# WSNF: Designing a Web Service Notification Framework for Web Services

Bahman Kalali, Paulo Alencar, Don Cowan
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
1-519-8884690
{bkalali,palencar,dcowan}@csg.uwaterloo.ca

## ABSTRACT

In this paper we extend current typical Web service architectures by providing a Web Service Notification Framework (WSNF). Besides the three standard roles found in current frameworks (i.e., the service provider, the service requestor, and the service registry), our approach introduces an additional role that we call the service notifier. The framework is designed in four layers: the Proxy Layer, the Web Server Layer, the Application Notification Server Layer, and the Application Worker Layer. Since the WSNF itself is a service provider, this framework is reflective in the sense that it checks and notifies itself about changes. The framework is documented using design patterns. The set of patterns applied in the framework design includes the following patterns: the singleton, the delegation, the factory method, the observer, the mediator, the notifier, which is a combination of the mediator and the observer, the item description, and the proxy. The notifier pattern is in fact a publisher-subscribe pattern with push semantics. The framework uses a requestor profile to support notifications related to a category of events related to changes, failures, and version control problems of Web services.

**Keywords:** Web Service, Event Notification, Profile, Framework, Design Patterns, Software Change, Failures, Version Control.

## 1. INTRODUCTION

As the Internet becomes the preferred media for businesses transactions, the number of business transactions is increasing every moment and the Internet is flooded with goods and services. For example, buyers enjoy locating the service providers and information sources globally from their desktop computers conveniently at real-time. In many cases, sellers should provide their good and services to the buyers 24 hours a day everyday.

If a business service provider wants to implement a Web service, it first needs to find the abstract part of its description represented in a Web Service Description Language (WSDL). Once the business service provider implements the service modules related to the abstract part, these concrete descriptions are added as a second part of the WSDL file. The mechanism to publish and discover these descriptions is based on Universal Description, Discovery, and Integration (UDDI) technology.

However, UDDI only addresses the problem of finding WSDL files in a static way. If a service provider modifies its underlying Web service implementation and in turn its WSDL file, the UDDI can be useful if the service provider has updated UDDI with a link to the latest version of the WSDL file. This is due to the fact that the UDDI specification (e.g., UDDI Version 2.0) only supports very basic operations such as publishing and subscribing APIs. Service requestors who used old versions of the WSDL and are currently bound to a service provider might fail to operate if this service provider changes its WSDL content by changing its Web service implementation.

Service requestors fail to operate because they were not notified about the change of the WSDL. For example, if a company that is currently providing a service to calculate the cost of shipping a package of certain size decides to change its Web service description (WSDL), then all the partners of this company that use this service must be notified of this change. As another example, once the end-point location of a Web service provider is modified in a WSDL file, the service requestors must be notified about this change. Service providers should not only update their UDDI registries with a link to the latest version of a WSDL file, but they should also send an event notification to service requestors using a notification service about any content and version changes in the WSDL file in a dynamic way.

In this paper, we extend current typical Web service architectures by providing a Web Service Notification Framework (WSNF). Besides the three standard roles found in current frameworks (i.e., service provider, service requestor, and the service registry), our approach introduces an additional role that we call service notifier. Since the WSNF itself is a service provider, this framework is reflective in the sense that it checks and notifies itself about changes. The framework is documented using design patterns.

This paper is organized as follows. In the remainder of this section we provide some motivation for our work and propose a solution to the problems we have identified in current Web service architectures. Section 2 discusses background and related work. Section 3 presents a conceptual architecture of the WSNF. Section 4 presents the design of the proposed WSNF framework and describes its typical use case scenarios, the patterns applied in this design and a hierarchy of Web service events. Section 5 summarizes the paper, concluding with future work.

## 1.1 Problem Statement

Web services go through the usual lifecycle of newer versions and upgrades. Users of a Web service should be aware of these changes. It is assumed that Web services requestors can adapt themselves to WSDL interface changes by downloading the latest WSDL interfaces and adapting their proxy Web services based on the latest WSDL. However, service providers change their WSDL interface without informing service requestors.

The implication of this problem is that aggregation of Web services might fail as a result of interface mismatches. For example, in the aggregation of Web services, a new service provider can be created as an aggregation of service providers Y and X as shown in Figure 1. The operations of service provider Y and service provider X are hidden from the service requestor W. The service requestor W communicates only with service provider Z in this scenario. If a business process model is implemented as shown in Figure 1, then changing the Web service interface of any service provider Y and X without notifying service provider Z can result in a business transaction failure between W and Z. In other words, the service provider Z must automatically be notified about content and version changes of the WSDL of X and Y.

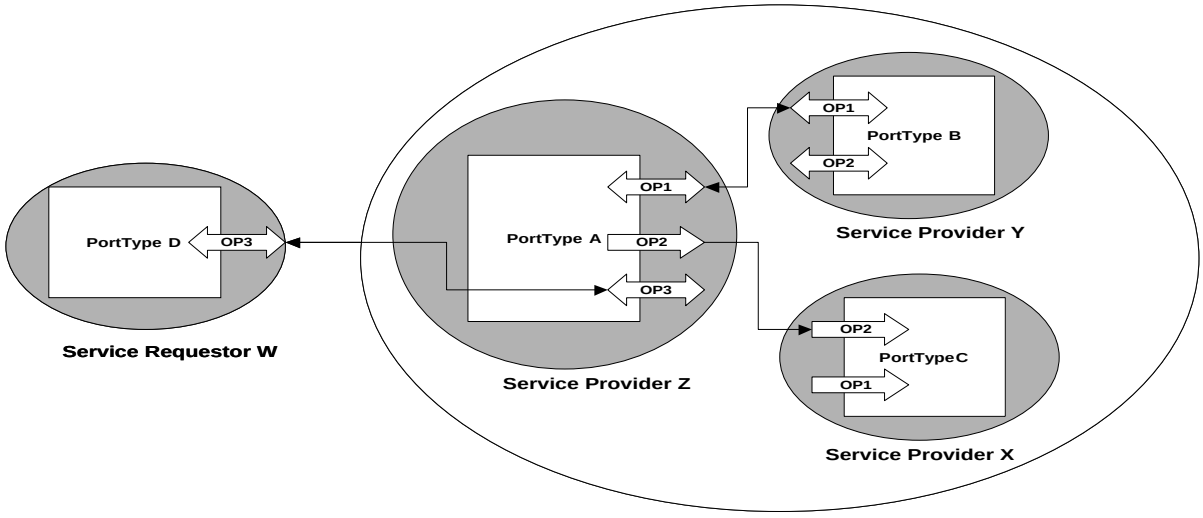



**Figure 1. Web service Aggregation**

A second problem of the current Web service architectures is that it uses the UDDI registry directly to find Web service providers. Further, anyone can have an UDDI account in public registries such as IBM [3], Microsoft [8], HP and register a business without even providing services. For example, we tried to search for a business entity that provides a Currency Exchange Service. We have discovered that out of twelve businesses entitles that provide such a service, only two services were in operation and the other ten were either disabled or their access-point links were broken. In addition, if any changes happen to a WSDL interface, the UDDI will not notify users of a Web service about this change.

A third problem is related to what happens when a Web service becomes disabled. What would happen to service requestors that are currently using this Web service? Who should inform the service requestors about possible alternative WSDL

interfaces that they can use? No mechanism exists to deal with this problem so far in current Web service architectures to our knowledge.

A fourth problem is related to that what happens if a new service provider offers a better service. Who should notify service requestors about better quality services offered by new service providers that offer the same services?

Finally, from a software engineering point of view, solving these problems in an ad-hoc fashion will not lead to a solution that promotes reusability, modularity and extensibility. For this reason, we provide a systematic solution to the previous problems using the concept of object-oriented frameworks [7]. In addition, the lack of appropriate framework documentation has been a significant problem that can prevent frameworks to be effectively (re-)used. However, our framework is documented using design patterns.

## 1.2 Proposed Solution

We propose that a new role should be added to the current Web service architectural framework. We call this role the service notifier, a role that is realized by WSNF services. With this new role, a service provider sends its WSDL interface to be stored in the WSNF's XML database. A service requestor can search for a service provider through the WSNF. A service requestor can also be notified when any changes happen in the interface of a service provider.

Our assumption is that a service provider finds the abstract part of a Web service description and then implements a service. Once a service is implemented, the service provider fills in the second part of the Web service description (i.e., the implementation part). As a result, the service description is then completed and ready to be registered in an UDDI registry and stored in WSNF. An UDDI registry only has a link to a WSDL, but WSNF stores a copy of a Web service description into its native XML database. Each WSDL file is uniquely identified in the native XML database of WSNF.

As another aspect of the communication, a service requestor may search the UDDI registry for a service provider. Once the service requestor finds the Web service provider (i.e., a link to a WSDL file) it will subscribe its profile to the WSNF along with that WSDL. The profile of a notification consists of the type of events that a service requestor wants to be notified about. The WSNF establishes an association between the service requestor, the WSDL and Web service provider. This Web service interface will be stored in the native XML database. If the Web service provider had already stored its WSDL in the WSNF, then the WSNF establishes an association between the Web service requestor, the WSDL and service provider as well as version controlling the WSDL that is sent by the provider. In the following paragraphs we describe some features of the WNSF.

- **Notification of changing Web Service Interface**

If the service provider changes one of its Web service interfaces, it will send a new version of the interface to the WSNF. The new service interface version will be kept through version control along with the old version in the WSNF. Therefore, the WSNF will always keep the latest Web service interface file as well as the old versions. Once the WSNF receives a new version of the Web service interface, it will notify all the Web service requestors about this new interface version.

- **Notification of Inactive Web Service Provider and Replacement**

WSNF has a copy of all the Web service interface providers who have previously stored their WSDL file in its native XML database. WSNF will then periodically bind to each service provider that has its WSDL file in the WSNF. By doing this, WSNF tries to sort out active from inactive services. This management task has two main benefits. First, if a service is inactive, the WSNF will notify all the Web service requestors who have been using this service. Second, it will try to find a replacement for the inactive service and notify all the requestors who were using this inactive service.

- **Notification of a new available Web Service Interface**

Once a new Web service interface is produced and sent by a Web service provider to the WSNF, the WSNF will search its internal native XML database for all the requestors who have used similar service interfaces before. The WSNF will then notify all the requestors of this new Web service interface.

- **Notification of best Web Service Interfaces to Web Service Requestors**

WSNF periodically searches for each service requestor in its local native database in order to find associated WSDL files of the service providers. It then reads the abstract part of WSDL file and searches for *portType* element of the abstract part. Once it finds the *portType* element name, it will search for all service providers who have the implemented part of the WSDL file corresponding to the same *portType* name. The result will be a list of all service providers that implemented the same Web service interface. Based on this list, the WSNF will select one or more service provider interfaces that have the highest number of users. All Web service requestors might be notified about these top-ranked interfaces.

## 2. BACKGROUND AND RELATED WORK

### 2.1 Web Service Architecture

To identify existing problems with current Web service architectures, we have analyzed some Web service frameworks.

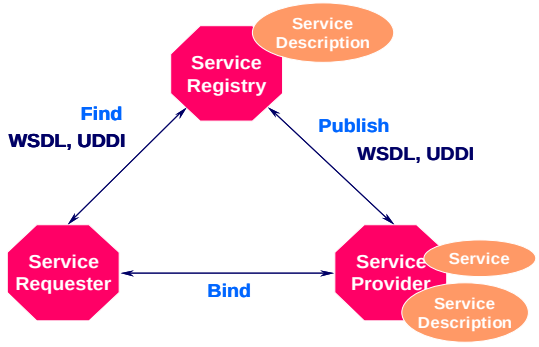



**Figure 2. IBM Web Service Framework**

A typical Web service architecture such as the IBM Web Service Conceptual Architecture [3] is based upon the interactions between three roles: the service provider, the service registry and the service requestor. The interactions between these roles are defined as publishing, finding and binding a Web service. These roles and operations act upon the Web service software module and its Web service description. In a typical scenario, a service provider implements a software module and hosts this module. Then, the service provider defines a Web service description called WSDL and publishes it to a service requestor or service registry such as UDDI registry via an UDDI operator (e.g., the Microsoft UDDI operator). The service requestor uses a find operation to retrieve the service description locally or from the service registry such as UDDI registry and uses the service description to bind with the service provider. Figure 2 illustrates these roles, operations and their interactions [3].

### 2.2 Web Service Enabling Technologies

We have also identified relevant enabling technologies in the area of Web services by analyzing Web service stacks similar to the IBM [3], Microsoft [8], IONA [10], and HP. Figure 3 illustrates the IBM Web Service Stack.

The Web service standard languages and protocol applied at each layer of stack consists of the following elements:

- **SOAP**

SOAP is an extensible XML messaging protocol that provides the foundation for Web services. It is a general-purpose technology for sending messages between endpoints, and may be used for RPC or XML document transfer [1]. SOAP defines the use of XML and HTTP to access services, objects and servers in a platform-independent manner. SOAP messages are presented using XML and can be sent over any transport layer. HTTP is the most common transport layer protocol used to send SOAP messages. SOAP messages can also be transferred over other protocols such as Simple Mail Transport Protocol (SMTP), Java Messaging Service (JMS) and IBM MQSeries. SOAP messages consist of three parts: an envelope that defines a framework for describing what is in a message, a set of encoding rules for expressing instances of application defined data types and a convention for representing remote procedure calls [3].

- **WSDL**

The Web Service Description Language (WSDL) is an XML-based language describes Web services functionality. WSDL describes the operations a Web service can support, parameters each operation accepts and return values. It is through the service description that the service provider communicates all the specifications for invoking the Web service requestor. A basic service description is often divided into two parts: the service interface and the service implementation. A service interface definition is an abstract definition that can be instantiated and referenced by multiple service implementation definitions. Once a WSDL file is instantiated and published into a directory such as UDDI, the other SOAP clients can find the service and bind to it.

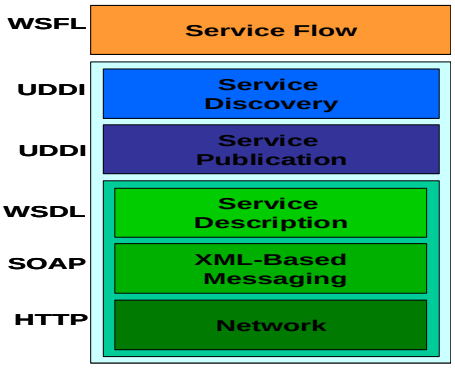



**Figure 3. A Typical Web Service Stack**

One of our main goals of designing the WSNF is to address problems associated with the WSDL files and the content changes of these WSDL files. A WSDL file contains six main XML tags that describe how and where a Web service provider can be accessed. Usually a WSDL file can be automatically created by a tool that is supported by a Web service platform such as Glue [9]. We were motivated to investigate problems related to interface changes, service failures, and version control after we tried to create a simple Web service using Glue. We began by publishing the WSDL file of our service provider to a Web server and by trying to bind to this service from a client machine. To this end, we first retrieved the WSDL file from the Web server and then we created a proxy to the Web service provider, which was running on the server machine, on the client machine. The binding operation between service provider and requestor was successful. However, when we changed the service provider interface at the server machine and, in turn, the WSDL file associated with that Web service and then tried to bind again to the same service provider on the server machine using the old WSDL file, the binding failed. We in fact changed a Java interface name on the server machine, then implemented this interface again and finally published a new WSDL file to the Web server. The binding failure was due to the fact the content of the WSDL file was changed by the service provider, but service requestor had no knowledge about that change. In this scenario, the end user of the service provider was a human and the effects of the interface mismatch were not so serious. However, these effects could be critical if a real-time application requestor used the interface.

- **UDDI**

Universal Description, Discovery, and Integration (UDDI) is a standard that allows information about businesses and services to be electronically published and queried. Published information is stored into one or more UDDI registries, which can be accessed through a Web browser or via SOAP messages. UDDI provides an API for publishing and retrieving information about Web services. UDDI allows us to store and manipulate four main types of entities: *Business*, *Service*, *Binding Template* and *TModel* [2].

A *Business* represents an owner of Web services. It has a name, a unique key, zero or more services, an optional set of contacts, descriptions, and categories as well as identifies. The categories and identifiers can be used to specify attributes such as the business's NAICS (North Standard Products and Services Classification) and the DUNS (Data Universal Numbering System) codes, which can be very useful when performing searches [2].

A *Service* represents a group of one or more Web services. It has a name, a unique key, one *Binding Template* per Web service, an optional set of descriptions and categories and the *Business* that owns the service [2].

A *Binding Template* represents a single Web service, and contains all the information that is needed to locate and invoke the service. It has a unique key, an access point that indicates the URL of the Web service, an optional description, and a *TModel* key for each WSDL type that the Web service implements [2].

A *TModel* represents a concept. This concept can be concrete such as a WSDL file or abstract such as the NAICS categorization scheme. A *TModel* has a name, a unique key, an overview URL that points to data associated with the *TModel* and optional set of descriptions. UDDI uses *TModels* for several different proposes, but their main use is for representing WSDL interface types [2]. By allocating a unique *TModel* to each WSDL type, UDDI allows Web services to be located based on the set of operations that they provide.

- **WSFL**

The Web Services Flow Language is an XML language for the description of Web service composition as part of a business process definition. The WSFL relies and complements existing specifications such as SOAP, WSDL and UDDI.

WSFL considers two types of Web service composition: a Flow Model and a Global Model

In the Flow model, the unit of work in WSFL is an activity and activates are defined as nodes in a linked graph. The *dataLink* and *controlLink* represent the data flow and the control flow between these activities. A *dataLink* specifies that its source activity pass data to the flow engine as part of the process instance context, which in turn has to pass this data to the target activity of the *dataLink*. Data always flows along *controlLinks*. However, the *controlLink* path does not have to be direct and can comprise multiple activities. The *dataLink* enables the specification of a mapping between a source and a target document if necessary [4].

The main goal of WSFL is to enable Web services as implementations for activities of business processes. Each activity is associated with a service provider responsible for the execution of the process step. This relationship defines the association between activities which participate in the control flow and operations offered by the service provider. Activities correspond to nodes in a graph. Thus, an activity can have an input message, an output message, and multiple fault messages. This is how the message flow is specified. Each message can have multiple parts, and each part is further defined in some type system. This is the binding between the message flow and the data flow [4].

In the Global Model, WSFL provides a facility to model interactions between business partners. A global model can specify a new service provider type. This service provider type can be used in turn in a composition through a flow model or another global model. Figure 1 illustrates a Web service global model.

## 3. CONCEPTUAL ARCHITECTURE OF THE WSNF

Our proposed notification service is designed to notify other service requestors about content and version changes of the WSDL files. Other purposes of the notification service include: to test the operation of a service provider, to discover similar service providers when a service provider fails to operate and notify its related service requestors, to notify service requestors about better service providers, to maintain all versions of the WSDL files, to allow service requestors to search for service providers from UDDI registries and to store service requestor profiles (i.e., the Web service profile notification - WSPN). Figure 4 illustrates the conceptual architecture of the framework.

Once a service requestor finds the WSDL of a service provider through the WSNF based on providing a set of search criteria, the WSNF will require a service requestor to subscribe for interested event types to the WSNF. This operation requires a service requestor to provide its Web Service Profile Notification (WSPN) that is stored in the WSNF's native XML database.

If the WSDL of a service provider is not stored in the WSNF's native XML database, then the WSNF will contact the UDDI registry, get a copy of the WSDL and store it into its native XML database.

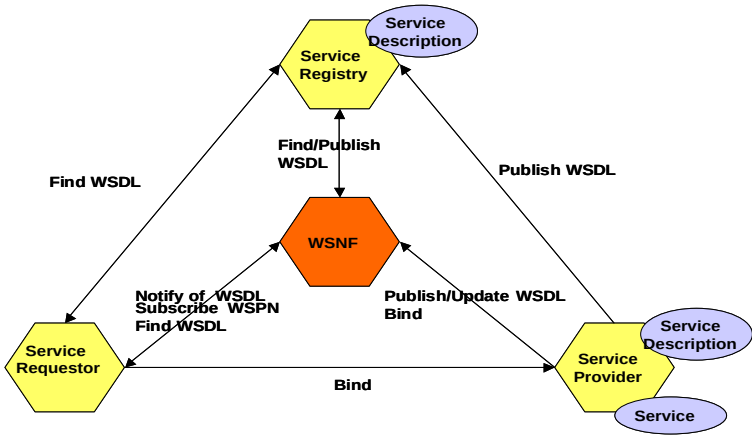



**Figure 4. WSNF Conceptual Architecture**

# 4. DESIGN OF THE WSNF

To provide enhanced flexibility, performance and scalability, our proposed framework is based on a multi-tiered architecture and design patterns. The WSNF is designed into four layers: the Proxy Layer, the Web Server Layer, the Application Notification Server Layer and the Application Worker Layer. Because the WSNF framework addresses mainly notifications related to Web service interfaces and their changes, and not database changes, we did not mention a fifth layer, which is a Database Layer.

Service providers and service requestors that intend to use operations provided by WSNF must first access the WSDL of the WSNF. Once they find this WSDL, they can implement a proxy to the WSNF service in order to use *FindWSDL*, *NewVersionWSDL*, *StoreWSDL*, and *storeWSPN*. At this stage, these are the four basic operations that can be used directly by service providers and requestors.

All message exchange between the Proxy Layer and the event notification system is based on the SOAP messaging protocol. Therefore, the second layer, which we call the Web Server Layer, has *HTTPServer* and *SOAPProcessor* mechanisms. The event notification service of WSPN, which has a push semantic, is designed at the third layer -the Application Notification Server Layer. All the events are published to the event service through a *WManager* object. The *WMangager* object is also in charge of contacting its worker objects for storing the WSDL and the WSPN files, contacting service requestors in case events need to be notified, contacting UDDI registry in case a WSDL file of a service provider cannot be found in the local XML database, and supporting the version control of the WSDL files. All worker objects operate at the Application Worker Layer, which is the fourth layer. Figure 5 illustrates the design of this framework.

## 4.1 Use Case Scenarios

In the following use cases we describe some dynamic behaviors of the WSNF.

- **UseCase#1: Storing the first version of WSDL to WSNF**

By assumption, a service provider who wants to store its WSDL to the WSNF's database first needs to find WSDL of the WSNF provider in a UDDI registry. Once a service provider finds the WSNF's WSDL, the service provider can use it to store its WSDL in the WSNF's native XML database using the *storeWSDL* operation.

*WManager* object delegates the responsibility of storing a WSDL file into a native XML database to the *NativeXMLDatabaseWorker* object. A WSDL file is annotated with an identity ($W_{ij}$, where "i" the identity of a provider and "j" is the identity of an interface). The *WManager* object also creates an *InterfaceProvider* object via the *OODatabaseWorker* object. Each *InterfaceProvider* object is stored into an OO database. Each *InterfaceProvider* object has an identity and a WSDL identity attribute. The identity of a WSDL uniquely identifies the producer of an interface.

- **UseCase#2: Storing subsequent WSDLs to WSNF**

When a new version of the *InterfaceProvider* object is created using the *NewVersionWSDL* operation, the *OODatabaseWorker* object sends a message to the OO database in order to retrieve all the *InterfaceRequstor* objects that have used previous versions of the interface. At this moment, the *OODatabaseWorker* object knows which requestors have used which providers' interfaces. The *OODatabaseWorker* object can then update the version number of the interface that is going to be used by a service requestor. Once the *OODatabaseWorker* object completes its operation, it will inform the *WManager* obect to notify requestors about the new version of the *InterfaceProvider* object. The *WManager* object will contact the *NativeXMLDatabaseWorker* object in order to get the WSPNs of the requestors. Once the *WManager* receives these WSPNs, it will forward them to the *WSDLPublisher* object. The *WSDLPublisher* object will then publish the *InterfaceChangeEvent* object to the Event Service. The *WSDLPublisher* only publishes this event to the Event Service if the requestor had registered for this type of notification in its WSPN.

**Example:**

Assume that the service providers P1 and P2 have previously produced the interfaces (W1, W12, W13) and (W24, W25) interfaces respectively:

$$P1 \rightarrow_{(produced)} (W11, W12, W13)$$

$$P2 \rightarrow_{(produced)} (W24, W25)$$

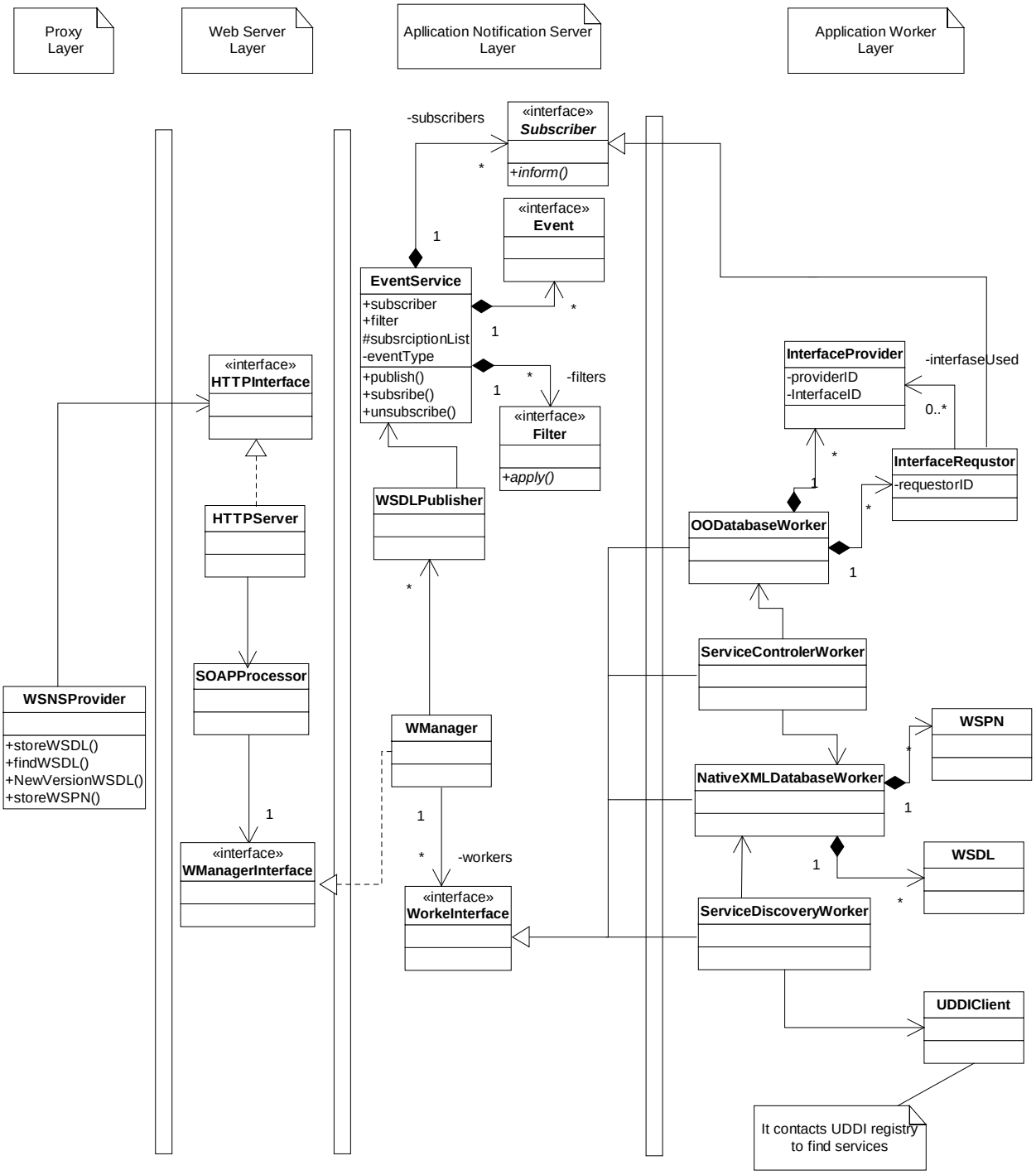



**Figure 5. Design of the WSNF**

Therefore, we have five objects of type *InterfaceProvider* stored into an OO database via the *OODatabaseWorker* object as follows:

PO1 [P1, W11] ; PO2 [P2,W12] ; PO3 [P3,W13] ; PO4 [P4,W24] ; PO5 [P5,W25]

Then, assume that the service requestors R4, R5 and R6 have previously used PO1, PO1 and PO2 respectively:

R4 $\rightarrow_{(used)}$ PO1 [P1, W11]

R5 $\rightarrow_{(used)}$ PO1 [P1, W11]

R6 $\rightarrow_{(used)}$) PO2 [P2, W21]

This means we have three *InterfaceRequstor* objects in the OO Database as follows:

RO1 [R4, PO1 [ P1, W11]]

RO2 [R5, PO1 [ P1, W11]]

RO3 [R6, PO2 [P2, W21]]

Later, let's say a new interface is produced by P1 (i.e., W14). In deed, this is the forth WSDL file that is produced by P1 and submitted to the WSNF.

P1$\rightarrow_{(produced)}$ (W14)

After the WSDL file is stored into the native XML database of the WSNF, an *InterfaceProvider* object is created (i.e., PO6 [P1, W14] and stored into the OO database. Consequently, the *OODatabaseWorker* object sends a query message to the OO database that provides the identity of the service provider (i.e., P1). The *OODatabaseWorker* object will receive the following results:

RO1 [R4, PO1 [P1, W11]]

RO2 [R5, PO1 [P1, W11]]

In this way, the *OODatabaseWorker* object knows that R4 and R5 have used W11, the interface produced by P1. This *OODatabaseWorker* object also knows that P1 has just produced a new interface (i.e., W14).

Thus, the *OODatabaseWorker* object sends a message to the *WManager* object to notify R4 and R5 about W14 that is produced by P1. The *WManager* object will forward the message to the *NativeXMLDatabaseWorker* object in order to get the WSPN of R4 and R5. Once it receives the profiles of R4 (i.e., *WSPN4*) and R5 (i.e., *WSPN5*), it will send these profiles to the *WSDLPublisher* to publish the *InterfaceChangeEvent* to the Event Service. Finally, the Event Service will notify R4 and R5 of this interface change.

Finally, *DatabaseWorker* object updates the attributes of requestor objects with the latest interface version identity.

RO1 [R4, PO1 [P1, W11]] ➔ RO1 [R4, PO1 [P1, W14]]

RO2 [R5, PO1 [ P1, W11]] ➔ RO2 [R5, PO1 [ P1, W14]]

- **UseCase#3: Checking for an inactive service**

The *WManager* object periodically sends a message to its *ServiceControlerWorker* object to check whether a service is active or not. The *ServiceControlerWorker* object contacts *OODatabaseWorker* object to get an *InterfaceProvider* object whose interface identity is the latest one in the OO database. Once the *ServiceControlerWorker* object receives the *InterfaceProvider* object, it will send its identity to the *NativeXMLDatabaseWorker* object in order to get the WSDL of the interface provider. The *NativeXMLDatabaseWorker* object sends back the WSDL file to the *ServiceControlerWorker* object. The *ServiceControlerWorker* object uses the WSDL file to contact the service provider in order to test each operation that is supported in the WSDL file. If all the operations that are being tested terminate successfully, then the *WManager* object will receive a message notifying about this success; otherwise, the *ServiceControlerWorker* object contacts *NativeXMLDatabaseWorker* object in order to get the WSPN of all the requestors that were using this WSDL and pass them back to the *WManager* object. Once the *WManager* object receives the WSPN, it will forward it to *WSDLPublisher* object. The *WSDLPublisher* object will then publish the *ServiceEnactiveEvent* object to Event Service. The *WSDLPublisher* object only publishes this event to the Event Service if the requestor had registered for this type of notification in its WSPN.

## 4.2 Patterns Applied to the Design of WSNF

In designing the WSNF, we applied the following design patterns: the singleton, the delegation, the factory method, the observer, the mediator, the notifier, which is a combination of the mediator and the observer, the item description, and the proxy. Figure 6 illustrates how the interaction between two adjacent layers of the WSNF can be realized through design patterns.

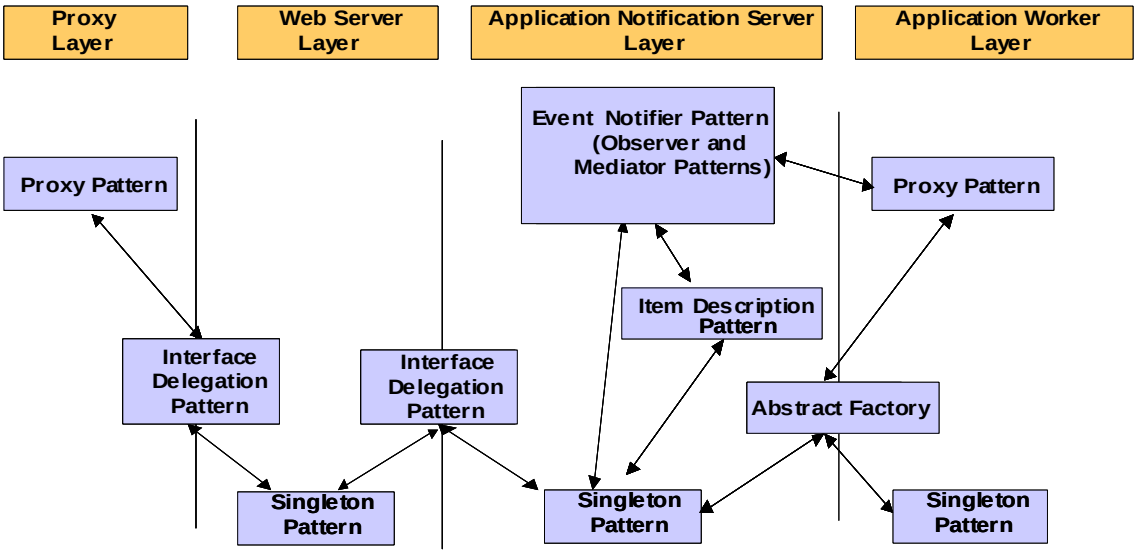



**Figure 6. Design patterns of WSNF and their interaction**

- **Singleton Pattern**

This singleton pattern [6,11] ensures a class only has one instance, and provides a global point of access to it. A singleton class can ensure that no other instance of the class can be created. It can provide a way to access the instance of the class. We have used this pattern at the Web Server Layer, the Application Notification Server Layer and at the Application Worker Layer within the WSNF.

- **Delegation Pattern**

The delegation pattern [6,11] allows one object to delegate responsibility of performing one task to another object through an interface. In designing the WSNF, the delegation pattern is used between the Proxy Layer and the Web Server Layer as well as between the Web Server Layer and the Application Server Layer.

- **Factory Method Pattern**

The factory pattern [6,11] is used when it is necessary to create one of several polymorphism objects until runtime. The factory method pattern lets subclasses to decide which factory to use. The decision can be made by subclasses of a base class. We have used this pattern to realize the interaction between the Application Notification Layer and Application Worker Layer in the WSNF.

- **Observer Pattern**

The observer pattern [5,6,11] allows observer object to watch the subject object. The observer pattern allows the subject and observer to form a publish-subscribe relationship. Through the observer pattern, observer objects can register to receive events from the subject object. When the subject object needs to inform its observer objects of an event, it simply sends the event to each observer object. In designing the notification service of the WSNF, we have used a combination of this pattern with the mediator pattern.

- **Mediator Pattern**

The mediator pattern [5,6,11], which is a behavioral pattern, allows modeling a class whose object at run-time is responsible for controlling and coordinating the interactions of a group of other objects. The mediator pattern helps encapsulate collective of behaviors in a separate mediator object. The mediator pattern promotes a loosely coupled interaction between the objects by keeping these objects from referencing one another explicitly. This pattern along with the observer pattern has been used in the notifier pattern, which is described next [5,6,11].

- **Notifier Pattern**

The notifier pattern [5,6,11] is the combination of the mediator pattern and the observer pattern used to design a publish/subscribe notification service. In the WSNF, the Event Service, as shown in Figure 5, mediates notification so that *WSDLPublisher* objects and *InterfaceRequestor* objects do not need to know about each other. In addition, the Event Service allows us to add or remove subscribers (*InterfaceRequstor* objects) dynamically. In contrast with the observer approach, the addition or removal of subscribers is centralized in the Event Service.

- **Item Description Pattern**

The purpose of this pattern is to put instance variables of a class into a separate class descriptor [6,11]. In designing the Web Service Hierarchy of Events, we have used the item description pattern. In our case, the Event class is an Item and the Property is the ItemDescriptor. Figure 7 illustrates this pattern.

- **Proxy Pattern**

The proxy pattern [6,11] describes how to provide a level of indirection access to an object that resides in another expectable environment. In the Web service architecture, a proxy Web service object acts as an interface between the Service requestor and Service providers. We have applied the proxy design pattern at the Proxy Layer of WSNF since service providers and service requestors should first retrieve the WSDL of the WSNF from a UDDI registry and then create a proxy object to the WSNF service provider.

## 4.3 Web Service Hierarchy of Events

Each service requestor can subscribe to an event type that is going to be published by the service providers to the event service of the WSNF. A natural way of organizing events in object-oriented paradigm is by using inheritance to have a hierarchy of events sharing a common interface or set of attributes on all events, as explained in [5]. For example, the time of occurrence of an event is relevant to all events and therefore should be presented in a base class. Another advantage of such hierarchical organization is for a subscriber to be able to register interest in an event sub-tree and automatically inherit from all the events in that sub-tree [5]. Figure 7 illustrates the hierarchy of events that we adopted for the event type of the WSNF.

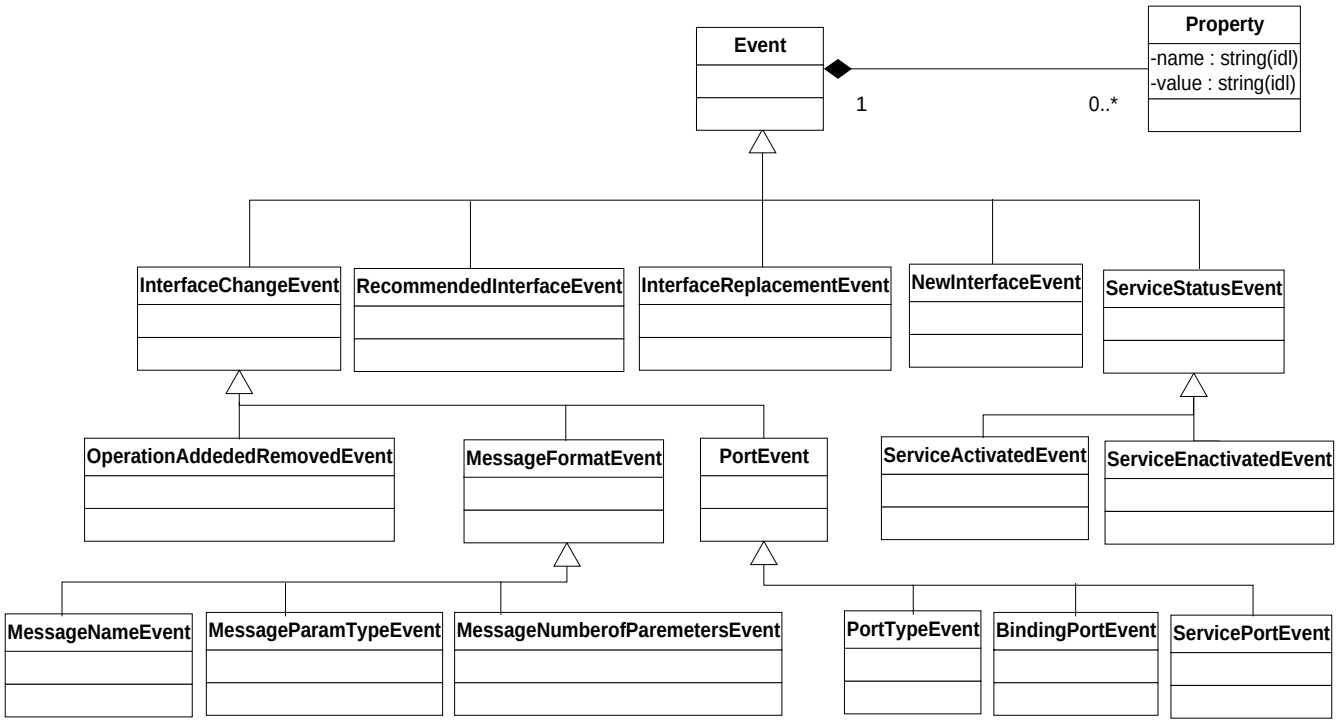



**Figure 7. Web Service Hierarchy of Events**

## 5. CONCLUSIONS AND FUTURE WORK

In this paper we have described an extension of current typical Web service architectures that provides a Web Service Notification Framework (WSNF). Besides the three standard roles found in current frameworks (i.e., the service provider, the service requestor, and the service registry), our approach introduces an additional role that we call the service notifier. We are working on a prototype that incorporates these techniques as a platform for experimentation, demonstration, and requirements elicitation.

The framework is designed in four layers: the Proxy Layer, the Web Server Layer, the Application Notification Server Layer, and the Application Worker Layer. Since the WSNF itself is a service provider, this framework is reflective in the sense that it checks and notifies itself about changes. The framework is documented using design patterns. The set of patterns applied in the framework design includes the following patterns the singleton, the delegation, the factory method, the observer, the mediator, the notifier, which is a combination of the mediator and the observer, the item description, and the proxy. The notifier pattern is in fact a publisher-subscribe pattern with push semantics [5,6,11]. However, other patterns may be included in the framework design and documentation as we experiment and proceed with the framework development. So far, we have only applied object-oriented design patterns in our approach, but we exploring the possibility of having specific service-oriented patterns in our design.

The framework uses a requestor profile to support notifications related to a category of events related to changes, failures, and version control problems of Web services. We are experimenting with these different types of events and their impact in the notification process. In any case, although not complete in any sense, the event set we are currently using constitutes a representative set related to practical circumstances revealed by our conceptual analysis of the Web service application. We are currently working on a more elaborated set of event types and on defining a XML schema for this set.

In summary, by providing a more dynamic perspective to Web service architectures based on the introduction of a service notifier, our approach constitutes both research and practical advances in the design, implementation, and maintenance of Web service applications.

## REFERENCES


[1] W3C Soap Version 1.2, w3c working draft 17, December 2001, "http://www.w3.org/TR/2001/WD-soap12-part0-20011217/"

[2] UDDI. The UDDI Technical White Paper. "http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.doc"

[3] Web Service Conceptual Architecture (WSCA 1.0), IBM Technical White Paper, May 2001, "http://www-3.ibm.com/software/solutions/webservices/pdf/WSCA.pdf"

[4] Web Services Flow Language (WSFL 1.0), IBM Technical White Paper, May 2001, "http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf"

[5] Suchitra Gupta, Jeffrey M. Hartkopf, Suresh Ramaswamy, "*Event Notifier: A Pattern for event Notification*", in Java Report Magazine, July 1998.

[6] Wolfgang Pree, "*Design Patterns for Object-Oriented Software Development*", Addison-Wesley, 1997.

[7] Mohamed E. Fayad, Ralph E. Johnson, "*Domain-Specific Application Frameworks*", Wiley Computer Publishing, 2000.

[8] Web Service Description and Discovery Using UDDI, Microsoft Corporation Technical White Paper, October 2001.

[9] Glass, Graham, "*Web Services – Building Blocks for Distributed Systems*", Prentice Hall, 2002.

[10] IONA Technologies Ltd.: Orbix Programming Guide – Release 3.2, November 1997.

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995.