A^{2}OP: Aspect+Agent Oriented Programming

Z. Obrenovic

Abstract:
In this paper we present a hybrid approach to software development based on aspect-oriented programming (AOP) and software agents. Joining aspect-oriented and agent-oriented software development paradigms could provide more powerful approach to development of complex software systems. While AOP can significantly improve the modularity of programs, usage of agents can improve the flexibility and robustness. We propose a framework where aspect's implementation is not fixed in the code, but is instead implemented by software agents. We have defined a generic mechanism that enables the agents to access functionality of the system over aspects' join points. Aspects behaviour is defined by agents that could be replaced or reorganized, so the programs behavior can be both added and removed at runtime by adding or removing agents. We also show how proposed approach has been implemented using two popular Java based AOP and agent technologies, AspectJ and Aglets.

1. Introduction

Aspect-oriented programming (AOP) is a promising new programming technique based on the idea that computer systems are better programmed by separately and explicitly specifying the different concerns or aspects of a system [1]. We then weave or compose aspects together into a program using mechanisms in the underlying AOP environment. AOP has received lots of attention in last few years. Special issues of the Communications of the ACM in October 2001, and IEEE Software in January 2006 cover this issue while International Conference of Aspect-Oriented Software Development conference has been organized annually since 2002.

AOP has opened many novel possibilities for improving organization and modularity of program systems. Many researchers also discussed how AOP could provide programmers with the abilities to modify the default behavior of a program system, further increasing flexibility of programs. In view of that, in this paper we present a hybrid approach to software development based on AOP and software agents. We propose a framework where aspect's implementation is not fixed in the code, but is instead implemented by software agents. While AOP can significantly improve the modularity of programs, usage of agents can improve its flexibility and robustness.

Many researchers have explored usage of aspect-oriented approach to deal with crosscutting concerns in multi-agents systems, but usage of agent to improve flexibility of AOP has received less attention [2, 3]. Others also emphasis that analyzing, designing, and implementing complex software systems as a collection of interacting, autonomous agents affords software engineers a number of significant advantages over contemporary methods, but these systems have to solve many practical problems, such as integration with legacy components, which are better addressed with AOP [4, 5]. Joining aspect-

oriented and agent-oriented software development paradigms could provide more powerful hibrid approach to development of complex software systems.

In next section we present some of the existing solutions. Then we present the basic idea of our solution. After that, we show how proposed approach has been implemented using AspectJ and Aglets, two popular Java-based AOP and agent technologies. In the end, we give conclusion and discussion.

2. Existing solution

The basic idea of our approach is to increase the flexibility of programs, enabling runtime change and adaptation of its implementation [6]. Several other technologies can also be used to provide similar results, but with various drawbacks.

For example, one approach is to design programs so that the main functionality is implemented on a remote server, and the program only consists of interfaces or aspects that call these remote services. It is possible to use various technologies to achieve this, such as remote procedure call (RPC), remote method invocation (RMI), or Web services [7]. In this way it is easy to change the implementation, as it is centralized, and changing it immediately affects all the instances of the programs. However, this approach introduces various problems. There is network latency in service calls, it is hard to personalize and adapt the services, complex session management is usually required, while applications are less fault-tolerant as problems on the server or on the network could disable many applications. In order to address complexity caused by network communication, Nishizawa et al proposed an interesting concept of a remote pointcut that enables developers to modularize crosscutting concerns distributed on multiple hosts [8].

Sullivan discussed benefits of using reflection and metaobject protocols (MOP) in AOP [9]. Computational reflection enables a program to access its internal structure and behavior, and also to modify its behavior by programmatically manipulating the structure [10]. In this way, it is possible to implement and change aspect code without a static compilation phase, making aspect behavior more robust and adaptive, while the places where this instrumentation can happen are not restricted by points defined by a static compilation process. However, MOP has several practical problems. Most of existing widely used programming languages support reflection and metaobject protocols in a limited amount. Java’s reflection is "read-only", a program can query the methods of a class, but it cannot dynamically change the implementation of methods of a class. Full reflection allows modification of any program metainformation. Other languages have even less support for reflection than Java. There is also research in reflective middleware that adds flexibility to middleware by exploiting the concept of the meta-object protocol [11, 12]. Several other AOP approaches apply aspects at runtime. Popovici et al proposed dynamic weaving for aspect oriented programming allowing aspects to be woven, unwoven, or replaced at run-time [13].

Many developers have also exploited usage of aspects in development of multi-agent systems [14, 15], but these systems do not exploit usage of agents for implementing agents.

3. Aspect-Oriented Programming with Software Agents

In AOP we define join points, places where aspect code interacts with the rest of the system, and code that can run before, after or instead of the code at the join points. The main idea of our approach is to enable that aspect implementation be defined by agents which can be dynamically added or removed, instead of being fixed in the aspect code. We have defined a generic mechanism that enables the agents to access functionality of the system over aspects' join points, exploiting the fact that AOP systems offer implicit invocation mechanisms for invoking behavior in code whose writers often were unaware of the additional concerns.

3.1. Basic terms

In AspectJ, and other languages, you define the pointcuts (or join points), advices, aspects that group pointcuts and advices [16]. *Pointcut* are defined by designators that identify particular join points by filtering out a subset of all the join points in the program flow. For example, in AspectJ the pointcut designator:

```
call (void *.set*(int)) || call(void *.get*(int))
```

identifies any call to either the set or get methods defined by any class. *Advice declarations* are used to define additional code that runs at join points. *Before advice* runs just before the method begins to run. *After advice* just after the method has run, but before control is returned to the caller. *Around advice* runs when the join point is reached, and has explicit control over whether the method is allowed to run. This advice check values right before any set method call:

```
before():  call(void *.set*(int)) {
    <code to check value>
}
```

More detailed description of Aspect oriented programming with AspectJ can be found in [17].

3.2. Architecture

Figure 1 illustrates basic architecture of our approach. The central concept in our architecture is *the agent aspect*. This aspect provides definition of pointcuts at which the agents will access the systems functionality, as well as mechanisms for registering and invoking agents from aspects. Agent aspect keeps the list of registered agents, and accesses them over *the aspect agent interface*, which defines basic methods that each agent has to implement in order to be accepted by the aspect. Each pointcut is identified by its signature, and, when registering, agents can specify for which pointcuts they are interested. Optionally, agent aspect can provide the agents with a reference to *the*

application context, where the aspect can put and maintain additional references to the objects which agent can access. Agent aspect also has a reference to *the agent server interface*, which concrete agents server has to implement in order to be started and stopped by the aspect. The functional link between the aspect and the agents is achieved by aspect advices, which call appropriate agent methods.

Agent aspect could also provide default implementation of the advices, and use generic mechanisms for registration, and invocation of the agents, to extend or replace this default behaviour. When there are no agents registered, the default code of the aspects is unaffected.

Agents can be pre-registered by the system, and launched when the systems starts, while the other can run temporary, when they arrive from other machines. Agents that arrive at the system can be saved and launched when the system starts again. Agents can also be organized in a complex multi-agent system, where functionality is achieved by interaction among many agents. We are also working on hierarchically organized multi-agent systems that process data according to model-driven principles [18, 19].

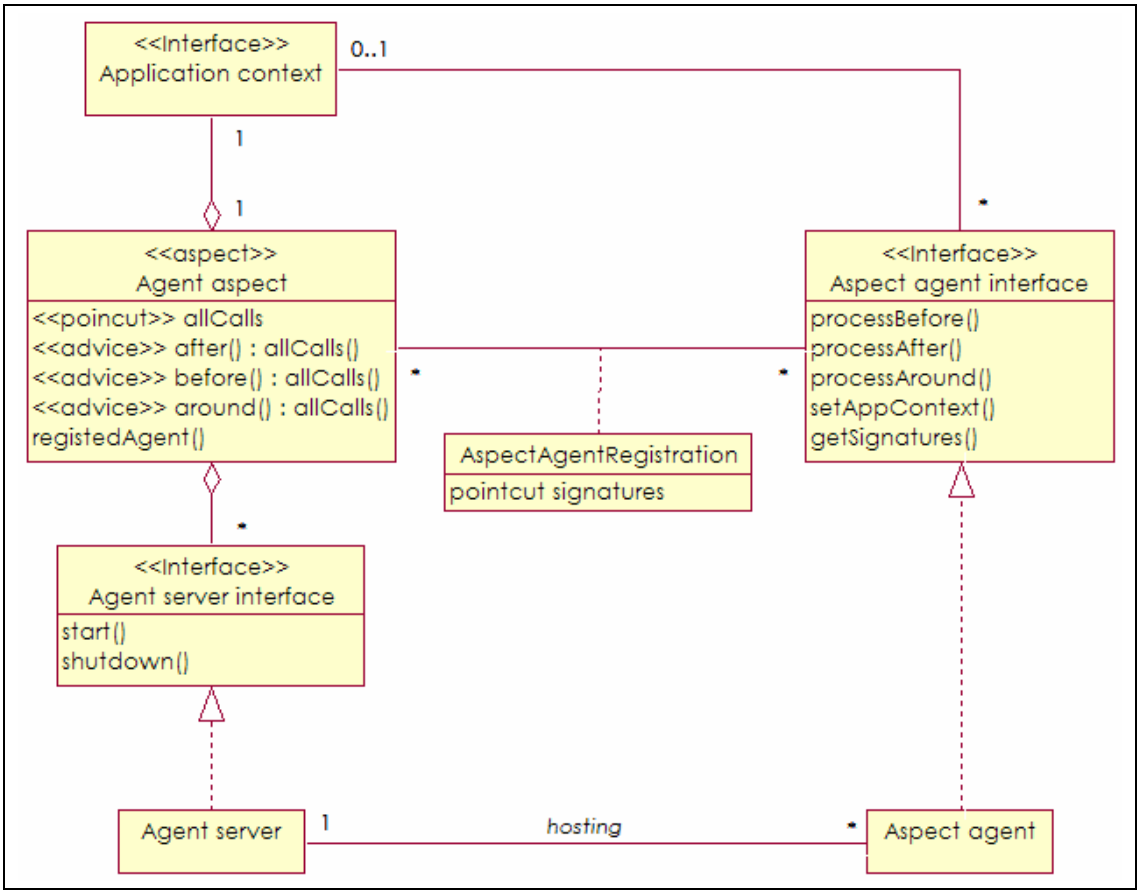

Figure 1. *The architecture of proposed aspect-agent architecture.*

Proposed model is flexible, as neither the aspect nor the agents do not have to know the details of the other side. When you want to introduce a novel type of agents platform, agent server adapter has to implement agents server interface, while new agents has to implement aspect agent interface, and they will be automatically accepted by the system.

3.3. Expected benefits

With the proposed framework, it is possible to create flexible, easily extensible programs, with several advantages:

- The programs could be more functional but better organized, inheriting this from aspect-oriented treatment of cross-cutting concerns.
- The programs could be more flexible, as aspects are defined in agents that can register/unregister in a central aspect to demonstrate interest in specific system joinpoints. Therefore, the programs could evolve over time, even based on runtime data. The programs behavior can be both added and removed at runtime by adding or removing agents.
- Existing applications could be easily extended in a flexible way without actually being reprogrammed (just a few generic aspects should be introduced).

Especially useful component of agents can be their mobility, which enables agents to reduce network load, and overcome network latency. Mobile agents can also be used to encapsulate protocols [20]. In a distributed system, each host has the implementation of the protocols required to correctly encode outgoing data and decode incoming data. However, it is often cumbersome to upgrade protocol code as it progresses in order to support various requirements such as efficiency or security. Consequently, protocols often turn into a legacy problem. Mobile agents can solve this problem by moving to remote hosts, and establishing channels based on proprietary protocols. Aspect-oriented approach can provide them with points of access to parts of the system that need to be updated.

Compared with "pure" multi-agent systems, using AOP approach can enable better integration with legacy applications, and with the code that is build on conventional, more stabile development platforms.

4. Implementation

We have implemented proposed approach using AspectJ and Aglets, two popular Java-based AOP and agent technologies.

AspectJ is a general-purpose AO extension to Java, originally developed by the PARC group [16, 21]. It is distributed under the terms and conditions of the Common Public License Version (CPL), and has gain developing support in most of popular Java developing environments.

Aglets is a Java mobile agent platform and library that facilitates the development of agent based applications, originally developed at the IBM Tokio Research Laboratory [22]. An aglet is a Java agent able to autonomously and spontaneously move from one

host to another. Aglets technology is now hosted at sourceforge.net as open source project, and is distributed under the IBM Public License.

4.1. Aspect Agent Interface

`AspectAgentInterface` is a Java interface that represents the link between the aspects and the agents. It introduces five methods implemented by agents and called by aspects (Figure 2). Each of these methods receives the signature of the pointcut, a reference to the object whose method is called, as well as the arguments of the method call.

```
public interface AspectAgentInterface {
    public void before( String pointcutSignature, Object target, Object[] args );
    public void after(String pointcutSignature, Object target, Object[] args );
    public void afterReturning(String pointcutSignature, Object target, Object[] args );
    public void afterThrowing(String pointcutSignature, Object target, Object[] args );
    public Object[] around(String pointcutSignature, Object target, Object[] args );
    public String[] getSignatures();
    public void setApplicationContext( ApplicationContext appContext );
}
```

Figure 2. *Simplified Java code of* `AspectAgentInterface`.

4.2. Aspect Agents

In Aglets, every agent has to extend the Aglet glass. Our agents additionally have to implement `AspectAgentInterface` (Figure 3). The agent code is the application specific, and when the system is established, should be the only part of the code that should be changed or extended.

```
public class AspectAglet extends Aglet implements AspectAgentInterface {
    public void before(String pointcutSignature, Object target, Object[] args ) {     }
    public void after(String pointcutSignature, Object target, Object[] args ) {   }
    public void afterReturning(String pointcutSignature, Object target, Object[] args ) {   }
    public void afterThrowing(String pointcutSignature, Object target, Object[] args ) {   }
    public Object[] around(String pointcutSignature, Object target, Object[] args ) {    }
    public String[] getSignatures() {
        return { "void javax.swing.JFrame.pack()",
                 "javax.swing.JFrame.setDefaultCloseOperation(int)", ... };
    }
    public void setApplicationContext( ApplicationContext appContext ) {   }
}
```

Figure 3. *Simplified code of AspectAgent implementation.*

4.3. Agents Server

Aglets enable development of customized extensions of its agent server. The key component for integration of custom code is the `ContextAdapter` class, which has to be registered with the agent server instance. Figure 4 shows how we have extended this adapter class to enable integration of agents with our aspects.

```
class AspectAgentContextAdapter extends ContextAdapter {
    public void agletCreated(ContextEvent ev) {    registerAgent( ev );     }
    public void agletArrived(ContextEvent ev) { registerAgent( ev );    }
    public void agletDisposed(ContextEvent ev) {
        Aglet agent = ev.getAgletProxy().getAglet();
        if (agent instanceof AspectAgentInterface)
            AgentAspect.unregisterAgent( (AspectAgentInterface) agent );
    }
    private void registerAgent(ContextEvent ev) {
        Aglet agent = ev.getAgletProxy().getAglet();
        if (agent instanceof AspectAgentInterface) {
            AgentAspect.registerAgent( (AspectAgentInterface) agent );
        }
    }
    ....
}
```

Figure 4. *Simplified code of* `AspectAgentContextAdapter` *implementation*.

This extension is simple and straightforward. When agents arrive at the server or when it is created, it calls the `registerAgent` method, which registers all the agents that implement `AspectAgentInterface` interface.

4.4. Agent Aspects

Figure 5 and 6 shows simplified AspectJ code for integration of agents into code via aspects.

```
aspect AgentAspectBeforeAfter {
    pointcut allCalls(): call(* *(..)) && !cflow(adviceexecution());

    before(): allCalls() {
        AgentAspect.processBefore( thisJoinPoint.getSignature().toString(),
            thisJoinPoint.getTarget(), thisJoinPoint.getArgs());
    }

    after(): allCalls() { ... }
}

aspect AgentAspectAround {
```

```
    int around(): call(int *(..)) && !cflow(adviceexecution()) {
        Object intValue = (Integer) aai.processAround( thisJoinPoint.getSignature().toString(),
            thisJoinPoint.getTarget(), thisJoinPoint.getArgs());
        if (intValue == null) {
            return proceed();
        } else {
            return ((Integer) intValue[0]).intValue();
        }
    }
    …
}
```

Figure 5. *Simplified code of the* `AgentAspectBeforeAfter` *and* `AgentAspectAround` *aspects.*

We developed three aspects to support this integration:

- `AgentAspectBeforeAfter`, device before and after advices to call registered agents using `AgentAspect` methods *before* and *after* given pointcut has been reached.
- `AgentAspectAround`, define around advices to call registered agents instead of the code at a given pointcut. The code at the pointcut will be proceeded or bypassed, based on the values that agents returns. This aspect is more complicated as it is necessary to support all the types which can be returned by methods (Object is used for all object types, but it is necessary to support all the primitive types).
- `AgentAspect`, which provides mechanisms for registration and deregistration of agents, starting of the agent server, and service functions for calling agents based on pointcut signatures. When agent is registered from the `AspectAgentContextAdapter` class, it firstly receives the reference to the application context. After that, it is registered in a hash table for every signature, and will be called when pointcuts with a given signatures are reached. If it returns null for signatures, it is registered in the `allJointPointsAgents` vector, and will be notified for every pointcut call. One agent can be registered for many join points, while one join point can be processed bay many agents. When there are many agents for one join point, agents are called as they have registered.

All presented aspects are generic, and define pointcuts that capture all method calls - `call(* *(..))`. However, it is also possible to extend these examples so that they be more appropriate for concrete context or application, and for other joinpoints types except method calls. If some existing application wants to use proposed framework, its code does not have to be changes. Instead, its code just has to be recompiled using AspectJ compiler, adding aspects code on the list of the files that have to be compiled.

```
public aspect AgentAspect {
    private static Vector allJoinPointsAgents = new Vector();
    private static Hashtable signatureJoinPointsAgents = new Hashtable();
    private static ApplicationContext appContext;

    public static void registerAgent( AspectAgentInterface agent ) {
        agent.setApplicationContext( appContext );
        String signatures[] = agent.getSignatures();
        if (signatures == null) {      // if there is no signatures, than calls from all join points
            allJoinPointsAgents.addElement( agent );
        } else {                       // if there are signatures, register agent for each of these signature
            for (int i = 0; i < signatures.length; i++) {
                Vector v = (Vector) signatureJoinPointsAgents.get( signatures[i] );
                if (v == null)
                    signatureJoinPointsAgents.put( signatures[i], v = new Vector() );
                v.addElement( agent );
            }
        }
    }
    public static void unregisterAgent( AspectAgentInterface agent ) { ... }
    public static void processBefore( String signature, Object target, Object[] args  ) {
        processBeforeVector( allJoinPointsAgents, target, args );
        processBeforeVector( (Vector) signatureJoinPointsAgents.get( signature ), target, args );
    }
    private static void processBeforeVector( Vector v, Object target, Object args[] ) {
        Iterator e = allJoinPointsAgents.iterator();
        while (e.hasNext()) {
            AspectAgentInterface agent = (AspectAgentInterface) e.next();
            agent.before( target, args );
        }
    }

    public static void processAfter( String signature, Object target, Object[] args  ) { ... }
    private static void processAfterVector( Vector v, Object target, Object args[] ) { ... }
    public static Object[] processAround( String signature, Object target, Object[] args  ) { ... }
    private static Object[] processAroundVector( Vector v, Object target, Object args[] ) { ... }

    pointcut appStart() : execution(public static void *.main(String[]));

    before(): appStart() {   AgentServer.start();   // Start the agent server }
}
```

Figure 6. *Simplified code of the* `AgentAspect` *aspect.*

4.5. Performance evaluation

Having in mind that aspects introduce additional code before, after, and around each method, it is important to see how this affects the performance of the system. We particularly wanted to see how this affects the performance when there are no agents in the system, e.g., what is the cost of the proposed system, compared with the same version of the system compiled without our generic aspects.

We did several tests, using the HP OmniBook notebook computer with a Pentium III processor on 800 Mhz, and with 375 MB RAM memory, using a test program shown on the Figure 7. This program calls dummy method test 10000000 times, while each method call also introduce three aspect method calls: after, before, and around.

```
public class Test {
    public static void main(String args[]) {
        Test t = new Test();
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 10000000; i++) {
            t.test();
        }
        long endTime = System.currentTimeMillis();
        System.out.println( endTime - startTime );
    }
    public void test() {}
}
```

Figure 7. *A test program. Each method call introduces three aspect advices method calls: after, before, and around.*

Initial tests showed us that there is 0.0165ms average overhead per method call, compared with the same program compiled without aspect code. However, when we introduced small changes in aspect code, introducing a logical variable that described if there are agents in the system, and when we call the routines of the `AspectAgent` class methods only if this variable is true, this overhead was reduced to 0.0012ms per method call. Starting of the Aglets agent server introduced memory overhead of about 3 MBs.

5. Applications

We are currently applying proposed approach in several application domains:

Adaptive monitoring and notification. Incorporation of generic aspects in program enables users, administrator or developer to send agents at any time to monitor the program behaviour and state, or to send notification when some event happens. For example, a simple agent can register to receive a call after the main method throws, which correspond to abnormal program termination, and can send notification to the

administrator or other agents (Figure 8). It is also possible to send agents to monitor particular program behaviour in detail. This way of monitoring and notification is much more flexible. In classical AOP approach, you could define the debugging aspect, which could monitor the system during development and testing, but it is necessary to recompile the code each time when you want to apply different debugging policy, or when you want to remove monitoring.

```
public class ThrowingAglet extends AspectAglet {

    // send a UDP package to the server each time when some method end throwing
    public void afterThrowing(String pointcutSignature, Object target, Object[] args ) {
        String message = System. + "; " + pointcutSignature;
        DatagramSocket socket = new DatagramSocket();
        byte[] buf = new byte[256];

        DatagramPacket packet = new DatagramPacket(
                    message.getBytes(),    // content
                    message.getBytes().length, // content size
                    address,     // address of the client
                    7778);    // port
        socket.send(packet);
        socket.close();
    }
    public String[] getSignatures() {
        return  null;    // We are interested in all signatures
    }
}
```

Figure 8. *Simplified `ThrowingAglet` notification agent, which sends UDP packages to the server each time when some method ends throwing.*

Upgrading and patch distribution. Due to marketing pressure, many software products are distributed with many known and even more unknown bugs. Correcting these errors, often mean reinstalling the application, or installation of various "patches", which in many situations is not convenient. However, if the program has support for aspect as we have described it, it is possible to send an agent that can replace the wrong code, or upgrade it to a new version.

Adaptive fault-tolerant applications. By combining adaptive monitoring and upgrading, it is possible to create more robust applications, than can adapt its behaviour. For example, when an error occurs, the agent can create or call a new agent to replace a code that produced the error. This new agent could provide safe or dummy behaviour, for example, by returning a constant value, enabling the application to function until the patch agent is received. Similarly, the agent could have several versions of the code, for example, from previous version stabile version of the product, and try to use this version when there are problems.

Adaptive performance tuning of applications. It is often hard to predict how some component will work when integrated with the rest of the system, and when used in real-world situations. With the proposed framework, it is possible to have several alternative implementations of some component, represented with different agents. The agents could use several versions, and monitor program performance, analyzing which one produce better performance. Finally, it can register component with best results. It is possible that the same applications in different environments will use different versions of the components, due to various limitations. For example, some recursive algorithms could produce stack-overflow on one system, but function without problems on the other.

Computer supported collaborative work. Proposed framework can also support integration of various communication protocols that can support interaction among distributed user. Agents could enable recording of user interaction, simultaneous work, or introduction of various communication channel such as chat, email, blogging or voice. Introduction of new communication protocol only requires replacing of the agents, which can be made automatically by the server.

6. Conclusion

In this paper we have presented a hybrid approach to software development based on aspect-oriented programming (AOP) and software agents. In our framework aspect's implementation is not fixed in the code, but is instead implemented by software agents. We defined a generic mechanism that enables the agents to access functionality of the system over aspects' join points, exploiting the fact that AOP systems offer implicit invocation mechanisms for invoking behavior in code whose writers often were unaware of the additional concerns. We also showed how proposed approach has been implemented using AspectJ and Aglets, two popular Java based AOP and agent technologies, and described some of the areas in which we are currently applying these ideas.

Program system build in this way can get several advantages for practitioners in software engineering and software agents community. The programs could be more functional but simpler and better organized, inheriting this from aspect-oriented treatment of cross-cutting concerns. The programs could be more flexible, as aspects are defined in agents that could be replaced or reorganized. And, existing applications could be easily extended in a flexible way without actually being reprogrammed.

In our future work, we plan to introduce support for other agent platforms, and to continue applying of our framework to various novel practical situations. One of the open problems is a problem of security, which we also plan to address in more details in our futur work.

7. References

1. T. Elrad, R.E. Filman, A. Bader, "Aspect-oriented programming: Introduction", Communications of the ACM, Vol. 44, No. 10, pp. 29 - 32 (2001).
2. C.W. Thompson. "Agents, Grids, and Middleware," IEEE Internet Computing, vol. 08, no. 5, pp. 97-99, September/October (2004).
3. U. Kulesza, A. Garcia, C. Lucena, "An aspect-oriented generative approach", Conference on Object Oriented Programming Systems Languages and Applications, Vancouver, BC, Canada, pp. 166 - 167 (2004).
4. N. R. Jennings, "An Agent-based Approach for Building Complex Software Systems", Comm. of the ACM, Vol. 44, No. 6, April 2001, pp. 35-41;
5. M.J. Wooldrige, Nicholas R. Jennings, "Software Engineering with Agents: Pitfalls and Pratfalls", IEEE Internet Computing, May/June 1999, pp. 20-27;
6. P. K. McKinley, S. M. Sadjadi, E. P. Kasten, B. H. C. Cheng, "Composing Adaptive Software", IEEE Computer, 37(7):56-64, July 2004.
7. K.J. Ma. "Web Services: What's Real and What's Not?," IT Professional, vol. 07, no. 2, pp. 14-21, March/April 2005.
8. M. Nishizawa, Shigeru Chiba, Michiaki Tatsubori, "Remote pointcut: a language construct for distributed AOP", Proceedings of the 3rd international conference on Aspect-oriented software development table of contents. Lancaster, UK, 2004, pp. 7 - 15.
9. G.T. Sullivan, "Aspect-oriented programming using reflection and metaobject protocols", Communications of the ACM, Vol. 44, No. 10 (October 2001), pp. 95 - 97
10. S. Vinoski. "A Time for Reflection," IEEE Internet Computing, vol. 09, no. 1, pp. 86-89, January/February 2005.
11. M. Román, F. Kon, R. Campbell. "Reflective Middleware: From Your Desk to Your Hand". IEEE Distributed Systems Online (Special Issue on Reflective Middleware). Vol. 2, No. 5. July, 2001. ISSN: 1541-4922.
12. F. Kon, F. Costa, R. Campbell, G. Blair. "The Case for Reflective Middleware". Communications of the ACM. Vol. 45, No. 6, pp. 33-38. June, 2002.
13. A .Popovici, T. Gross, G. Alonso. "Dynamic Weaving for Aspect Oriented Programming". In: 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, April 2002.
14. A. Pace, F. Trilnik, M. Campo. "Assisting the Development of Aspect-based MAS using the SmartWeaver Approach". In: "Software Engineering for Large-Scale Multi-Agent Systems", LNCS 2603, March 2003.
15. A. Garcia, C. Lucena, D. Cowan. "Agents in Object-Oriented Software Engineering". Software: Practice and Experience, Volume 34, Issue 5, April 2004, pp. 489-521.
16. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, "Getting started with AspectJ", Communications of the ACM, Vol. 44, No. 10 (October 2001), pp. 59 - 65.
17. A. Colyer, A. Clement, "Aspect-oriented programming with AspectJ", Volume 44, Number 2 (2005), Special Issue on Open Source Software, pp. 301-308.
18. Z. Obrenovic, D. Starcevic, E. Jovanov, V. Radivojevic, "An Agent-Based Framework for Virtual Medical Devices", Proceedings of The First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2, July 15-19, 2002, Bologna, Italy, pp. 659 - 660.
19. Z. Obrenovic, D. Starcevic, B. Selic, "A Model Driven Approach to Content Repurposing", IEEE Multimedia, Vol. 11, No. 1, January-March 2004, pp. 62-71;
20. D.B. Lange, M. Ochima, "Seven Good Reasons for Mobile Agents", Comm. of the ACM, Vol. 42, No. 3, March 1999, pp. 88-89.
21. AspectJ Web Site, http://eclipse.org/aspectj/, Last visited: November 11, 2005
22. The Aglets Web Site, http://aglets.sourceforge.net/, last visited November 11, 2005