 language to describe RESTful Web services, uses a WSDL document to invoke the referring service. To invoke a Web service, the user enters the WSDL 2.0 document's URI, the document is retrieved and parsed and then the user selects the desired operation and enters the required input arguments. The service is invoked and the output is presented to the user.

Although DRWSC seems to have a lot in common with our work, there are two key differences. First of all, DRWSC is used to invoke RESTful services which are described by WSDL 2.0 documents. However, even though WSDL 2.0 is a W3C recommendation, it is currently neither widely accepted nor supported in commercial and academic areas, and therefore very few RESTful services are exposed by WSDL 2.0 documents. This problem on the other hand does not exist with SOAP services, since describing a SOAP Web service by WSDL has been the standard methodology for several years.

The second and very important difference between the two projects lies in the implementation approach. DRWSC is implemented as a standalone application

which means that all it can provide is the actual service invocation, without any logic integrated. The only way to enter a service's input is manually and its output can only be displayed to the user and cannot be used by another application. On the contrary, we integrated our dynamic Web service client generator in a mashup environment. In this manner the invocation can actually be usable, since the logic of the client can be implemented by the mashup application. The service's input can be supplied by numerous data sources (including other Web services), and its output can be processed and used by other parts of the mashup.

Chapter 3

Mashup Platforms

3.1 Current State

Many mashup tools have been developed to support the creation and execution both of consumer-focused and enterprise mashup applications. Here, we briefly introduce some of the most popular tools, which are more extensively analyzed in [3, 4].

- *Yahoo! pipes*[1] (developed by Yahoo! Inc.) is a Web-based, consumer-oriented mashup platform. Mashups – here called pipes – are created by connecting widgets provided by the platform. Currently data from Web feeds, Web pages and other services, like flickr[2], can be mashed. Output can be accessed by a client as RSS or JSON, or can be visualized on a Yahoo! Map, or through an HTML page.

- *Damia*[1, 8], is an enterprise-oriented mashup platform developed by IBM. It enables users to create mashups by assembling data feeds from Internet as well as enterprise data sources. It mainly focuses on data feed aggregation and allows additional tools – like feed readers – to be used at the presentation layer for the data feeds that it provides.

[1] http://pipes.yahoo.com/pipes/
[2] http://www.flickr.com

- *Apatar*[1] is a mashup data integration platform developed by Apatar Inc. It allows users to aggregate and integrate locally-stored data with the Web by using a visual editor to create mashups. Apatar mainly aims in manipulating data that will be used from other applications and thus its output can be consumed by external tools.

- *Exhibit*[2] is a framework for creating web pages with dynamic and rich visualizations of structured data. It enables its users to aggregate data obtained in various formats, like RDF/XML and Bibtex. Exhibit uses HTML pages as output but it also provides functionality for exporting its output to different formats, such as RDF/XML or Exhibit JSON.

- *MashMaker*[11, 12] is an interactive Web-based tool developed by Intel Corporation for editing, querying, manipulating and visualizing semi-structured data. It differs from other tools in the sense that it works directly on Web pages and allows users to create mashups when browsing by combining content from different Web pages. The final goal of MashMaker is to suggest mashups or widgets for the visited Web pages, that the user may want to use.

There are certainly many more mashup tools and platforms currently available, but describing and analyzing them is beyond the scope of this thesis.

3.2 Platform Selection

The goal of our work is to enable non expert users to dynamically invoke Web services in mashup applications. Hence, we needed a basis to start building on, so that we could avoid creating our own mashup platform and, rather, concentrate on our real goal.

Apatar is a mashup data integration tool that best meets our needs for several reasons. First of all, it essentially uses the mashup paradigm to support data integration. It provides a visual job designer through which the user may intuitively connect widgets to create a data integration schema by dragging and

[1] http://www.apatar.com

dropping. The absence of the need to write code, enables the platform's usage by users with very limited technical knowledge. Apatar also provides mechanisms – called connectors – for accessing and manipulating data stored locally or data from corporate resources. Additionally, connectors for accessing content on the Web, such as RSS feeds and some popular Web 2.0 APIs as Flickr, Salesforce.com and Amazon Simple Storage Service (Amazon S3) are provided. Therefore, one can access data stored in spreadsheet documents in a local hard drive and mash it with data residing in a corporate database or even data stored in the cloud (e.g, in Amazon S3). This feature also qualifies Apatar as a potential enterprise-oriented mashup tool. Finally, Apatar supports operators for manipulating data by performing aggregations, filtering, joins, transformations and so on. Hence, we concluded that, as far as functionality is concerned, this set of capabilities makes Apatar a good choice as a base to start building on.

However, there are also some technical aspects that we had to consider. The most important one is the source code availability. Apatar is open source software, distributed under the GNU General Public Licence[1] (GPL). This is a key feature, because it allows us to actually use its source code, modify it and extend it to meet our goals. We must note here, that Apatar is one of the very few options offering source code availability since the source code of most mashup platforms is not publicly available. Another significant feature is that Apatar's source code is very well structured thus simplifying the procedure of reading and understanding it, which is necessary in order to modify and extend it. Finally, Apatar is designed to be extensible, and thus, it provides a standard procedure for the developer to create new functionality and plug it in the core application engine. These features, as well as those mentioned in the previous paragraph, make Apatar the best available choice for a starting point for our work.

3.3 Apatar's Architecture

Apatar is an open source Extract Transform and Load (ETL) project. As illustrated in Figure 3.1, it is structured in three basic components: The core component, the connectors' component and the user interface component.

[1] http://www.gnu.org/licences/gpl.html

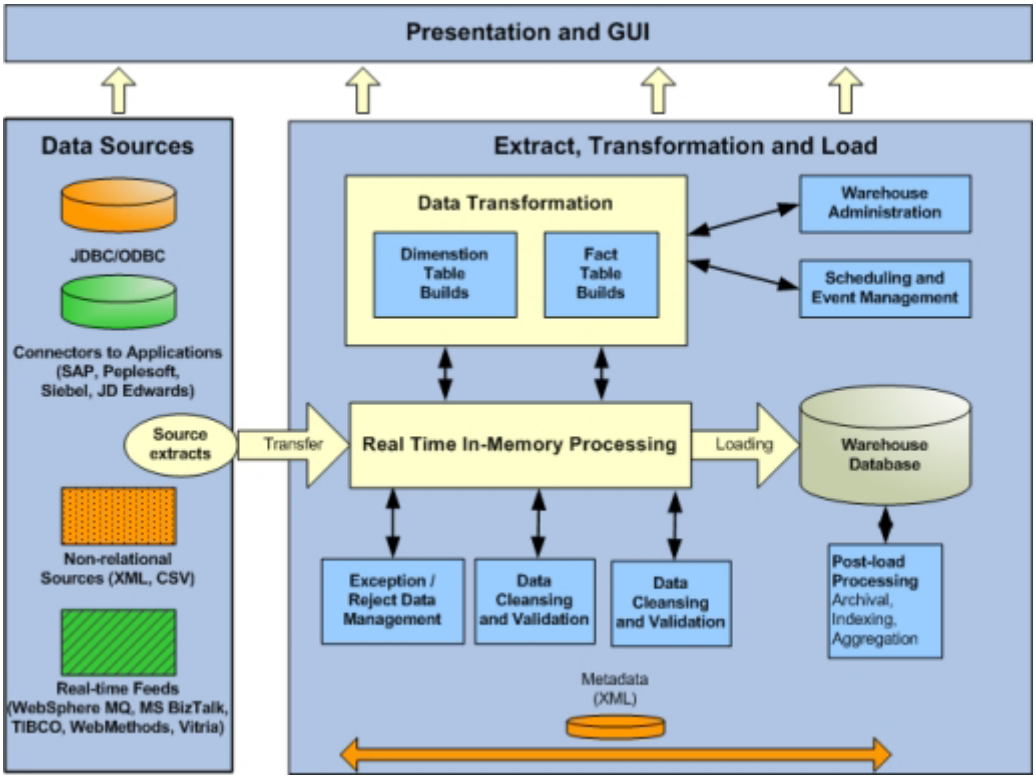

Figure 3.1: Apatar's architecture.

3.3.1 Core Engine

The core component consists mainly of the application's ETL engine, which is where the actual data processing takes place. For every operation, data is retrieved from one or more data sources through the connectors' component and is transformed to tuples in Apatar's internal database. In this form, data is processed by the application's engine, and then loaded again to one or more connectors and probably to the presentation layer. Some of the operations that are currently available are high level operations, such as joins, selections, aggregations, filtering and so on, as well as lower level operations, such as transformations between different data types.

The core component is also responsible for defining fundamental structures to hold information relevant to the data manipulation, as well as for providing a mechanism to support and ensure the platform's consistency and extensibility. For example, the core component defines structures to represent the platform's

internal database, its tables and its records, and also abstractly defines the way that a connector must be structured in order to be functional.

3.3.2 Connectors

The connectors' component is used to connect the application's core engine with data sources. Every connector provides a connection point for a specific data source through which data can be read, written, or both. Currently, a large set of connectors are provided and the supported data sources can vary from corporate databases and personal files, to e-mails and Web 2.0 APIs.

3.3.3 GUI and Data Representation Layer

Finally, the third component consists of a graphical user interface and a simple data presentation layer that simplifies both the use of the application and user control over data. Through the GUI, users interact with Apatar, to create, modify, publish or run mashup applications, while the data presentation layer enables data supervision at any stage of the workflow.

The main application window is divided in two areas. The connectors and functions' area, where the different connectors and functions are displayed as widgets, and the work area which is used for creating mashups – called datamaps. To create a mashup application, the user just needs to drug and drop the necessary connectors in the work area, configure them and connect them together to form a datamap.

3.3.4 Extensibility

As stated earlier, Apatar is designed to be an extensible platform, a goal achieved through the use of the Java Plug-in Framework (JPF). JPF provides a runtime engine which can dynamically discover and load plug-ins. A plug-in is considered to be a structured component that describes itself by the use of a manifest. Plug-ins and the functions they provide are added to a registry at start-up-time or at run-time, but are not loaded until they are called. In this manner,

applications using JPF avoid paying any memory or performance penalty for plug-ins that are installed but not used.

Everything in Apatar is implemented as a plug-in. Every component, from core components and functions to connectors and GUI, is described by an XML document called `plugin.xml`. This document contains all the necessary information to describe the plug-in to JPF so that it can be registered in the framework and loaded upon call. This information would be the plug-in's identification, the path of the implementation classes, references to other plug-ins that are required, and so on. Every time the application starts, a predefined plug-in folder is scanned for the manifest files, the available plug-ins are registered to JPF, and, from this point on, they are ready to be used on demand.

Chapter 4

The DYNAMO Platform

The key contribution of our work is to extend the Apatar platform with extra functionalities we consider quite important and useful. We developed a connector for the Google Maps API and implemented a set of functions to support geographical data representation according to user needs. The second and more important part of our contribution consists of an effort to give every user the capability to extend the Apatar mashup platform at will, by providing support for any available Web service, without the need to write code.

4.1 Google Maps Connector

Apatar lacks a mechanism for providing the user with intuitive presentation of simple geographical data such as street addresses. Since this is a very useful feature – especially for designing mashups – we had to implement a special connector that would be able to visualize a given set of street addresses and generally handle at some level geographical data. To accomplish our goal, we needed a service that would provide us with the maps and a means to mark locations on them. Some of the available options were Yahoo! maps by Yahoo! Inc, Bing Maps by Microsoft Corporation and Google Maps by Google Inc. We decided to make use of the Google Maps service since a great percentage of Web users are already familiar with it and it also provides a simple to use API [6].

The main obstacle in the implementation of the connector is that while Apatar is a desktop Java application, the Google Maps API is based on JavaScript and

designed to be accessed via a Web browser. To resolve this conflict, we need an external HTML file that implements all the necessary JavaScript and HTML code. The main idea is to create this file dynamically and then feed its URI to the system's Web browser. The code included in the file – called `gMaps.html` – is generated at run-time according to the needs of each function of our connector. The JavaScript part of `gMaps.html` is responsible for calling the Google Maps API while the HTML part is responsible for presenting the output to the user. An address is represented at the output as a marker pinned on a corresponding position on a map. The connector's input must be one or more valid street addresses and may come as the output of any other Apatar connector. If an address is not valid, it will not be presented and the map's output will either be the default Google Maps location or just the remaining valid addresses. Figure 4.1 illustrates the architecture of our connector.

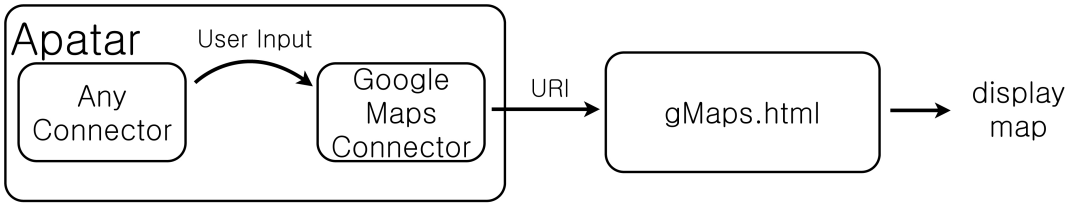

Figure 4.1: Google Maps connector's architecture.

Currently, three functions are implemented in our Google Maps connector: `pinAddresses()`, `pinAddressesInRange()`, and `pinClosestAddress()`. `pinAddresses()` takes as input one or more street addresses and simply displays them as markers on a map, while `pinAddressesInRange()` is a little more complicated: Its input is an origin address, a range distance and a set of addresses representing points of interest, such as hotels, restaurants, banks and so on. The output map are depicts only the points of interest whose driving distance from the origin address respects the limit imposed by the range. To acquire the driving distances between locations we use the Google Directions API [5]. This API allows us to obtain an XML document that contains the directions from one location to another as well as the driving distance between them. Note that the XML document may contain many alternative routes in which case we consider only the distance of the shortest one. After acquiring the document for each

address, it is only a matter of a simple comparison between each distance and the specified range to decide which address will be displayed on the output map. Finally, `pinClosestAddress()` takes as input a set of reference addresses as well as a set of addresses of interesting points, and for each reference address displays the closest interesting point on the output map. For this function we also use Google Directions API to acquire the driving distances between two addresses.

Our three connector functions also support an extra information presentation layer by making use of Information Windows, a functionality provided by the Google Maps API. For every address at the connector's input, the user may define an information string, referring to the particular point of interest, to be displayed on the map. By clicking any marker on the output map an information window pops up and the corresponding address as well as user defined information are displayed.

4.2 Dynamic Web Service Client Generator

A main goal of this thesis is to present a simple way for non expert users to dynamically invoke Web services without the need to implement a client application by themselves. Such functionality would enable a mashup platform to be extended dynamically to support any Web service and not merely a predefined set. This, obviously, allows for significant agility in data manipulation tasks that can be achieved by mashup applications, since the available data sources would be significantly increased, and they could be appropriately customized to each user's needs.

Generally, to utilize the data exposed by a Web service, a client application is required. After the procedure of engaging a Web service, which was presented earlier, the *Requester Agent* (client) will interact with the *Provider Agent* (Web service implementation) to consume its output. Our goal then, is to automate the creation of the client application for a given Web service and integrate it with our mashup platform. Of course, the whole process is concealed from the user who only interacts with the system through the mashup GUI (e.g widgets). We currently consider the Web service discovery task as external to our application, and assume the user already knows the location of the desired Web service's

WSDL document. This location forms the input to our Dynamic Web Service Client Generator. However, future versions of DYNAMO will support intuitive Web service discovery based on UDDI, which will simplify the procedure for the user.

As discussed earlier, everything in Apatar is implemented as a plug-in. Our solution in DYNAMO basically works as a meta-plug-in that is able to create new plug-ins, each of which corresponds to a Web service connector. The main idea is to analyze the WSDL document provided by the service and generate all the necessary code that implements a client for the particular service. The client generation process consists of three stages. The first stage creates the plug-in's file structure and generates of the client proxy. The second stage then generates the necessary source code, and the final stage compiles the whole plug-in. After these stages, the user has to restart the application in order for the plug-in to be registered to JPF and be ready for usage. In the remainder of this chapter, we examine the three stages of the DYNAMO meta-plug-in in more detail.

4.2.1 Stage One: Plug-in structure and Proxy Generation

This first stage of the procedure can be considered as a preparation stage. Based on the service name we extract from the WSDL document, we create the necessary folders and the `plugin.xml` file that is the manifest file of the new plug-in. Then, we make use of Apache Axis2 framework to generate the client proxy.

Generally, there are three implementation options for the client proxy. The first option implements static binding for the Web service. This way, the client proxy is compiled and bound at development time. This binding is tightly bound to one and only one service implementation, but also provides the fastest performance of the three options. With the second option – dynamic binding – only the interface to a service type is compiled at development time and the client can bind to any service implementation that supports the specific `<portType>`. However, this approach adds a performance penalty since the WSDL document has to be retrieved and processed in order to complete a binding. Finally, the

third option is dynamic invocation. With dynamic invocation nothing is compiled at development time; rather, for every invocation, the application retrieves and interprets the WSDL document at runtime and dynamically constructs calls. It obviously supports maximum flexibility, but also carries a large performance penalty that essentially occurs on every invocation.

For DYNAMO, we choose the static binding option for the client proxy so that it can achieve maximum performance. We compromise with the least flexibility that static binding offers, firstly because we consider performance very important in mashup applications and secondly because it is quite easy for the user to replace the connector to a service with its updated version when necessary.

To create the client proxy, we use the WSDL2Code utility of Axis2 framework, which is responsible for generating stub classes matching a given WSDL document. Here there are two options with respect to the data binding system. Axis Data Binding (ADB) and XML Beans. ADB provides good – and growing – support for code generation from schema and also produces a very simple Java model for a given schema. Furthermore, ADB supports unwrapped service methods and automatic attachment handling, which makes it a top choice when working from existing WSDL service definitions as in our case. XML Beans, on the other hand, provides the most complete support for modelling schema structures, but also creates more complex Java models than ADB and does not support unwrapped service methods. Thus, we chose to use ADB data binding system.

With ADB, WSDL2Code generates (1) a Java class corresponding to every type element defined in the `<types>` container element, and, (2) a Java method corresponding to every operation exposed by the Web service. The mapping between WSDL types and Java classes as well as WSDL operations and Java methods is pretty straightforward. Invoking the service is as simple as calling the method corresponding to the operation the user wants to invoke and passing as arguments the classes corresponding to the operation's input. The method's output will be a class corresponding to the operation's output type.

However, even if this procedure appears to be a simple programming task, performing it dynamically (at execution-time) raises several more complex issues. Since WSDL uses W3C XML Schema as the default data typing system, we cannot expect that every type in a WSDL document would be a simple type.

In fact, in real world Web services, the vast majority of the types are complex, meaning that a type element consists of not only simple data types (integers, strings, boolean values, etc.), but also other complex or simple types, arrays of types, enumerations, and so on. To handle such cases, we need to analyze the WSDL document and the types schema, to decide which classes should be initialized and used as input to which method. Apart from the technical nature of the problem, there is also a logical issue due to the fact that semantics are not standard. For example, a person can b described by their first and last name or by their name and address. Even if we could technically avoid analyzing the complex types to simple ones, we would not be able to overcome the logical problem of semantics mismatch. In our case, the problem would be manifested in the form of data mapping across two widgets. A DYNAMO user would easily map a text value to a text field that describes a name, but she would not be able to map anything to a "person" field. This makes it obvious that every complex data type must be broken down to simple ones. This complex data type analysis process is part of the second stage of the plug-in creation in DYNAMO.

4.2.2 Stage Two: DYNAMO Connector Architecture Implementation

4.2.2.1 Conector's Architecture

This stage includes the implementation of the new data connectors' architecture. Since we need our new plug-ins to be visible and usable by Apatar, they must conform to a specific architecture imposed by the platform. Thus, every connector must be created by a class that implements the *NodeFactory*[1] interface which is declared in the Core Engine of the application. To implement the *NodeFactory* interface, we create the *ConnectorNodeFactory* class. Its task is to instantiate a *ConnectorNode* object, which implements the Web service connector. The *ConnectorNode* object, among other functionalities, is used to retrieve data from Apatar's internal database and pass it to the function of the plug-in that needs it. We model each operation exposed by the Web service as a

[1] Any widget can be considered a *node* in Apatar. So, connectors, functions, operations etc. can all be considered as *nodes*.

ConnectorTable object, which defines the argument and return types of the operation as well as its name. The *ConnectorTable* object invokes a method of the *Functions* object which implements the operation invocation. The operations are not directly implemented in the *ConnectorTable* object so that the generality of the connector generation process can be preserved. The connector's UML class diagram is presented in Figure 4.2. In our implementation, we also use an extra auxiliary class – called *ConnectorUtils* – which is not depicted in the class diagram in order to keep it as simple as possible.

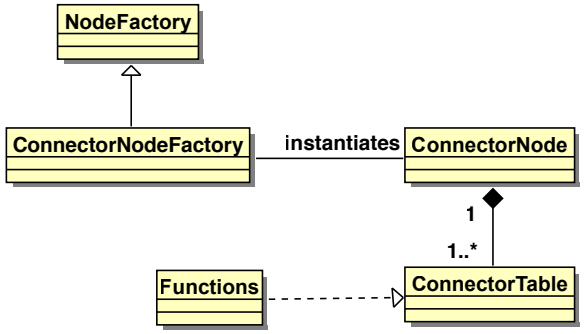

Figure 4.2: DYNAMO connector architecture.

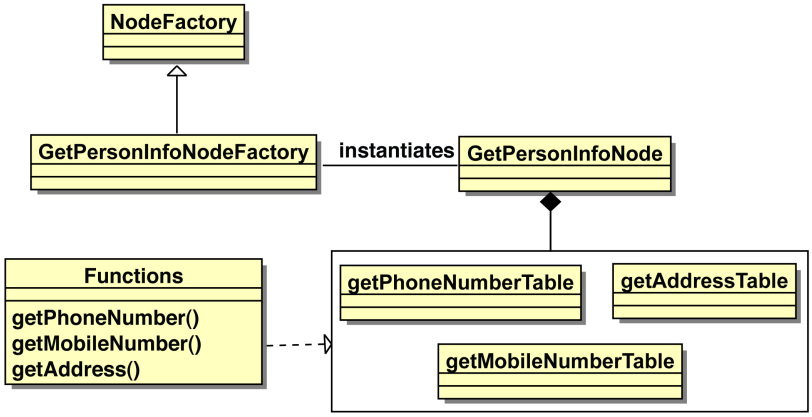

Figure 4.3: Example DYNAMO connector.

To further clarify the DYNAMO connector architecture, Figure 4.3 illustrates an example connector for the `getPersonInfo` Web service that exposes the operations `getAddress`, `getPhoneNumber` and `getMobileNumber`. At the lower right

corner of the figure, the box depicts three instances of the *ConnectorTable* object, one for each service operation.

To implement this architecture, we have to dynamically generate source code for numerous classes. This process is illustrated in Figure 4.4. We begin with the *ConnectorNodeFactory*, *ConnectorUtils*, and *ConnectorTable* – step (1) of Figure 4.4 – whose source code is quite generic and more relevant to complying with the underlying mashup platform's architecture, rather than to implementing a specific service client. Consequently, the same code can be used for every Web service simply by making minor adjustments, such as changing the name of the *Node* class or the path to the widget's icon to suit the different services.

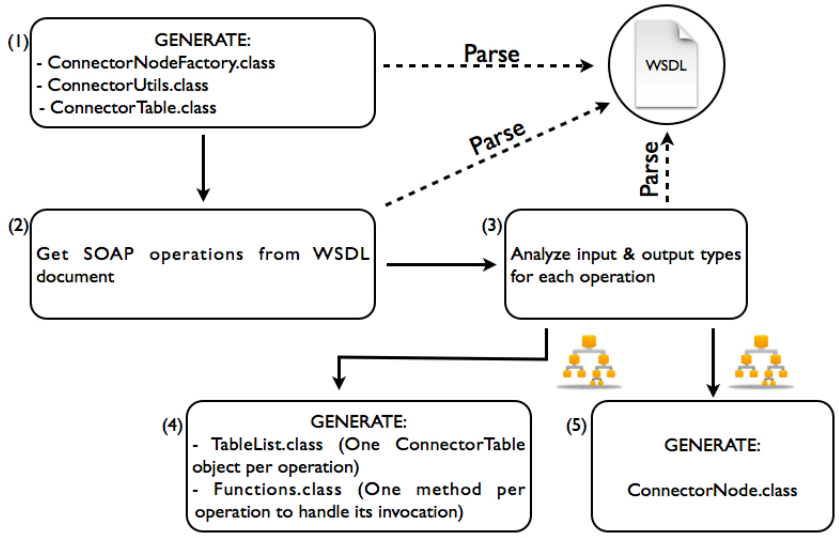

Figure 4.4: DYNAMO Execution Flow.

The next step is to parse the WSDL document to extract information about the described operations. Currently, DYNAMO supports only the SOAP and SOAP 1.2 protocols and, thus, extracts a list of all the available operations that conform to these two protocols – step (2) of Figure 4.4. Based on this information, we generate the *TableList* and *Functions* classes. The *TableList* class is basically a data structure that holds the different instances of the *ConnectorTable* object. In this class, one instance of the *ConnectorTable* object is created for every available operation, and then inserted in *TableList*. The name of the table stems directly from the name of the operation, while the argument and return types

come as a result of a more complex analysis of the document, which takes place simultaneously with the generation of the *Functions* class – step (4) in Figure 4.4. This class contains one method for each service operation and each method implements the invocation of the related operation. To generate the source code for each method we must process information referring to the schema of the input and output types of each operation – step (3) in Figure 4.4.

4.2.2.2 Complex Data Type Analysis

Our approach for this task suggests the use of a tree structure to hold any complex type. The root would represent the complex element, every level of the tree would hold the nested elements while the leaves would be occupied just by simple types. Each node of the analysis tree holds information relevant to the element it represents such as its name and type, or flags indicating an array or an enumeration etc. As an example, Figure 4.5 presents a sample complex type and the output of our analysis for that complex type.

4.2.2.3 Input Data Type Analysis

From the WSDL document we extract a list of the input types corresponding to the service operation we are currently processing. For every type in the list, we parse its schema from the `<types>` element of the document, and we generate a tree to represent it (as described in the previous paragraph). Then, using the Depth First Search (DFS) algorithm, we traverse our structure and generate code to initialize, in a bottom-up way, the classes that represent each node in our tree. Intuitively, the bottom-up initialization can be considered as "building" a complex data type from simpler components. After every class that corresponds to the inputs of the operation has been initialized, we call the method from the stub class that matches our operation and we pass as arguments the classes we just initialized. If for instance the `personInfo` type – depicted in Figure 4.5 – was referring to the input of a `foo()` operation, the generated code would be:

```
String street = user_input_for_street;
String number = user_input_for_number;
String city   = user_input_for_city;
```

```
<complexType name="streetAddress">
    <sequence>
        <element name="street" type="string"/>
        <element name="number" type="string"/>
        <element name="city" type="string"/>
    </sequence>
</complexType>

<complexType name="pesronName">
    <sequence>
        <element name="lastName" type="string"/>
        <element name="firstName" type="string"/>
    </sequence>
</complexType>

<complexType name="personInfo">
    <sequence>
        <element name="address" type="streetAddress"/>
        <element name="name" type="personName"/>
    </sequence>
</complexType>
```

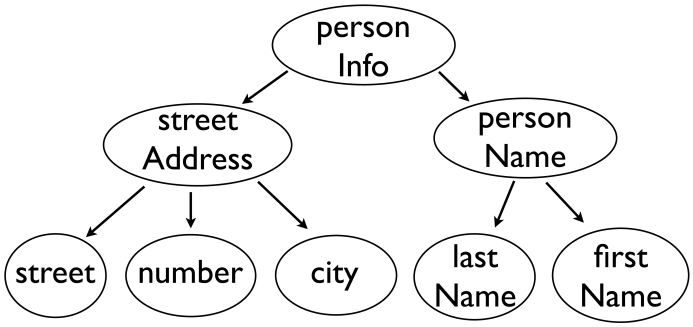

Figure 4.5: Analysis output for personInfo complex type.

```
StreetAddress address = new StreetAddress(street,number,city);

String lastName  = user_input_for_lastName;
String firstName = user_input_for_firstName;
PersonName name  = new PersonName(lastName,fisrtName);

PersonInfo info  = new PersonInfo(address,name);

OutputType output = stubClass.foo(info); /* Invocation of foo() */
```

which is the necessary code for invoking the `foo()` operation. Finally, we register to the *ConnectorTable* object the simple types that our analysis returned, so that they can be treated as inputs for the method that corresponds to the operation that is currently being processed. In our example, these inputs would be the variables `user_input_for_street`, `user_input_for_number` etc.

4.2.2.4 Output Data Type Analysis

Having invoked an operation we need a way to consume its output. Similar to the case of the input, the output of an operation is most often a complex data type and, thus, we need to break it down to simple data types as well. Following the same procedure as we did for the input types, we create a tree to represent complex output types. Afterwards, we use again the Depth First Search algorithm to traverse our tree and generate code to export the simple data types from the classes representing the complex types. While we initialized the input classes in a bottom-up way, to extract the simple types we work in a top-down fashion. This is mandatory because all the simple data types we need are wrapped in the return complex data type. Therefore we traverse our analysis tree from the root to the leaves and finally extract the information we need. If the `personInfo` complex type was the output type of `foo()`, the generated code would be the following:

```
PersonInfo info = stubClass.foo(inputs); /* Invocation of foo() */

StreetAddress address = info.getStreetAddress();
String street = address.getStreet();
```

```
String number = address.getNumber();
String city   = address.getCity();

PersonName name  = info.getPersonName();
String lastName  = name.getLastName();
String firstName = name.getFirstName();
```

The extracted simple types are registered to the *ConnectorTable* object so that they can be used as outputs of the service operation we are currently processing and the output values are transformed into tuples and stored in the Apatar internal database, in order to be used by other widgets or fed to the presentation layer to be displayed to the user.

4.2.2.5 Connector's Generation Completion

The final phase of the stage is to create the *ConnectorNode* class – step (5) in Figure 4.4. This class acts as a data entry point for the plug-in, since it retrieves data from the mashup platform database and passes it to the plug-in's functions. *ConnectorNode* also handles other important matters, such as the connector's user interface. A crucial function of this class is to expose the plug-in's inputs and outputs as widget endpoints so that the generated widget can be connected to others. In order to dynamically create this class, we need to know all the data types that the plug-in's functions use – this is exactly the information acquired by the analysis phase of this stage.

4.2.3 Stage Three: Compilation

With stage two, the source code generation for the new plug-in ends. In this stage, the system's Java compiler is invoked to compile the generated source code. If there are errors in the source code, all the generated files are deleted and an information window pops up to inform the user about the failure. Otherwise, the user is prompted to restart the application. When restarting, the JPF framework discovers, identifies and registers the new plug-in. The new connector is displayed among with the other connectors under the category "*Web Services*" and the user may use it as any other available widget.

Chapter 5

Demonstration

To demonstrate our work we have created a mashup application in Apatar, that would use both DYNAMO and our geocoding functions. The scenario for the application is that of a travelling agency that organizes trips to Athens. The agency stores the addresses of the sights that will be visited in a spreadsheet file and needs to find information about five-star hotels close to these sights. To accomplish that, an employee creates a mashup application to combine data from the spreadsheet with data acquired by a Web service that exposes information about hotels in Athens. The output of the application is a map showing the interesting sights, and, for each sight its closest five-star hotel.

However, finding the desired Web service was not easy. In January of 2006 IBM, Microsoft and OASIS discontinued their public UDDI registry, shifting their interest in private, enterprise versions of UDDI, thus creating problems in finding public WSDL Web services. To surmount this difficulty we had to create our own hotel-information Web service. To do that, we extracted information about hotels in Athens from tripadvisor.com[1], and stored them in a private database. For the data extraction, we used a custom-made Web crawling application and extracted data on hotel class, address and name, as well as user ratings and reviews for every available hotel. Afterwards, we used Apache Axis2 framework and Apache Tomcat Web server to create and deploy a simple hotel Web service which would expose our data. Our service (`hotelsInfo`) returns upon request an

[1] http://www.tripadvisor.com

`ArrayOfHotel` object which obviously is an array of `Hotel` objects. Figure 5.4 shows a snippet of our service's WSDL document referring to the return type.

To create the application, the agency employee uses DYNAMO to dynamically create a widget responsible for invoking the `hotelsInfo` service. The service returns information about every available hotel in Athens but the user is interested only in luxury accommodations. Therefore, she filters the results and keeps only five-star hotels, whose address is then passed in the `pinClos/estAddress()` geocoding function of Google Maps connector. The other argument of this function is the sights' addresses that are stored in the spreadsheet file which is acquired by the proper Apatar connector, and the function's output is the desired map. The employee can also choose more information to be displayed on the result map, such as hotel names, user ratings, user reviews etc. Figure 5.1 depicts the complete mashup application, Figure 5.2 shows the data transformation from the filtered `hotelsInfo` results to the Google Maps input and Figure 5.3 illustrates the result map of our DYNAMO mashup.

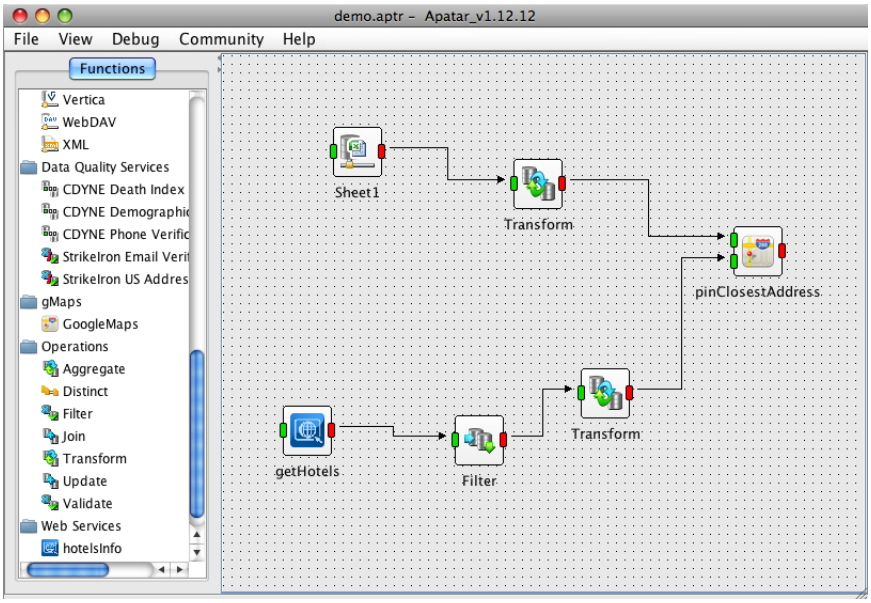

Figure 5.1: The mashup application.

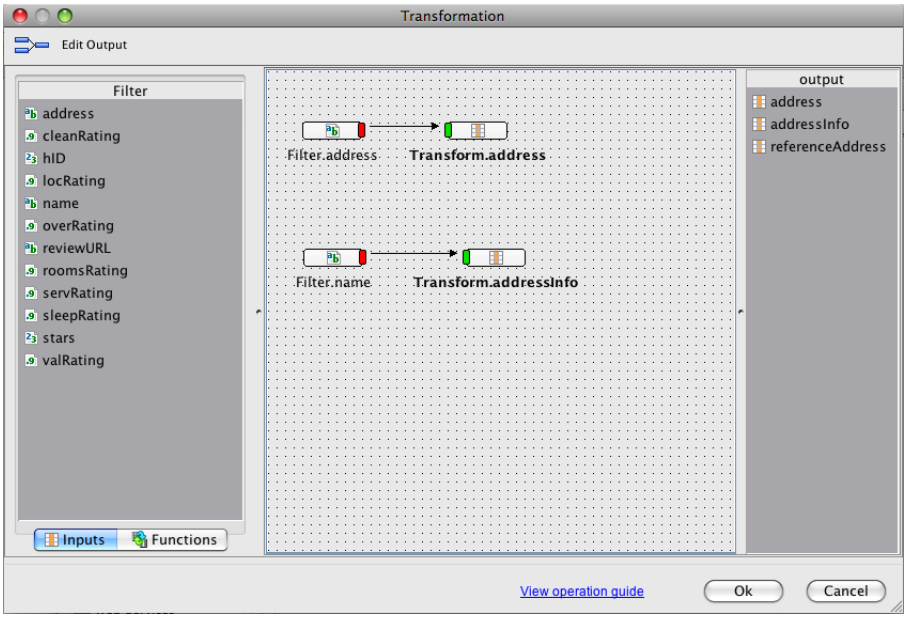

Figure 5.2: Transformation between hotelsInfo and Google maps.

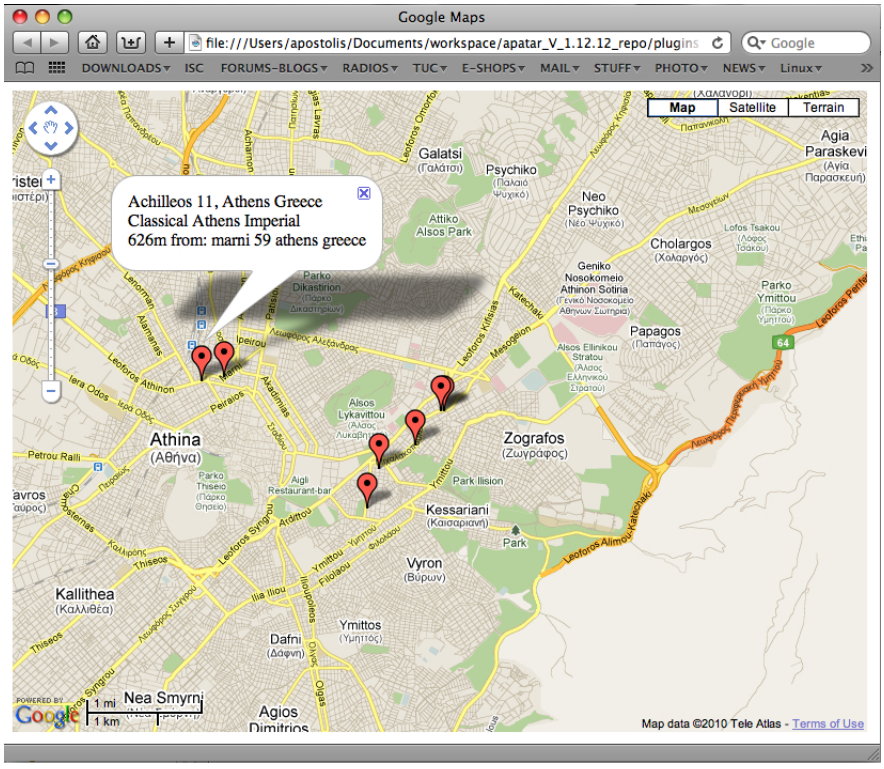

Figure 5.3: Result map.

```
<xs:schema attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="http://hotels/xsd">
  <xs:complexType name="ArrayOfHotel">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="hotels"
        nillable="true" type="ax21:Hotel"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Hotel">
    <xs:sequence>
      <xs:element name="address" nillable="true" type="xs:string"/>
      <xs:element name="cleanRating" type="xs:float"/>
      <xs:element name="hID" type="xs:long"/>
      <xs:element name="locRating" type="xs:float"/>
      <xs:element name="name" nillable="true" type="xs:string"/>
      <xs:element name="overRating" type="xs:float"/>
      <xs:element name="reviewURL" nillable="true"
                  type="xs:string"/>
      <xs:element name="roomsRating" type="xs:float"/>
      <xs:element name="servRating" type="xs:float"/>
      <xs:element name="sleepRating" type="xs:float"/>
      <xs:element name="stars" type="xs:int"/>
      <xs:element name="valRating" type="xs:float"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 5.4: WSDL snippet of hotelsInfo Web service.

Chapter 6

Conclusions and Future Work

Starting from the Apatar open-source mashup platform, we designed and implemented a novel system, DYNAMO, that automatically generates client applications for Web services and integrates them in a mashup environment. By extending a mashup platform with such functionality, we enable users to create custom widgets at run-time to invoke any Web service, without the need to write code or even understand programming concepts. Furthermore, we added support for handling and displaying geographical data in a map, which Apatar lacked. We consider both these features very important and helpful for mashup applications since the first one significantly extends data availability and agility in data manipulation as well as application personalization, and the second provides an intuitive visualization of geographical data which is becoming increasingly prominent in today's applications.

However, our system is still in a prototype version and apart from debugging and optimization, we also intend to add further functionality. First of all, as mentioned previously, WSDL 2.0 is not yet widely accepted and adopted by the industry and, thus, our application does not currently support it. Nevertheless, the newer version of WSDL is, for some years now, a W3C recommendation and, if we want to maximize the amount of the available Web services that can be used by DYNAMO, we need to support it as well.

Furthermore, considering the large number of RESTful services that are currently available, it is easy to see that we also have to enable their dynamic invocation through DYNAMO, in order to meet the needs of as many users as possible.

This would definitely be quite a challenge, since REST is an architectural style, not a protocol like SOAP, and, hence, there is no official standard for RESTful services. In addition to that, real world RESTful services are not described by a formal language like WSDL and their APIs are meant to be read by human beings and not by machines as in the case of SOAP and WSDL services.

Another interesting task is the extension of DYNAMO to also support dynamic Web service discovery. Since DYNAMO is intended for non-expert users, it is rather unrealistic to expect them to be able to provide the WSDL document URI for the Web service they need to invoke. Therefore, we need to provide them with a tool that can handle the discovery process, so that the input of the application would be more intuitive for the user.

Last, but not least, we intend to open-source our system. Since DYNAMO is based entirely on open-source projects – like Apache Axis2 and Apatar – we consider it our obligation to offer our work back to the open-source community so that it can be used or even improved by anyone willing to do so.

References

[1] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, A. Singh. Damia: Data Mashups for Intranet Applications. *SIGMOD '08*, 2008. 15

[2] D. F. Huynh, D. R. Karger, R. C. Miller. Exhibit: lightweight structured data publishing. *WWW '07, New York ,USA*, pages 737–746, 2007. ACM. 16

[3] G. D. Lorenzo, H. Hacid, H. young Paik. Data Integration in Mashups. *SIGMOD Record*, 38(1), March 2009. 15

[4] G. D. Lorenzo, H. Hacid, H. young Paik, B. Benatallah. Mashups for Data Integration: An analysis. *Technical Report UNSW-CSE-TR-0810*, 2008. 15

[5] Google Inc. Google Directions API, http://code.google.com/apis/maps/documentation/directions/. 22

[6] Google Inc. Google Maps JavaScript API V3, http://code.google.com/apis/maps/documentation/javascript/. 21

[7] J. Anant. Enterprise Information Mashups: Integrating Information Simply. *VLDB '06*, 2006. 5

[8] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, A. Singh. Damia – A Data Mashup Fabric for Intranet Applications. *VLDB '07*, 2007. 15

[9] M.P. Papazoglou and D. Georgakopoulos. Service Oriented Computing. *Comm. ACM*, 46(10):24–28, 2003. 1

[10] OASIS. UDDI Version 3.0.2 – UDDI Spec Technical Committee Draft, 19 October 2004, http://www.uddi.org/pubs/uddi_v3.htm. 8

[11] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, P. Gandhi. Intel Mash Maker: Join the Web. *SIGMOD Record*, 36(4), December 2007. 16

[12] R. Ennals, M. Garofalakis. MashMaker: Mashups for the Masses. *SIGMOD'07*, June 2007. 16

[13] World Wide Web Consortium. SOAP Version 1.2, 27 april 2007 http://www.w3.org/TR/soap12-part1/. 10

[14] World Wide Web Consortium. Web services architecture, W3C working group note, 11 february 2004, http://www.w3.org/TR/ws-arch/. 7

[15] World Wide Web Consortium. Web services description language (WSDL) 1.1, W3C note, 15 march 2001, http://www.w3.org/TR/2001/NOTE-wsdl-20010315. 9

[16] World Wide Web Consortium. Web services glossary, W3C working group note, 11 february 2004, http://www.w3.org/TR/ws-gloss/. 6

[17] Yanguang Chen, Jiehui Li, Yi Lv, Haihuan Qin and Liang Zhang. DR-WSC – To simplify Dynamic Invocation for RESTful Web services. *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference*, August 2010. 13