Extending an Open-Source BPEL Engine with Aspect-Oriented Programming

Alejandro Houspanossian and Mariano Cilia *

UNICEN, Faculty of Sciences
Campus Universitario Tandil, Argentina.
{ahouspan,mcilia}@exa.unicen.edu.ar

Abstract. This work presents an experience report that basically deals with the problematic of extending an existing and evolving software system. The ultimate purpose of the work is to incorporate new functionality into a base system in a maintainable way. Particularly interesting is that the efforts for developing the base system and the efforts for extending it are carried out by two unrelated teams, which have different goals and responsibilities. This document presents our experience working with Aspect-Oriented Programming (AOP) as the primary mechanism for adding functionality to an existing and evolving open-source project.

1 Introduction

There is a growing trend towards open-source software. Open-source is becoming a real option for academic, governmental and even industrial software projects. In contrast to proprietary/closed software, open-source comes with the source code and thus it can be modified and extended as desired, in order to meet the particular requirements of a particular organization (without the need to wait for a new release, a bug-fix patch or to request a new feature to the software provider). In such a context, there is a community (or a set of organizations) that use and extend the open-source software. Because of the fact that software evolves, it is interesting to analyze what happens when new versions of the base software are released. In this work we tackle the problem of the software evolution from the point of view of an organization that is extending an evolving open-source system.

Open source software can be modified (adapted) in order to accommodate to a particular set of requirements of a particular organization (i.e. the system can be *customized* by changing the source code). The question is: what happens with the customized version when a new version of the base system is released? Basically, there are at least three possibilities: the customized version can be replaced by the new system (losing the extensions); the new release can be simply ignored; or, the new release can be customized again in order to meet the particular requirements.

* Also Databases and Distributed Systems Group, TU Darmstadt, Germany.

This work documents the experience of extending an open-source software system in order to add new functionality. Articular to the experience is that the system to be extended is currently in evolution, evolution that we do not control. Additionally, there is a requirement on adapting to new releases of the base software as they appear. That is why we explore a maintainable and flexible approach to the problem of software extension.

In this work we approach the extension of an open-source software system through Aspect-Oriented Programming (AOP) [2]. We propose to introduce the new functionality as aspects of the base system. By doing this, we have extended the open-source system in a flexible and maintainable way (successfully accommodating to several releases of the open-source system).

Certainly, the idea of extending software by using AOP is not a new one [20]. Also, the literature about software extension and evolution is widespread [19], and there are several works focused on the dynamics of open-source software [21] [22]. However, to the best of our knowledge, the experience of using AOP for extending evolving open-source software has not been reported in detail yet.

The context and the analysis of the problem are introduced in sections 2 and 3, respectively. Section 4 presents our approach and section 5 the conclusions. This work is developed as part of the ReFFlow project [15].

2 Context - Problem Domain

This work deals with the the problematic of extending an existing and evolving software system with the purpose of incorporating new functionality. Particularly challenging is that both the system and its extensions are developed by different and uncoordinated teams.

The base system (i.e. the system to be extended) is an open-source *BPEL engine* called ActiveBPEL [6]; its evolution is managed by ActiveBPEL LLC, an open source company.

The functionality we want to offer considers the ability to attach (loosely-coupled) external tools (like auditing, process monitoring, process evolution, etc.) to the ActiveBPEL engine.

The following sub-sections briefly introduce the concepts of BPEL and Publish/Subscribe, which are basic background for this work. Afterward, the intention of the desired extensions is presented to put all this in context.

2.1 About BPEL4WS

BPEL4WS [1] (or BPEL) stands for *Business Process Execution Language for Web Services*. It is the emerging industry standard for describing and executing business processes (see Figure 1). Current version of the standard is v1.1. The standard is now being managed by an OASIS T.C. [8], but it was originally developed by Microsoft, IBM, BEA, SAP and Siebel.

A *BPEL engine* is a workflow engine that executes processes specified with BPEL. There is a big number of implementations of the BPEL standard, a few of them are open-source projects, as it is the case of ActiveBPEL.

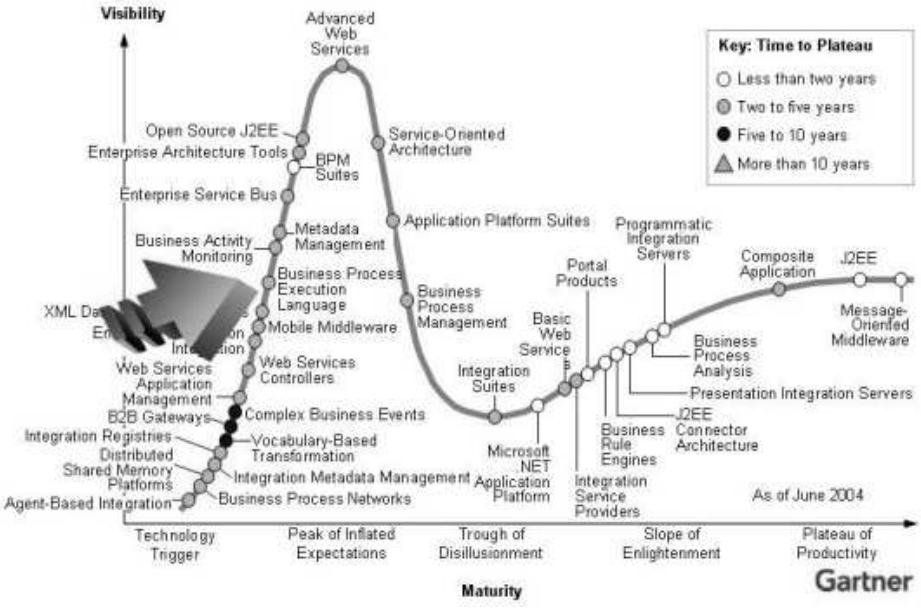

Fig. 1. Application Integration, Application Platform and Architecture Hype Cycle - Gartner 2004

BPEL processes are composed of activities, the execution of a process implies the execution of its activities. A lot of BPEL events occur in the engine at runtime: new processes are deployed, some others are un-deployed, process instances are created, suspended, resumed, and terminated. There are also a lot of events related to the internal execution of the processes (i.e. related to the execution of process activities). Additionally, there are events such as errors and exceptions. In order to facilitate the further analysis we have classified BPEL events in 3 main categories:

Engine Level Events are related to process deployment, instantiation, termination, suspension and resumption.

Process Instance Level Events are related to the execution of a process instance, usually presented in term of process activities.

Errors and Exceptions although these are events undoubtedly inherent to the execution of the processes and the engine, it is convenient to keep them separated.

These are the categories of events that will be exposed, in order to reflect the status of execution of BPEL processes.

2.2 About Publish/Subscribe

Publish/Subscribe (pub/sub) [5] is a mechanism for disseminating data. In a pub/sub based solution, there is no direct relationship between producers and consumers of data. Consumers (also known as subscribers) manifest their interest, in the form of subscriptions, with the pub/sub infrastructure. Data producers (publishers) are not aware of data consumers and they make data available through the pub/sub infrastructure. Actual data dissemination (which first includes the matching process between published data and subscriptions) is also in charge of the pub/sub infrastructure.

The main advantages of this approach include the following: it decouples consumers and producers of data/messages; the number of participants can vary dynamically; and consumers only receive the messages of their interest.

2.3 About our Extension

The functionality we want to offer considers the ability to attach (loosely-coupled) external tools (like auditing, process monitoring, process evolution, etc.) to the engine. These tools need to be aware of what is happening with the execution of processes running within the engine. But we do not want to specify which tools neither to whom events need to be sent since this would imply a closed set of tools. For this reason, we rely on a pub/sub infrastructure to disseminate the status of the execution of BPEL processes. The information will be published in terms of BPEL events[1]. Applying the pub/sub approach in this context is really meaningful: nothing about the nature of the participants (e.g. tools) needs to be assumed, the number of participants can vary at runtime and different participant have different purposes and interests.

In summary, the system to extend is a BPEL engine and the new functionality has to do with the publication (exposition) of the status of the engine by using a pub/sub mechanism. The purpose and benefits of this is clear, to gain visibility into business process execution. Security and privacy issues are outside of the scope of this work. In this work, we do not deal with the underlying mechanism used for the publication neither.

3 Analysis

In order to associate the new functionality to the base system, the ActiveBPEL documentation and its source code must be analyzed. The purpose of the analysis is to identify the places within the source code that need to be extended with our functionality. Additionally to that, and in order to define the actual mechanisms for extending the engine, also the way the engine has evolved in the past must be analyzed.

[1] BPEL events are those events signaled as consequence of the execution of BPEL processes

The next two sub-sections present this analysis. Afterwards, the traditional implementation approach for the extensions in question is introduced. Finally, some concluding remarks are presented.

3.1 ActiveBPEL: Inside View

The ActiveBPEL engine is an open-source project developed in the Java programming language. It was originally developed by a commercial company, Active Endpoints, Inc. (AEI) [7]. AEI offers now commercial products based on the open-source engine. As mentioned before, the evolution of the engine is currently managed by ActiveBPEL, LLC, an open source company (which was created by AEI).

The system is implemented following an Object-Oriented (OO) approach. The project is divided and structured into components according to their functionality (*functional decomposition*). The main functionality of the system includes: process deployment, Web Services handling, process representation and process execution.

The source code is composed of about 700 Java classes. From its inspection arose the following: one of the main classes offers a set of public methods which is used by other components for signaling events (related to process execution). These methods are invoked from all parts of the code and encapsulate the functionality for event notification. The purpose of this mechanism is to provide for remote debugging and console administration. In order to disseminate the events there is an implementation of the observer pattern [14]. It means that new event listeners can be implemented and attached to the engine.

From the analysis of the events and mechanisms used by the engine, we noticed that: the event model we want to expose is not fully supported by the event handling mechanism implemented in the engine. Basically, we need to augment the information provided as part of the existing events and in some cases trigger events that were not considered originally (like those related to errors and exceptions). Although a detailed analysis of this is not the focus of this work, the important thing to remark is that our pretended functionality of event publication could not be solved in the context of a single ActiveBPEL component (i.e. it can not be solved with a simple event listener).

3.2 ActiveBPEL Evolution

We have been following the evolution of ActiveBPEL, even before of its first public release (which appeared in October 2004). Since then, no less than 10 incremental versions have been released. One of the characteristics of open-source software is that releases are made often than in commercial software [22]. Currently, the evolution continues, typically with monthly releases. We would expect major releases encompassing the evolution of the BPEL specification.

Each release is announced with a message into an email-list, a summary of the changes is provided as part of the message. The source code as the binaries

can be downloaded from the project's web site [6] without major inconvenient. Detailed *release notes* are also available there.

The changes introduced in a release are usually related to bug correction and feature enhancement. In the later case the changes are usually encapsulated in one single component, not so in the former. Usually, most of the changes observed are completely unrelated to our extensions and do not involve dramatic changes in the underlying structure of the system. One of the goals of the team that develops the ActiveBPEL engine is to provide, continually, a stable and robust system. In [21], the authors state that open-source projects that aim to provide stable software usually are very conservative against evolutionary and rapid changes.

3.3 Extension by Subclassing

The conventional OOP approach for extending functionality is by applying subclassing. Following this approach and according to our goal we would modify a set of classes, and create others, in order to implement the desired functionality. Basically the code of the extended functionality would be added to the code of the engine (this approach is schematized in Figure 2). The code related to *event publication* is scattered in the original code, involving many classes.

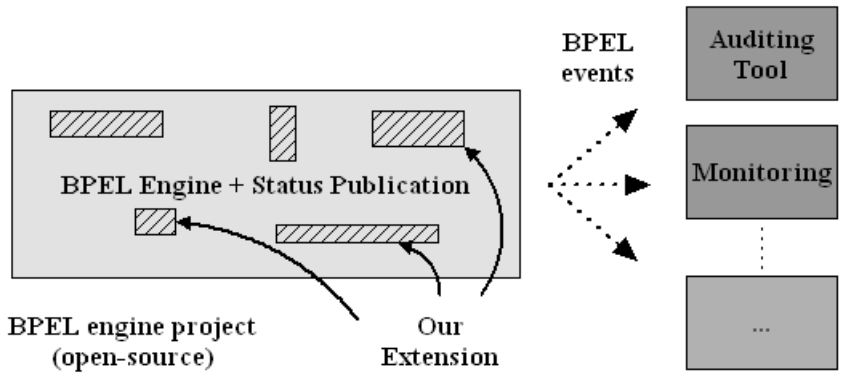

Fig. 2. Extension by Subclassing

A first approach to the solution would be to develop an *event listener*, as defined in the ActiveBPEL project. However, as has been noticed, it will not be enough. The reason is that the ActiveBPEL event model does not provide all the information we need (i.e. we want to support an enhanced event model). Because of this, we should also identify and modify a set of classes in order to signal new events and add information to existing ones.

The number of classes to be modified is potentially high (see section 3.1). The concern of *event publication* can not be solved in the context of a single component (a single event listener, in this case). Additionally, this approach is very dependent to changes in the base system (accommodating to new engine releases would be very difficult). Please remember that the changes in the base system occur regularly and are not necessarily limited to a single component of the engine.

All these factors configure a situation that is not maintainable. In fact, accommodating to a new version of the engine would be a very cumbersome task. This is basically founded on the spread locality of the extensions. This problem should be faced once and again, with every single release of the engine (even if the changes introduced in the release were completely unrelated to our extensions). Applying this approach would probably prevent the use of new versions of the engine.

3.4 Preliminary Conclusions

As has been stated, the ActiveBPEL engine is evolving and new releases appear at a regular basis. We need to implement the extensions in a modular and maintainable way, making it possible to benefit from ActiveBPEL improvements over the time, without excessive maintainability costs. The conventional approach (extending by subclassing) is not as flexible as desired, and is not a viable solution for our problem. Because of this, we started to explore the AOP approach.

4 Extending ActiveBPEL with AOP

In this section we present our approach to extend the ActiveBPEL engine. The basics about AOP is introduced first. The design and implementation of our solution are presented in sub-sections 4.2 and 4.3, respectively. The section concludes with an analysis (basically focused on maintainability issues).

4.1 Aspect-Oriented Programming

AOP [2] is a technology aiming at modularity and maintainability. AOP fosters the goal of separation of concerns. The AOP technology emerged for modularizing crosscutting concerns. Classical examples of crosscutting concerns are (to name just a few): logging, security or exception handling. Crosscutting concerns are concerns (aspects) that can not be encapsulated into single components. On the contrary, the implementation of these concerns crosscut the software structure of a system.

With AOP a crosscutting concerns like *logging* can be encapsulated into a single component and coupled to the original system by defining the relevant points in the code. In fact, using AOP the base system will not even note that

its method invocations are being logged. For an introduction to AOP the reader is referred to [2].

The AOP approach enables the separation of concerns. Different concerns (aspects) of a system can be developed modularly, and incorporated into a base system automatically. Typically, in the infrastructure of an AOP system there is a component (weaver) that is in charge of merging the code of the aspects with the code of the base system (weaving). Important is, the source code of the base system is not directly modified during the procedure. Depending on when the weaving occurs, the tools for AOP are categorized into: static (compile and load time weaving) and dynamic (run time weaving).

AOP is basically founded on the introduction of four major concepts: join points, pointcuts, advices and aspects. *Join Points* are well defined points in the execution flow of the underlying system. *Pointcuts* encapsulate a set of join points, which match a given pattern. *Advices* are pieces of code to execute when a pointcut is reached (actually before, after or instead of a pointcut). *Aspects* are entities that encapsulate the definition of pointcuts and advices.

AspectJ [9][10] is by far the most known tool for AOP. The AspectJ approach to weaving supports compile and recently also load-time weaving. It means that, aspects are associated and disassociated to the base system statically, and not at runtime. It is common knowledge, in the field of AOP, that the dynamic weaving approach gains on flexibility, at the cost of performance. A deepest analysis of this is certainly outside the scope of this work. Because we do not require any dynamic (runtime) features, we have chosen AspectJ as the AOP tool to use. AspectJ can be integrated, as a plugin, to Java development environments (like Eclipse [12] and Borland JBuilder [13]).

4.2 Our Extension

Figure 3 clearly sketches our approach where our extension is completely isolated and is coupled to the engine by using AOP.

As has been stated, the efforts related to the development of the base system and the efforts related to the extension of the base system are orthogonal, independent and are carried on in parallel by different teams. The item that completes the configuration of the problem is the regular appearance of new versions of the base system.

In this context, and after analyzing a conventional approach, we have decided to analyze an AOP-based approach for extending the base system. This decision was supported by our beliefs in the benefits of the *separation of concerns*. The idea was clear: to keep the base system and our extension as separated as possible in order to make the system maintainable.

4.3 Integrating the New Functionality

The main topic of this section is about how the new functionality is integrated into the engine execution flow. We use AOP techniques to intercept the execution

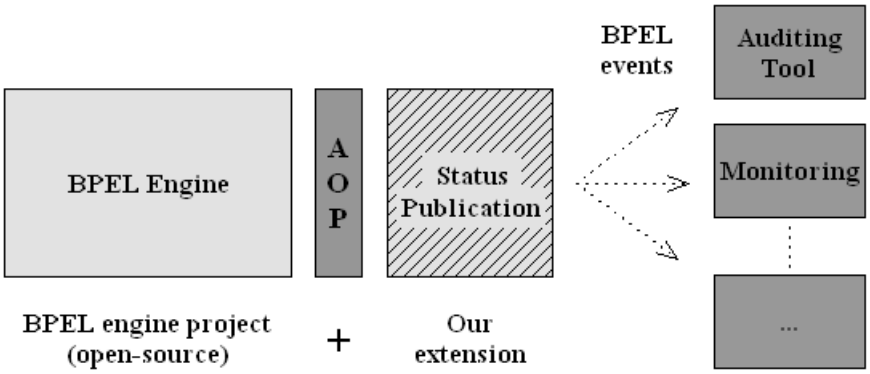

Fig. 3. Extension by AOP

of the engine with the purpose of connecting the new functionality. Namely, we define a set of AOP join points that, in turn, define the connection points between the engine and the extensions.

The first step in order to extend the engine is to detect the join points and define the pointcuts. Such a task requires, of course, knowledge and understanding about the base (underlying) system. From the analysis presented in section 3.1 we already know that the engine currently fires some events to announce status changes. The class *AeBusinessProcessEngine* (package *org.activebpel.rt.bpel.impl*) provides two methods that are good places to add two first pointcuts. One of these methods is:

```
//Class: org.activebpel.rt.bpel.impl.AeBusinessProcessEngine
public void fireEngineEvent(IAeEngineEvent aEvent)
```

This method is executed whenever an engine event (according to ActiveBPEL event model) needs to be signalled.

The pointcut (*fireEvent*) is defined as follows:

```
pointcut fireEvent(IAeEngineEvent event):
    ( call(void AeBusinessProcessEngine.fireEngineEvent(IAeEngineEvent))
    && args(event)
    ) ;
```

The pointcut *fireEvent* comprises all the invocations to the method *fireEngineEvent(IAeEngineEvent)* (of all the instances) of the class *AeBusinessProcessEngine* that do not return a value. We have defined a first connection point for the new functionality. Other connection points can be defined in the same way.

Once the pointcut has been defined, it is time of integrating the extended functionality, which is done in terms of advices. Both pointcuts and advices are defined in aspects. The following code listing presents an aspect called *EventPublishing*:

```
import org.activebpel.rt.bpel.impl.AeBusinessProcessEngine;
import org.activebpel.rt.bpel.IAeEngineEvent;
public aspect EventPublishing {
    ...
    pointcut fireEvent(IAeEngineEvent event):
        ( call(void AeBusinessProcessEngine.fireEngineEvent(IAeEngineEvent))
        && args(event)
        ) ;
    after (IAeEngineEvent event): fireEvent(event) {
        publish(event);
    }
    ...
}
```

The *EventPublishing* aspect will intercept every single method invocation related to event signalling and will involve the extended functionality. The syntax used to define an aspect is quite simple. It is easy to define new pointcuts, advices and aspects. Once detected, other pointcuts can be added in the same way as *fireEvent* to the same aspect *EventPublishing* or to a new one.

As mentioned before, there are some connection points specifically related to exceptions, an example where such a case is managed is presented next. A portion of the original Java code follows:

```
public void AeAxisBase.deployToWebServiceContainer(..)
    throws AeException
{
    ...
    AeException.logError( "Axis deployment failed: ");
    throw new AeException( "Axis deployment failed.", error );
}
```

AspectJ provides *before throwing* and *after throwing* constructs to manage exceptions as it is shown below:

```
pointcut deployment():
    (call (void AeAxisBase.deployToWebServiceContainer(..));

after throwing (AeException e): deployment{
    publish(e);
}
```

These examples show the basic AOP mechanisms we are using in order to extend the base system.

Other Extensions In addition to the the functionality of event publication, and in the context of the ReFFlow Project [15], we are also working on other extensions [16] [17] [18] which are also being introduced to the BPEL engine with AOP. One of these extensions provides the ability to select, at run time, the ports (Web Service instances) that a BPEL process instance uses. Basically, we are using AOP to detect problems during the invocation of Web Services. Upon a port invocation failure, our extended functionality finds and binds a new port to the process instance. In the following we briefly present the *aspect* that introduces this recovery mechanism to the engine. More about the concepts behind this mechanism and its implementation can be found in [18].

The original ActiveBPEL code for managing the invocation of Web Services can be summarized in the following snippet:

```
//Class: org.activebpel.rt.axis.bpel.AeInvokeHandler
public IAeWebServiceResponse handleInvoke(IAeInvoke aInvokeQueueObject){
    ...
    try{
        invokeWS(portData);
    }catch (RemoteException e){
        e.printStackTrace();
        //the running process will be aborted
        ...
    }
    ...
}
```

Originally, whenever a fault is detected during the invocation of a WS, the involved BPEL process is aborted. A summarized version (non-relevant details have been omitted) of the aspect that we use to introduce our functionality is presented next:

```
0  public aspect DynamicRecoveryAspect{
1       pointcut invoke(PortData port):
2               call(private void AeInvokeHandler.invokeWS (port))
3               && args(port);
4
5       void around (PortData port)
6       throws RemoteException: invoke(port){
7           try {
8               proceed(port); //execute the original invokeWS method
9           }catch(RemoteException fault){
10              if ( !dynamicDiscoveryEnabled() ){
11                  throw fault;
12              }
13
14              // if the dynamic discovery feature is enabled,
15              // get a new target port and proceed
16
17              PortData newPort= getNewPort(port);
18              proceed(newPort);
19
20              // The original invokeWS method is executed again,
21              // but this time with a new parameter
22
23         }
24      }
25  }
```

In the *DynamicRecoveryAspect* aspect the *around* clause is used, the meaning of this is that the advice wraps the original join point. By using the *proceed* construct (lines 8 and 18), the computation related to the join point (the *invokeWS* method in the example) can be executed, one or more times, even with different parameters. The *proceed* clause adopts the signature of the join point (returning type, arguments, exceptions). The *DynamicRecoveryAspect* aspect executes the original join point with the original parameter (line 8). If an exception is detected, the running process is not aborted, but a new port is found and bound to it (line 17). Finally, the join point is executed again, but this time with a new parameter. In line 11 (and eventually also in line 18) a RemoteException could be thrown, because of this the declaration *throws RemoteException* in line 6.

In contrast to the pub/sub extension (where the engine was extended without changing its original behavior) here the base system is enhanced to support failure handling.

4.4 Evolution and Maintainability

Because we use AspectJ as the AOP tool, the source code of the base system is not directly modified. The first consequence of this approach is gaining flexibility and the ability to accommodate to new ActiveBPEL engine releases easily.

This approach fosters the separation of concern, in that the engine and the extensions can evolve almost independently.

AspectJ supports *jar weaving*, meaning that a jar file containing the base system can be weaved with the aspects in question without the need of having the source code. Similarly, the aspects can be provided as source files or bytecodes.

We exploit the jar weaving AspectJ feature in order to automate the weaving process. Namely, we defined an ANT task [11] that receives the jar files of the base system and the sources of the aspects and weaves them[2]. In those cases where the engine release does not include any major modification, the engine could be automatically enhanced by executing the following ANT task:

```
<target name="weave" depends="...">
    <!-- weave jar -->
    <iajc outjar="${lib}/ae_rtbpelsvr_woven.jar"
          injars="${lib}/ae_rtbpelsvr.jar">
        <sourceroots>
        <pathelement location="${src}/aspects/refflow/aop"/>
        </sourceroots>
        <classpath>
            <fileset dir="${lib}" includes="**/*.jar"/>
            <pathelement location="c:/tools/aspectj1.2/lib/aspectjtools.jar"/>
            <pathelement location="c:/tools/aspectj1.2/lib/aspectjrt.jar"/>
        </classpath>
    </iajc>
</target>
```

Regarding to accommodating to the evolution of the engine, the most dramatic case would be that in which the structure of the engine is extremely modified from a version to another (as it would be the case of major releases). In such a case, the aspects would need to be adapted accordingly. However, most of the modifications would be restricted to pointcuts adjustment, by detecting again the places to anchor the extension. These are not really bad news since the definition of pointcuts are relatively simple. But most important, the functionality of the extension do not need to be modified. The difficulties at the time of accommodating to a new version of the system will be mostly related to the task of detecting the places where the extensions must be placed, but it is inherent to the problem of extending a software system (and not to the approach followed to extend the system).

Since October 2004, we have worked with several ActiveBPEL releases, and we have not observed dramatic changes in the software internal structure. As a consequence, our extensions evolved along the new versions of the base software without major redefinitions of our pointcuts. The only change that actually had impact over our pointcut definitions was one related to the mechanism used for invoking Web Services. The change also involved the creation of a new java

[2] More about ANT tasks and AspectJ can be found in *The AspectJ Development Environment Guide* [10], chapter 2.

package and redistribution of some classes. This has affected the implementation of the *DynamicRecoveryAspect* aspect (namely the paths of some classes were changed). Anyway, the analysis required to extend the new version of the system was simple. It must be noticed that this kind of changes are announced in the corresponding release notes.

5 Conclusions

We started our work by analyzing the base system to evaluate the impact of our extensions in the source code. As a result, the potential changes were scattered among many classes and therefore the adoption of the traditional OOP approach was considered as inappropriate since the base software evolves continuously.

Thus, we adopted the AOP approach and concentrated our efforts on developing our extensions (dissemination of engine's internal state and failure handling) as aspects and then bound them to the base system by defining pointcuts.

Our extensions, developed as aspects, evolved along various releases of the open-source base software without major changes. It must be highlighted that the base software and our extensions were developed by unrelated developer teams.

We adopted AspectJ and since it supports *jar weaving*, an automatic mechanism to weave our extensions was defined. This can be obviously applied after reading/evaluating the corresponding release notes. We believe that a (semi-) automatic generation of release notes could also help with the detection of possible impact of a new releases on the developed aspects.

This experience report has shown the value of using AOP technology to extend an evolving open-source software system. In this work we have presented the way AOP has helped to manage successfully the continuous (natural) evolution of a base system. We have also shown the benefits of using AOP for supporting a non expected evolution (i.e. our extension) of a system.

6 Acknowledgements

We want to thank Jane Pryor and Claudia Marcos for their comments on the draft version of this document. Additionally, we would like to thank Dimka Karastoyanova for the fruitful discussions related to BPEL, the engine and the extensions.

References

1. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Weerawarana, S.. Business Process Execution Language for Web Services (BPEL4WS Specification v1.1): http://ifr.sap.com/bpel4ws/
2. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. ECOOP97, Finland, June 1997.

3. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. An Overview of AspectJ. ECOOP01, Budapest, Hungary, 2001.
4. Ruzanna Chitchyan and Ian Sommerville. Comparing Dynamic AO Systems. In *Proceedings of the 2004 Dynamic Aspects Workshop*, Lancaster, England, 2004.
5. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114-131, 2003.
6. ActiveBPEL Engine. http://www.activebpel.org/
7. Active Endpoints, Inc. http://www.active-endpoints.com/
8. OASIS. http://www.oasis-open.org/
9. AspectJ Home Page. http://www.aspectj.org
10. The AspectJ Development Environment Guide http://www.eclipse.org/aspectj/doc/released/devguide/
11. ANT Home Page. http://ant.apache.org/
12. Eclipse Home Page. http://www.eclipse.org
13. Borland JBuilder Home Page: http://www.borland.com/jbuilder/
14. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 1995. ISBN 0201633612.
15. The ReFFlow Project. www.dvs1.informatik.tu-darmstadt. de/research/refflow/
16. Karastoyanova, D., Buchmann, A.: Extending Web Service Flow Models to Provide for Adaptability. In *Proceedings of OOPSLA '04 Workshop on "Best Practices and Methodologies in Service-oriented Architectures: Paving the Way to Web-services Success"*, Vancouver, Canada, 2004.
17. Karastoyanova, D., Buchmann, A.: "Development Life Cycle of Web Service-Based Business Processes. Enabling Dynamic Invocation of Web Services at Run Time". In *Proc. of the Second International Workshop on Web Services: Modelling, Architecture and Infrastructure (WSMAI-2004)*, Porto, Portugal, April 2004.
18. Karastoyanova, D., Houspanossian, A., Cilia, M., Leymann, F., Buchmann, A.: Extending BPEL for Run Time Adaptability. *To appear in Proc. of the 9th International Enterprise Distributed Object Computing Conference (EDOC 2005)*, Enschede, The Netherlands, September 2005.
19. Lehman, M.M., Ramil, J.F.: An Approach to a Theory of Software Evolution. *In Proc. 2001 Intern. Workshop on Principles of Software Evolution*, 2001.
20. Walter Cazzola, Shigeru Chiba, Gunter Saake (Eds.): RAM-SE'04-ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Proceedings, Oslo, June 15, 2004. Fakultät für Informatik, Universität Magdeburg 2004.
21. Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, Yunwen Ye: Evolution Patterns of Open-Source Software Systems and Communities. *International Workshop on Principles of Software Evolution 2002 (IWPSE2002)*, Orlando, FL, May 19-20, 2002
22. Robins, J.:Adopting OSS Methods by Adopting OSS Tools. *In 2nd Workshop on Open Source Software Engineering, held at ICSE 2002*, Orlando, FL, USA, May 2002.