Author(s): Falcarin, Paolo; Goix, Laurent Walter
Title: An Aspect-Oriented Approach for Dynamic Monitoring of a Service Logic Execution Environment
Year of publication: 2006
Citation: Falcarin, P. and Goix, L.W. (2006) 'An Aspect-Oriented Approach for Dynamic Monitoring of a Service Logic Execution Environment', in *IEC Annual Review of Communications*, vol. 59, Chicago: International Engineering Consortium (IEC) pp.237-242.
Publisher link: http://www.iec.org

An Aspect-Oriented Approach for Dynamic Monitoring of a Service Logic Execution Environment

Paolo Falcarin[1], Laurent Walter Goix [2]
[1] *Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy*
paolo.falcarin@polito.it
[2] *Telecom Italia, Torino, Italy*
laurentwalter.goix@telecomitalia.it

Abstract

Service creation environments play a relevant role in new telecom applications because they enable openness and programmability by offering frameworks for the development of value added services.

The JAIN SLEE specification defines a Java framework for executing event-based distributed services made up of components, called Service Building Blocks.

In such a complex architecture, monitoring is an indispensable technique to test the dynamic behavior of a system, debug the code, gather usage statistics or measure the quality of service.

Program instrumentation is needed to insert monitoring code into the system to be monitored, which is typically a manual and time-consuming task.

This paper describes a language-based approach to automate program instrumentation and monitoring management using a dynamic Aspect Oriented Programming (AOP) framework.

The basic notions of AOP and the use of the JBoss AOP framework features are described, in order to allow a highly modular and easily configurable implementation of reusable monitoring code. Using an Eclipse-based system administration console, it is possible to manage remotely the dynamic deployment and update of monitoring code in a service deployed on a JAIN-SLEE container.

1. Introduction

A service creation environment addresses the main feature of service programmability. This means the ability of implementing new services faster, with higher software reuse and rapid configuration. Another important issue is the capability to offer to users the same service everywhere, providing a seamless access from different terminals (mobile phones, SIP-phones [6], UMTS phones…). Among different service creation technologies [1, 2], the JAIN APIs for Integrated Networks bring service portability, convergence, and secure network access to telephony and data networks.

By providing a new level of abstraction and associated Java interfaces for service creation across Public Switched Telephone Network (PSTN), packet or wireless networks, JAIN technology enables the integration of Internet and telecommunication networks.

Moreover, by allowing Java applications to access resources within the network, the JAIN idea is shifting the communications market from many proprietary closed systems to a single network architecture where services can be rapidly created and deployed.

The JAIN Service Logic Execution Environment (SLEE) [14, 15] is an integral part of the set of JAIN API's. It is the logic and execution environment in which communication applications are deployed to use the different network resources defined by the other JAIN API's. Basically, the JAIN SLEE specification defines interfaces and requirements for communication applications relying on JAIN standards.

2. JAIN SLEE

JAIN SLEE is a standard architecture defining an environment targeted at communication-based applications.

The specification includes a component model for structuring the application logic of communications applications as a set of object-oriented components, and for arranging these components into higher level and more complicated services. The programming language used by application developers in JAIN SLEE is Java.

The SLEE architecture also defines the contract between these components and the container that will host these components at run-time. The SLEE specification supports the development of highly available and scalable distributed SLEE specification-compliant Application Servers, even if it does not suggest any particular implementation strategy. More importantly, applications may be written once, and then deployed on any application environment that implements the SLEE specification. The system administrator of a JAIN SLEE controls the lifecycle (including deployment, un-deployment and on-line upgrade) of a service.

The lifecycle management is achieved through the use of the standard management interfaces provided by a compliant JAIN SLEE, typically reusing Java Management Extensions (JMX) techniques [16]. A service includes meta-information that describes it, for example its name, vendor and version, and any program code that is associated to it. The program code can include Java classes and Service Building Blocks.

The atomic element defined by JAIN SLEE is the Service Building Block (SBB). An SBB is a software component that sends and receives events and performs computations based on the receipt of events and its current state.

SBBs are stateful components since they can remember the results of previous computations and those results can be applied in additional computations. SBBs perform logic based on the receipt of events. Events are used to represent occurrences of importance that may occur at arbitrary points in time. For example the act of an external system delegating to the SLEE a call setup may occur at any point in time and is therefore easily modeled as an event.

An SBB definition includes meta-information that describes it (e.g. its name, vendor and version), the list of events that it can receive, and Java classes that provide the logic of the SBB itself.

An event represents an occurrence that may require application processing. It contains information that describes the occurrence, such as the source of the event. An event may asynchronously originate from a number of different sources, for example an external resource such as a communications protocol stack, from the SLEE itself, or from application components within the SLEE.

Resources are external entities that interact with other systems outside of the SLEE, such as network elements (Messaging Server, SIP Server...). A Resource Adaptor adapts the particular interfaces and requirements of a resource into the interfaces and requirements of the JAIN SLEE.

StarSLEE [7] is a prototype event-based execution engine for telecommunication applications inspired from JAIN SLEE specification that reuse the concept of SBBs and Resource Adaptors; in this work we applied dynamic AOP techniques for managing runtime monitoring on StarSLEE.

3. Aspect Oriented Programming

Aspect-Oriented Programming (AOP) [5] is a new programming paradigm extending object-oriented software development. The main purpose of AOP is separation of concerns, developed orthogonally from the main functionality of a software system.

While the term 'concern' represents whichever specific requirement to be implemented in a software system, cross-cutting concerns are requirements whose implementation is difficult to modularize, e.g. security, persistence, logging, etc…because the code involved by these concerns is scattered throughout several classes in an object-oriented application (see figure 1).

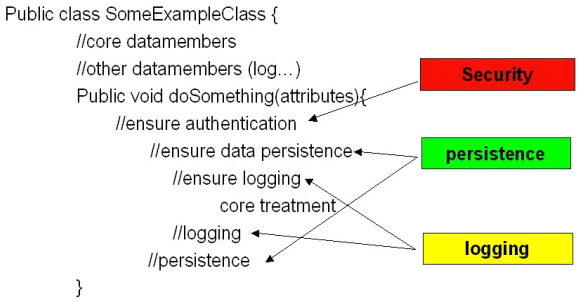

Figure 1. Example of crosscutting concerns

Instead, with AOP, developers can remove scattered code related to crosscutting concerns from classes and placing them into first-class elements called aspects. In this way the original classes are no more responsible of managing functionalities not related to their core functionality. A direct consequence of aspect use is that less code needs to be written, code that would otherwise be spread throughout the system can now be modularized in one place.

In figure 2 it is easy to see that now the *doSomething()* method contains only business related code.

It means that now we are able to completely separate crosscutting concerns from business ones, thus, at the implementation level, by keeping aspects separate from the target application methods they interact with, the application source code is easier to understand.

Therefore, with this new structure, if it is necessary to modify the logging-related code, this can be changed in one place and not in each class.

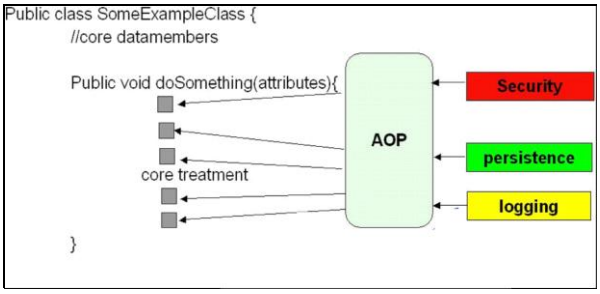

Figure 2. Crosscutting concerns in AOP

AOP methodology is implemented by different platforms, like AspectJ [10], AspectWerkz [11], AspectC++ [8], and JBoss-AOP [12] which are, among others, the most stable and widespread AOP frameworks;

all these tools rely on their own join-point model, which defines the points along the execution of a program that can be possibly addressed by an aspect.

Thus, AOP involves a compiling process, called weaving, for the actual insertion of aspect code into pre-existing application source code or byte code. Weaving can occur at compile-time, load-time, and run-time.

In AOP terminology an aspect is composed by a set of pointcuts and advices. The term 'advice' represents the implementation of a crosscutting concern, i.e. additional code to be executed in particular points of the application code.

Advices can be of three types: before, after and around; a *before* advice is executed before the join-point (e.g. before method execution), an after advice is executed after the join-point (e.g. after returning of method execution), and an around advice is executed instead of a join-point (e.g. it replaces the method body implementation).

AOP also involves means of identification of the join points to be affected by an aspect. The AOP term 'pointcut' implicitly defines at which points in the dynamic execution of the program (at which join-points) extra code should be inserted. Pointcuts can describe sets of join points by specifying, for example, the objects and methods to be considered, or a specific method call or execution. Moreover, wildcards and logical operators can be used to combine pointcuts in more complex ones, identifying a wider set of join-points.

The term "Dynamic AOP" is attributed to platforms allowing the insertion (and withdrawal) of aspects at runtime: this means that an aspect can be dynamically and remotely inserted (and then further changed) without stopping the application.

Moreover, AOP has been used to instrument source code and collect dynamic information about a system.

Putting together these features, in this work we have implemented a remote system monitoring and logging framework which is able to insert (and then change at run-time) monitoring code in a JAIN SLEE distributed application.

4. The monitoring Aspect in JBoss-AOP

JBoss-AOP is a Java framework for dynamic AOP that can be run within or outside of JBoss Enterprise Application Server. For example, JBoss-AOP allows you intercepting a method call and transparently insert additional code (aspect) when the method is invoked.

All AOP constructs are defined as pure Java classes and bound to the application code via an XML [5] file containing the pointcut definitions.

This XML file (jboss-aop.xml) is read at process start-up by the JBoss container which defines the maximum superset of join-points that can be defined in application code, i.e. where an interception could occur at runtime.

In figure 3 there is an example of XML file which prepares the body of method "method" of class "Foo".

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
<prepare
   expr="execution(public void Foo->method())"/>
</aop>
```

Figure 3. An Example of jboss-aop.xml file

After that, during application loading, JBoss prepares application classes, instrumenting their bytecode with the addition of "hooks", i.e. invocations to aspect's advice.

If joint-points are not instrumented in application code at first deployment, it will be impossible to bind them to a new interceptor at runtime. Such a mechanism reduces overhead on the process at runtime by limiting checks on joint points and enhances security avoiding any code to be intercepted.

The JBoss-AOP framework is based on invocation objects implementing the *Invocation* interface.

Invocation objects are the representation of join points at runtime. They contain runtime information about their join points and also drive the flow of aspects.

There are different invocation objects:

- *MethodInvocation* is created and used when a method is intercepted.
- *ConstructorInvocation* is created and used when a constructor is intercepted.
- *FieldInvocation* is an abstract base class that encapsulates field access.
- *FieldReadInvocation*, extends *FieldInvocation* and is created when a field is read.
- *FieldWriteInvocation* extends *FieldInvocation*, and is created when a field is written to.
- *MethodCalledByMethod* is allocated when using "call" pointcut expressions. This particular class encapsulates a method that is calling another method so that you can access the caller and callee.

Similarly, *MethodCalledByConstructor* and *ConstructorCalledByMethod* are allocated respectively when a constructor is calling a method and vice-versa.

In JBoss-AOP an aspect is a class implementing the *Interceptor* interface. This class must implement two methods: *getName()*, which returns the name of the aspect interceptor, and *invoke()*, which represents the advice method and provides the invocation object as input.

The most important method of the *Invocation* interface is the *invokeNext()*. Calling the *invokeNext()* method means executing the intercepted method or constructor (or other) and returns the return value of that method if any.

Not calling that method will not execute the intercepted code meaning overwriting it and interfering with the normal execution of the method.

```
public class InterceptorExample implements Interceptor {

    public String getName() {
        return "InterceptorExample";
    }

    public Object invoke(Invocation invocation) throws Throwable {
        System.out.println("Entering anything");
        return invocation.invokeNext(); // proceed to next advice or actual call
    }

}
```

Figure 4. Example of Interceptor

As exemplified above, one can easy see multiple usages of interceptors by acting either as *before*, *after* or *around advice* based on when the *invokeNext()* method is called.

Dynamic AOP hence becomes a powerful tool for Application Servers such as SLEE, enabling many monitoring applications such as testing, logging, Service Level Tracing, statistics gathering, bug fixing, etc.

5. Aspect Deployment on the Service Bus

The Service Bus [17] is an event-based distributed middleware that allows for runtime deployment and monitoring of service-level information over heterogeneous resources of a communication network. In particular, it allows deploying aspects over several running SLEE containers at the same time, using a new type extension called "Aspect". This overall mechanism enables the local development of aspects and their remote deployment on to the network.

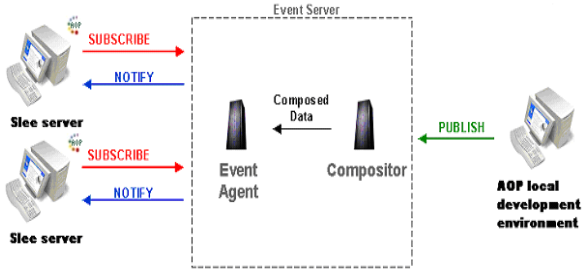

Figure 5. The SLEE service bus

The picture in the previous figure describes the mechanisms used within the Service Bus for SLEE servers to subscribe to AOP-related information while an AOP deployment console publishes the command to deploy or undeploy aspects.

The steps to follow for deploying an aspect through the Service Bus are:

1) Write, compile the interceptor and put it into a JAR.
2) Deploy the JAR file on to an HTTP server.
3) Send a publish message through the Service Bus, indicating the target SLEE(s) and specifying the AOP-related information as follows:
   a. *Name*: logical name, corresponding to a primary key, i.e. a unique identifier between an interceptor and a pointcut.
   b. *Pointcut*: the pointcut used to intercept the classes to be monitored.
   c. *Interceptor name*: the fully-qualified name of the interceptor class.
   d. *URL*: the HTTP URL of the JAR file containing the interceptor class.

Below is an example of such AOP-related information published through the Service Bus to deploy an aspect on a target SLEE.

```
<name>myAspect</name>
<pointcut>
    execution(void myPackage.Aclass->method(..))
</pointcut>
<interceptor>test.InterceptorExample</interceptor>
<jar>http://anySite/interceptionExample.jar</jar>
```

Figure 6. Aspect configuration in SLEE

The following figures respectively display the successful installation (figure 7) and the undeploy command of an aspect on a SLEE (figure 8) in the Eclipse-based Service Bus management console.

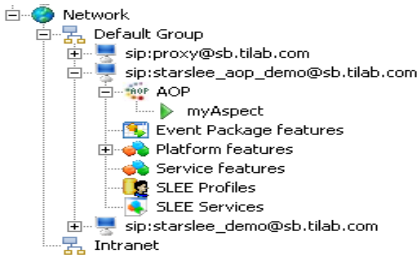

Figure 7. Monitoring Aspect deployment

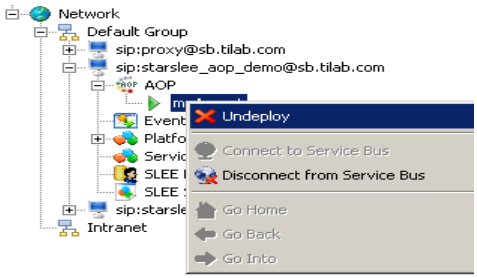

Figure 8. Monitoring Aspect undeployment

6. Discussion and Future Work

In JAIN SLEE architecture, monitoring [13] is an indispensable technique to test the behavior of a system, debug the code, obtain usage statistics or measure the quality of service.

Program instrumentation, which is typically a manual and time-consuming task, is often used to insert monitoring code into the system to be monitored before service deployment.

AOP has been already used to automate code instrumentation before deployment [4] but, with this approach, it is not possible to dynamically change monitoring code after service deployment.

In our approach, using dynamic AOP for managing monitoring tasks has revealed several advantages: a monitoring aspect is developed once and then deployed to different containers, using the Eclipse-based administrator console. The adaptation to different classes is eased by the power and flexibility of language-based constructs (pointcuts).

The added value of our approach is the use of a dynamic AOP framework. This allows the fast deployment of new monitoring aspects on different SLEE containers already running to dynamically modify their behavior.

Future enhancements of the monitoring platform will mainly target a friendly usage of this technology by creating a library of aspects of interest for SLEE containers, SBBs and services, and the addition of wizards for handling aspect templates. Aspect templates parameters could be easily instantiated to ad-hoc aspects for the system to be monitored.

7. References

[1] Licciardi, C.A., Falcarin, P., Analysis of NGN service creation technologies. In *Annual Review of Communications* vol. 57, IEC, December 2003.

[2] Licciardi, C.A., Falcarin, P., Next Generation Networks: The services offering standpoint. In *Comprehensive Report on IP services*, Special Issue of the International Engineering Consortium, October 2002.

[3] Glitho, R.H., Khendek, F., De Marco, A., Creating Value Added Services in Internet Telephony: An Overview and a Case Study on a High-Level Service Creation Environment. In *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Review*, Vol. 33, n. 4, November 2003.

[4] Mahrenholz, D., Spinczyk, O., and Schroeder-Preikschat, W. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, Washington DC, USA, April 2002.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming" Proc. of 11th European Conference Object-Oriented Programming, 1997, pp. 220-242.

[6] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, SIP: Session Initiation Protocol, RFC 3261, June 2002.

[7] A. Baravaglio, C.A. Licciardi, C. Venezia. Web Service Applicability in Telecommunications Service Platforms. In *Proc. of the International Conference on Next Generation Web Services Practices*, Seoul, Korea, August 2005

[8] AspectC++ project. On-line at http://www.aspectc.org

[9] XML (eXtensible Mark-up Language) specification. On-line at http://www.w3.org/XML/

[10] AspectJ project. On-line at http://eclipse.org/aspectj

[11] AspectWerkz project. On-line at http://aspectwerkz.codehaus.org

[12] JBoss AOP framework. On-line at http://www.jboss.org/developers/projects/jboss/aop

[13] Mahrenholz, D.: Minimal invasive monitoring. In Proceedings of Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, May 2001 pp. 251 – 258.

[14] JAIN SLEE API Specification, Java Specification Request (JSR) 22, 1999. On-line at http://www.jcp.org/jsr/detail/22.jsp

[15] JAIN SLEE (JSLEE) v1.1, Java Specification Request (JSR) 240, 2004. On-line at http://www.jcp.org/jsr/detail/240.jsp

[16] Java Management Extensions (JMX), On-line at http://java.sun.com/products/JavaManagement/

[17] G. Valetto, L.W. Goix, G. Delaire. Towards Service Awareness and Autonomic Features in a SIP-enabled Network. In *Proc. of the Workshop on Autonomic Communication (WAC2005)*, Athens, Greece, October 2005.