Incorporating applications to a Service Oriented Architecture *

José A. García
University A Coruña
Department of Computer Science

Antonio Blanco
University A Coruña
Department of Computer Science

Roi Blanco
University A Coruña
Department of Computer Science

Abstract: How to adjust the technological expenses and how to adapt systems to business requests aiming at improving operative efficiency have become fundamental questions for many companies. These companies must be able to maximise their technology, processes and services, building a framework where a Service Oriented Architecture (**SOA**) appears as a key element for the change.
In this paper, we present a design and implementation that allows to reuse the logic of existing applications in a new execution environment. These applications should be able to communicate according to some standards, based on W3C specifications. To achieve this purpose the architecture developed follows the architectural patterns Model-View-Controller (MVC) and Layers. A modular design based on open structures plays a crucial role to reuse each one of the subsystems needed by the company to incorporate, orchestrate or coordinate new business processes. This way it is possible to share part of the logic already deployed during the design phase of new business processes, avoiding high development costs.

Key–Words: Web Engineering, Web Services.

1 Introduction

Technologically, service oriented architectures provide a new way of deploying business applications, combining real-time data with component-based systems. The main concept incorporated by this architecture is the interoperability of the applications, an idea, original by Paul Allen[5], based on developing reusable components that allow the reduction of the software development costs inside the environment of a company. If the subsystems are incorporated following the SOA architectural approach the software components of the company can also be used by third-parties. In our case of study (section 4), we defined a group of facades that allowed the integration of the application with any other process of the organisation, without any programming language or operating system restrictions.

The organisation of this paper is as follows. In section 2 we give a brief reference on the standards used in the project. In section 3, we present types or layers of integration systems and we review the principles of service oriented architecture. In section 4 we present our case of study in a real environment. Finally, in section 5 and 6 we present the conclusions, as well as future lines of research.

2 Background

In the year 2000, the W3C[1] accepted the bases of a new access protocol for simple objects (SOAP[2]). SOAP is a XML-based framework for the transmission of messages in businesses communications. These communications go through the HTTP[3] protocol and they incorporate a technological alternative to proprietary protocols as Java-RMI[4] and DCOM[5].
Java-RMI was developed by Sun Microsystems as a standard mechanism to allow the development of applications based on distributed objects within the Java platform. Java-RMI provides an environment where distributed Java applications are able to send and retrieve remote objects. DCOM, developed by Microsoft, allows COM applications to communicate with RPC (Remote Procedure Call) mechanisms. Also, and during

*Partly supported by MEC TIN2005-08986

[1] http://www.w3.org
[2] http://www.w3.org/TR/soap12-part1/
[3] http://www.w3.org/Protocols/
[4] http://www.java.sun.com/products
[5] http://www.microsoft.com/com/default.mspx

the following years, W3C published the specifications of WSDL[6] (Web Service Definition Language) and UDDI[7] (Universal Description Discovery and Integration). WSDL specifies a standard language for the definition of web services, and UDDI specifies the way of finding a web services supplier.

With the definition of these three standards, the first web services platform generation was established. Figure 1 shows the relationships between those specifications. WSDL allows the description using a XML-based syntax, of a SOAP accessed web service. Also, the figure shows how a WSDL document is kept in the UDDI repository for some external customers.

Nowadays, these organisms are trying to adjust the bases related to the quality of service issues, to develop a second generation of the architecture.

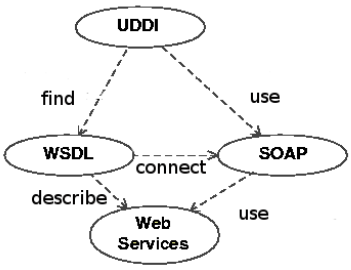

Figure 1: Relationships between specifications

3 Integrating applications

The mechanism of tackling an integration process should be thought as "how will the system inside the organisation evolve?" instead of taking an adhoc approach. This means that the software design should be made in such a way that it helps to incorporate modifications and new standardisation processes. This is, a system easily adaptable without a high-cost transformation process. There are four types or levels related to systems integration:

1. *Data-level* integration, allows the applications to share data without necessarily involving business logic. For example, the application A receives direct access to application B's database, and none of application B's logic is involved in the transfer of its data. This integration level, in most cases, leads to inconsistent shared data between applications, thus the poor reliability of this approach.

2. *Application-level* integration, allows the application A to make a request for information in the scope of the application B by accessing an interface or a group of interfaces (API). This way, A accesses some already programmed logic belonging to the application B. The main disadvantage of this integration level is that this API can be proprietary, this is, when working with different application platforms, an intermediate step is often introduced to translate incompatible communication protocols.

3. *Process-level* integration allows the merging of two or more existing processes. The new merge process is the result of integrating two or more applications, through some sort of shared-data bus. EAI (Enterprise Application Integration) encompasses methodologies such as object-oriented programming distributed, cross-platform communication message brokers with CORBA (Common Object Request Broker Architecture), and specialised middleware have made the messaging framework model a more suitable choice.

4. *Service-oriented* integration introduces Web services to establish a platform-independent interoperability model within various integration architectures. According to the W3C a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. Web services simply add new components that can be used an effectively in a wide number of architectures. Web services are based on the principles of service oriented architectures. This is the suitable integration level in our case, because:

   (a) The interfaces should be the less tied to a specific architecture as possible. This way, the *service* and *service description* definitions are independent, so there may be deployed just one description and many implementations. The description defines the service structure and indicates the type of the messages and the signature of each operation. The separation allows the construction of different implementations, according

[6]http://www.w3.org/TR/wsdl.html
[7]http://www.w3.org/TR/uddi.html

to the different performance requirements of the system.

(b) The way to locate these services should be transparent to the client of those services and they are able to locate a specific service by just using an UDDI repository or one service that acts as a deployed services registry.

(c) The communication protocol could be independent of the platform. SOAP is a messaging protocol of messages based on XML that follows this property.

3.1 Service Oriented Architecture

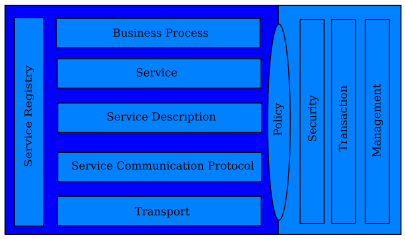

Figure 2: Elements of a SOA

Service-oriented architectures present an approach for building distributed systems that deliver application functionality as services to either end-user applications or to other services. The architecture stack is divided, according to IBM's vision[4], in two halves, with the left half addressing the functional aspects of the architecture and the right half addressing the quality of service aspects, as it is shown in figure 2. Each one of the components or layers of this architecture is described as follows:

- **Transport** is the mechanism used to move service requests between the service consumer and the service provider in any direction.
- **Service Communication Protocol** is an agreed mechanism that the service provider and the service consumer use to communicate what is being requested and what is being conveyed.
- **Service Description** is an agreed schema for describing what the service is, how it should be invoked, and what data is required to invoke the service successfully.
- **Service** describes an actual service that is made available for use.
- **Business Process** is a collection of services, invoked in a particular sequence with a particular set of rules, to meet a business requirement. A business process could be considered as a service on its own, which leads to the idea that business processes may be composed of services of different granularities.
- **Service Registry** is a repository of service and data descriptions which may be used by service providers to publish their services, and service consumers to discover available services. The service registry may provide other functions to services that require a centralised repository.
- **Policy** is a set of conditions or rules under which a service provider makes the service available to consumers.
- **Security** is the set of rules that might be applied to the identification, authorisation, and access control of service consumers invoking services.
- **Transaction** is the set of attributes that might be applied to a group of services to deliver a consistent result.
- **Management** is the set of attributes that might be applied to manage the services provided or consumed.

At the moment, the industry is working on developing new standards required to simplify the implementation of service oriented architectures.

4 SOA for legacy applications

Nowadays, the introduction of SOA's brings more flexibility to the behaviour of the applications of an organisation. In this work, we describe the process of moving from a static and inflexible model to a more adaptive one, where the standardisation of the procedures in all the layers arises as the operational key element.

4.1 Case of study

In this section, we outline the problem of adapting an application in an execution environment to the new requirements of the industry, i.e. we need to transform those business applications to the web parties, either for private use of companies or to offer services to others ones through the Internet. This particular case of study targets on a legacy

application built within a MVC approach. We were able to separate and to reuse the model layer, common to other applications inside the company, and to develop a framework for communicating any other process inside and outside the organisation in an uniform way.
For the client part, we built a small web application that implemented the authentication use case in the legacy application. This way, it is possible to show how a legacy application is able to offer its services in a new field and environment, without leaving the support in the old one.
To tackle these requirements, we employed the SOA architecture and web services to expose business processes.

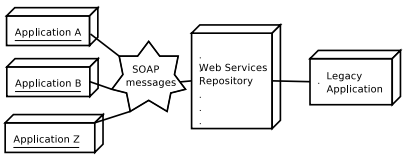

Figure 3: Main Approach

The controller layer of the application, establishes a communication at a web services repository using a set of SOAP messages. The message-passing protocol allows the retrieval of data from the model layer of the legacy application (fig. 3).

By using SOAP as a message protocol, any other process of the organisation is able to use this repository transparently. This repository should be standard and interoperable, given that the different deployed services might be used for applications running in different platforms. Even more, it is guaranteed that the repository can be used by different software manufactures due to the support of standard protocols.

Figure 4 shows an abstract diagram representing the main structure of the integration process studied. Client applications are able to access the

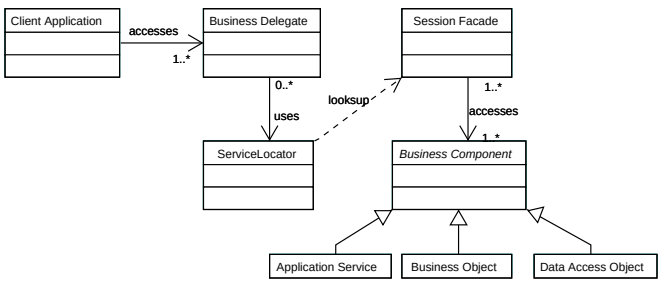

Figure 4: General Vision

logic of the remote service using the *Business Delegate* object. The process of communicating with the remote application is started by the Business Delegate structure, and it allows every client of the legacy application to access its services without the need of explicit knowledge about the communication system used.
The *Session Facade* design pattern acts as an encapsulation component for the business layer, and it sets the way of modelling the services offered by the different corporate applications. Therefore, every client application access the application business layer through this facade, instead of being directly plugged into the business logic. It also offers to those client applications one level of abstraction, because the same service can contain several *Business Components*, this is, it is possible to take into account every combination from many information sources inside a specific functionality from the *Session Facade*. This functionality would be completely transparent for clients.

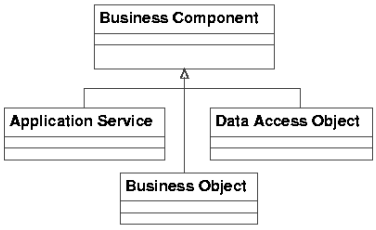

Figure 5: Alternative of implementation

As it is shown in figure 5, it is possible to implement *Business Component* objects in many different ways, according to the different encapsulation criteria, or if there are any underlying implementations or other applications below. The first alternative considers the use of *Business Objects* as the implementation of the Business Component; they allow the encapsulation of domain data from this pattern and the abstraction of the most important properties in the domain-logic objects, with the purpose of avoiding the communication with heavy objects.
Also, the *Business Component* object can be implemented using a *Data Access Object* (DAO) pattern that allows to hide the persistence of data coming from the remote application. In this specific case of study, it was not strictly necessary to implement any DAO because the information source was accessed using controller layer facades that were being offered by the legacy application.

Finally, *Application Service* can be used to implement the *Business Component* object. This pattern allows the centralisation and the addition of different data acquisition environments by offering a standard service layer. In the authentication use case, the Business Component object includes the process dealing with the legacy application and an external system. This external system performs logging tasks and establishes the different access rights for the users.

4.2 Analysis

One of the main tasks involved into any integration process is the correct definition of the business requirements and the real expectations of a project. In other words, each one of the requirements must be split with a high detail level, decomposing the original problem into smaller ones. Also, it is necessary to identify the functional requirements of the system and to evaluate every possible choice that the technology offers at this moment.

In the analysis stage of the application, we used the guidelines of the *Unified Modeling Language*, and the methodological approach presented in [7]. This approach is based on UML[3] and it establishes four layers representing the essential points of an application that uses web services: structure, transaction, security and workflow.

In the first methodology layer (*Structural Diagram*), we used the *state diagrams* provided by the legacy application, as well as *use case diagrams* that offered a more detailed vision of the actors involved in the process. This way, and following the notation specified in the methodology, we describe the life cycle of the web services: start elements, end elements, transitions and tasks. The starting elements indicate the execution entry points and end elements are the processes where the composite web service terminates. A transition indicates that another task of a web service is being handled next, or that a web service is started or terminated. Finally, a composite task contains inner elements (task or transitions).

In the next layer (*transactional diagram*), the *class diagrams* developed during the integration process were employed, but we added some restrictions over any class likely to have a transactional behaviour, following the OCL (*Object Constraint Language*)[3] notation. In other words, for every service that implies any modification in the state of the legacy application model, its transactional behaviour is established using the OCL. The security and workflow diagrams are currently being specified in the methodology. In our case, some security problems were solved incorporating encryption techniques in SOAP messages, using cypher policies and WSS4J[8](*Web Services Security for Java*). WSS4J is a primarily a Java library that can be used to sign and verify SOAP Messages with WS-Security information. Others problems related to authentication were sorted out by setting out security policies in the web services repository.

4.3 Design

A layer-based design allows the separation of the provider and client applications and to establish different implementation roles, using intermediate translation and delegation components. This is the approach followed by the communication section of legacy applications, which they use to access the web services repository, so full independence between the communication and logic layers is achieved. It is useful to have a translation and delegation layer between subsystems is useful if any change in the communication protocol is needed, so it can be fully reimplemented and replaced without affecting the structure of the application, as it is shown in figure 6.

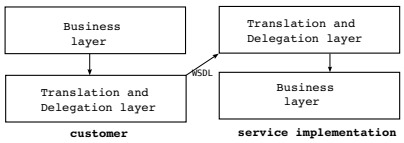

Figure 6: Layers design

In this particular case of study, the design for incorporating new applications followed these steps:

- Each group of related use cases was defined using the *Business Delegate* pattern[2] to provide a suitable data encapsulation layer and to avoid the need of any kind of logic for communication. Using this pattern, the business logic in many client applications can be reused, saving time and resources. Nevertheless, the pattern also hides the complexity involved in the remote communication process from the end users of the services.

  As it is shown in figure 7, the *Factory* pattern was employed to obtain a particular in-

[8]http://www.apache.org/wss4j

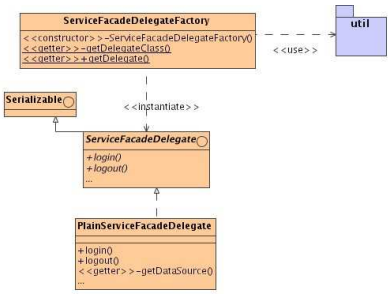

Figure 7: Service customer

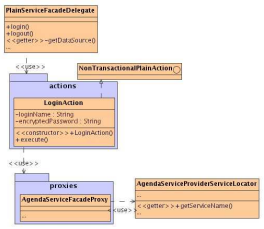

Figure 8: Proxy/Wrapper Service

stance of the facade. The obtained facade *PlainServiceFacadeDelegate* implements the delegation part in the translation and delegation layer, by using an approximation of the SOAP protocol. If it is needed to follow a different approach, for example a CORBA protocol, it would be required to write an implementation of the delegation facade and the translation and delegation layers. These changes will be completely transparent to the client application.
JNDI (Java Naming Directory Interface) is used to get the right type of facade instance, and it is included as a utility package. The facade class, for example, shows the login and logout methods definition implemented in the legacy application.

- The definition of the communication process was based on a web services and SOAP messages over HTTP protocol. This sort of communication allows the definition of a standard and interoperable framework, used for communicating a group of application with their own service repository.
  The main motivation for taking this approach to develop the communication process is to follow groups of standards fully accepted in the business world, making independent communications between platforms easier. This way, a wide range of choices opens for the customers, like the Microsoft .NET platform or a desktop application built with Java J2EE technology.

  The definition of a group of actions that a transactional process can implement is exemplified in the figure 8. For each action modeled we can specify if its logic implies a transactional process, and in that case the recovery mechanisms for an eventual failure are added automatically. In the particular case of the figure 8, the *LoginAction* action class uses a concrete implementation of the design pattern *Session Facade*[2] implemented by *AgendaServiceFacadeProxy*. This pattern is included in the design because it reduced the number of remote calls from the client to the service provider, and it centralises the available functionalities.
  The facade is responsible of starting the communication process with the repository, using another design pattern: *Service Locator*[2]. This pattern, allows to find *transparently* and in an *homogeneous* way business components and services already deployed.

- There is a set of interfaces defined over the web services repository, allowing the different applications to communicate with the legacy applications incorporated to the repository. Using the AXIS[9] project the repository interfaces encompassing the services are defined almost automatically.
  The use of AXIS as a web services development tool comes mainly motivated from the fact that the legacy application was written in the Java language.

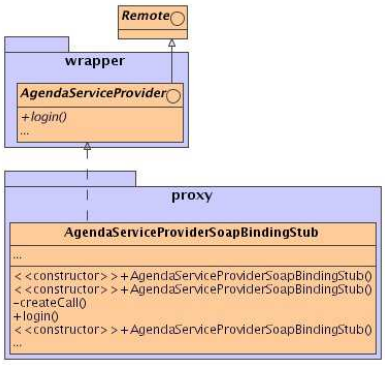

Figure 9: Wrapper Service

[9]http://www.apache.org/axis

Figure 9 shows how some of the defined interfaces are deployed in the repository for enabling the correct communication with the legacy application, and it also provides to different clients a way to locate them inside the repository through the *Service Locator*[2].

- For the implementation of the services defined over the repository (figure 10) the choice was to encapsulate the communication with the legacy application in an EJB[10] container, so some of the problems derived from the transactional and security issues were fixed. The use of this container is motivated by the lack, so far, of a robust implementation inside the SOA architecture that supports security and transactional aspects, in such a way that they are interoperable among different platforms.

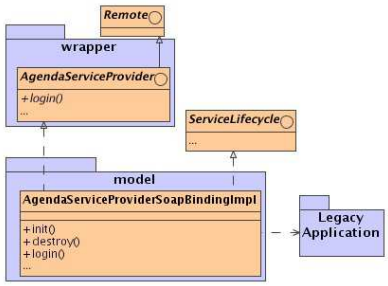

Figure 10: Service implementation

5 Future work

Future work will try to empathise the aspects related to the quality of service of the SOA architecture. A new research line would try to incorporate security based protocols on the standard specification WS-Security[11].
Another research line could try to establish semantic information over deployed web services. For the implementation of the semantic information, we will use the WSDL-S specification, developed by IBM and University of Georgia[12].
This semantic information allows to improve information manager incorporating significative descriptions, classification of resources with explicit semantic, formal models, inference rules and formal verification technics.
Another research line could try to establish the dataflow of deployed services using some formal language specification of business processes like BPEL[13].

6 Conclusions

The work presented in this paper shows how to integrate different legacy applications deployed inside the technological framework of an organisation, and how to build a new environment based on current W3C standards. As well, the solution described here, aims at completely reusing the underlying logic of the legacy applications. This way, we are able to obtain new business processes using a service composition principle. In the particular case shown in this paper, we used the authentication service as a base composition for generating other services that incorporate new functionalities to the original process. Another point, is the complete interoperability of different development platforms inside this framework. As the communication process is based on standards, any client is able to use the services offered by the platform server regardless the technology it is developed with, for example J2EE (Java Enterprise Edition), Microsoft's .NET, or any other platform that supports them.

References:

[1] David A. Chappel. *Using Java in Service-Oriented Architectures. O'Really.* (2002)

[2] Deepak Alur, John Crupi, Dan Malks. *Core J2EE Patterns, Best Practices and Design Strategies.* (2004)

[3] OMG: The unified modeling language, version 2.0. Specification. (2004)

[4] Mark Endrei, Jenny Ang, Ali Arsanjani. *Patterns: Service-Oriented Architecture and Web Services. IBM Redbooks.* (2004)

[5] Paul Allen, *Component-based Development for Enterprise Systems* Cambridge University Press. (1998)

[6] Ramesh Nagappan. *Developing Java Web Services. John Wiley And Sons.* (2003)

[7] Schmit, B.A., Dustdar, S.: *Model-driven development of web service transactions.* In: Proceedings of the 2nd GI-Workshop XML for Business Process Management.

[10] http://java.sun.com/products/ejb/2.0.html
[11] http://www.apache.org/wss4j
[12] http://www.w3.org/Submission/WSDL-S/
[13] http://www-106.ibm.com/developerworks/webservices