# Vrije Universiteit Brussel - Belgium

## Faculty of Sciences

### In Collaboration with Ecole des Mines de Nantes - France


### 2007



# On aspect-oriented concurrent and distributed patterns

A Thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)


By: Diego Fernando Rivera Pardo

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Mario Südholt ( Ecole des Mines de Nantes )
Co-Promoter: Wim Vanderperren ( Vrije Universiteit Brussel )

# Abstract

The implementation and maintenance of distributed and concurrent applications are complex tasks. Part of this complexity is introduced by heterogeneous distributed communication requirements, and the irregular topology context where they are defined and used. Distributed and concurrent patterns have been proposed to decrease the complexity and promote reuse of well proven solutions. However, the specification and implementation of these patterns have not succeed to handle communication and topology complexity requirements, resulting implementations are most of the time scattered around many classes and modules. This thesis shows how current state of the art technologies for distributed and concurrent AOP can improve, partially, modularization and reusability of pattern implementation.

Based on such study we also propose, AWED Prime, an extension to AWED language. AWED is an AOP language with explicit support for distribution: it provides remote pointcuts, remote advices, a/synchronous advice execution and statefull aspects. AWED Prime extends AWED in order to handle topologies and improve previously defined remote pointcuts, defining predicates between groups and hosts, these predicates can be used at pointcut or advice level to support communication and topology requirements. The syntax and semantics are defined, an implementation strategy and two use cases are described to show how AWED Prime can be used to implement distributed patterns, in particular the message group feature of the Apache ActiveMQ application.

# Acknowledgements

For the realization of this master many people have influenced, helped or supported me in many ways, that is why I would like to thank them all: to my family, they are always whenever I need them, giving me love, support and good advices, to Luce for giving me love, inspiration and motivation, my old friends I would not be what I am without you, to my new friends, this year would not have been the same without you, to Theo D'Hondt for turning upside down my object oriented world, to Rubby Casallas for her advices and guidance, to Rob Davies for his support on Apache ActiveMQ, to the people of The Ecole des Mines de Nantes and the Vrije Universiteit Brussel for sharing their knowledge and support, specially to Jacques, Mario, Daniel, Wim, Bruno and Peter, last but most important to God, I am grateful for everything you give me.

# Contents

# List of Figures

# Chapter 1

# Introduction

The implementation and maintenance of distributed and concurrent applications are complex tasks. Part of this complexity is introduced by heterogeneous distributed communication requirements, and the irregular topology context where they are defined and used. Distributed and concurrent patterns have been proposed to decrease the complexity and promote reuse of well proven solutions. However, the specification and implementation of these patterns have not succeed to handle communication and topology complexity requirements, resulting implementations are most of the time scattered around many classes and modules. This thesis shows how current state of the art technologies for distributed and concurrent AOP can improve, partially, modularization and reusability of pattern implementation.

Based on such study we also propose, AWED Prime, an extension to AWED language. AWED is an AOP language with explicit support for distribution: it provides remote pointcuts, remote advices, a/synchronous advice execution and statefull aspects. AWED Prime extends AWED in order to handle topologies and improve previously defined remote pointcuts, defining predicates between groups and hosts, these predicates can be used at pointcut or advice level to support communication and topology requirements. The syntax and semantics are defined, an implementation strategy and two use cases are described to show how AWED Prime can be used to implement distributed patterns, in particular the message group feature of the Apache ActiveMQ application.

Patterns are reusable definition of solutions to common problems , they include the participants, their interaction, the context in which they have to be used and also the implications of using them. Object Oriented implementation of distributed and concurrent patterns have proven to be complex, not reusable and not sustainable, because of this, the implementation of patterns using AOP [19] has been investigated, showing improvement of reusability, (un)plugability and modularity. Distributed pattern implementations are complex, because the computation takes place on distributed machines, this distribution has to be managed by the the programmer, using libraries, sockets or frameworks, among others, as result of this, implementation of distributed patterns are polluted and scattered

with concerns of distribution. Research of how distributed AOP can support the implementation of concurrent and distributed patterns will state current problems, limitations and new ways to support them.

Languages extensions to support requirements are common, Jonathan Aldrich proposed the use of types to enforce architectural structures definition [3], as a result of this propose, developers gained architectural documentation and validation at language level. AWED was created as an extention common AOP languages in order to support distributed AOP, and now we propose the extension of AWED, to promote topology use, better distribution communication requirements, and better deployment, this extension may result in better understanding, definition and evolution of application that use concurrent and distributed patterns. Pattern language support is currently being researched by the Computer Science department at Ecole des Mines de Nantes, this research [7] has achieved a definition of a pattern language, the foundation of this language are well known distributed patterns like gather, farm and pipeline and sequences of patterns, the pointcuts and advices can be applied to one or many hosts and the sequences of the patterns refers to the synchronization of the aspectized patterns. We propose the extension to current AWED language in order to handle previously defined requirements on the implementation of patterns. This extension is applied on Active MQ [15], this application uses the Publish Subscribe distributed pattern, one implementation of that pattern is the Message group feature, its code is scattered around many classes and packages, and tangled with other functionalities, the proposed solution of this problem uses the topology management extension to modularize host communication concern in the aspect.

This document is structured as follows: the current state of the art is documented(Chapter 2), focusing on on the design patterns(section 2.1) and on Distributed Aspect Oriented languages(section 2.2). Main contributions of this master research are: First, experimentation of distributed aspect oriented language to implement concurrent and distributed patterns(Chapter3), this experiment was done over AWED's [8] implementation DJAsCo [41]; Second, an extension to the current Aspect With Explicit Distribution(AWED) language is defined, to overcome the current difficulties encountered on the implementation of the concurrent and distributed patterns, specially irregular topology and communication requirements, it is called AWED Prime(Chapter4); Third, overview of how the extension gives support to the Master Slave and Publish Subscribe pattern implementations, the last pattern, on Apache ActiveMQ [15] message broker framework(Chapter 5); Finally conclusions and future work(Chapter 6).

# Chapter 2

# State of the Art

This chapter presents the state of the art on aspect oriented programming support for pattern implementation, first we consider software patterns. In section 2.1 we present sequential patterns, concurrent, and distributed definitions and examples of patterns, and how AOP has been used to implement them. In section 2.2, we present the definition for distributed AOP, some language approaches and examples of its use and section 3 conclusions on the state of the art.

## 2.1 Patterns

In this section we present sequential(section 2.1.1), concurrent(section 2.1.2) and distributed(section 2.1.2) patterns, pattern examples, and how AOP has been used to implement them, results of their implementation and lessons learned.

Design patterns were first thought as civil and architectural design concept by Christopher Alexander [4] . In 1987, Kent Beck and Ward Cunningham experimented with the idea of applying patterns to programming and presented their results at the OOPSLA conference in 1987, but it was not until 1995 when The Gang of Four(GoF: Erich Gamma, Richard Helm, Ralph Johnson and John Vliaaides) in *Design patterns* [17] that Patterns really encounter acceptance on the software engineering community. Each Pattern describes the context where it is applied, it also describes the consequences of using it. In addition, they must balance, or trade off, a set of opposing forces. The way we describe patterns must make all these things clear. Clarity of expression makes it easier to see when and why to use certain pattern, as well as when and why not to use these patterns. All solutions have costs, and pattern descriptions should state the costs clearly. [37]

Another common definition of design pattern is the reuse of high-quality solutions to commonly recurring problems. The use of patterns can achieve some of the advantages like: easier code modification, higher level of thinking, it also improves communication

because it establishes a common terminology and discover alternatives to large inheritance hierarchies [1].

On the other hand, architectural patterns express a fundamental structural organization or schema for software systems. also they provide a set of predefined subsystems, define their responsibilities, and include rules and guidelines for organizing the relationships between them[50].

AOP has been used to implement design and concurrent pattern, this study has encountered that AOP improves modularity, reusability and in some cases decreases complexity. The study of AOP implementations on sequential design pattern and concurrent, will give us an idea of how AOP can be applied and what results we can expect of AOP on concurrent and distributed patterns. The sequential, concurrent and distributed patterns are presented on this section, including examples of how AOP has achieve modularity, reusability and decrease of lines of code in some of them.

### 2.1.1 Sequential patterns

This section presents the definition of sequential design patterns, AOP study on them and results, that show how AOP has achieved modularity and reusability on their implementation.

The Gang of Four(GoF) Design Patterns are the most recognized, they have defined 23 design patterns, those first became popular with the wide acceptance of the book Design Patterns: Elements of Reusable Object-Oriented Software by the GoF. The GoF design patterns are grouped into three categories *behavioural* that are those that are specifically concerned with communication between objects, *creational*, that are concerned on class-creation patterns and object-creational patterns, and finally *structural* design patterns that are concerned in Class and Object composition.

Design patterns have encountered opposition for many reasons, but one of the main reasons is that it does not support any reuse, whenever a pattern is implemented in the traditional OO way the implementation of this is now inside the base code, **it cannot be reused** and the implementation code is now **scattered** around the base code and if there is composition of patterns there is also **tangling** of the implementation code of the patterns. The mediator pattern is a behavioural pattern, the access of the *Mediator* class to its mediated classes are found not only in its implementation but also in the mediated classes, so there is a bidirectional access that works only in this specific case.

**Ex**: *Mediator pattern* aims to solve communication needs, between several objects, they can differ or be from the same class. In the implementation of the Mediator pattern it is common to see Classes implementing interfaces that have no relation with its functionality

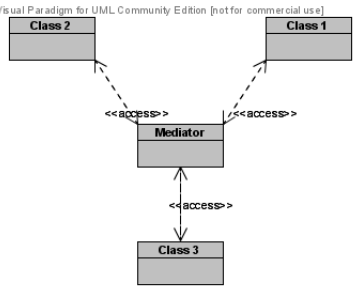


Figure 2.1: Mutual access in mediator pattern.

or declaration, the following code and figure shows one of the possible interface declaration of the pattern that will be translated to code pollution and bidirectional access as shown in Figure 2.1.

Some of the problems of design pattern OO implementation are: *code pollution*; Classes that implement the pattern specific behaviour, then complexity is increased. Design patterns composition will also create patterns code mix *scattered* around Classes and finally the patterns will not be *localized* or identified, making it harder for code evolution and maintain.

### AOP and sequential patterns

Many design patterns have been used to implement and experiment on how they affect the reusability and modularization, Jan Hannemann and Gregor Kiczales implemented the 23 GoF design patterns and studied the impact of AOP on them.

The method used by them was to implement the 23 GoF patterns using simple Java and AspectJ also, in many cases multiple implementations could be done to a single design pattern, the result of that is that at the end they have done 57 implementations. The result of this research gives that 74% of design patterns where implemented in a more modular way, and 52% are reusable using the AOP approach [19] . As an example of better modularization, the Mediator design pattern, the comparison of the OO and AOP implementation is showed in Figure 2.2 .

**Ex**: *Mediator pattern*, the following code defines a reusable aspect definition of the mediator protocol, it defines a WeakHashMap data structure to associate a colleague with its mediator, a definition 100% reusable [19] , now that only the general behaviour is specify on this protocol.

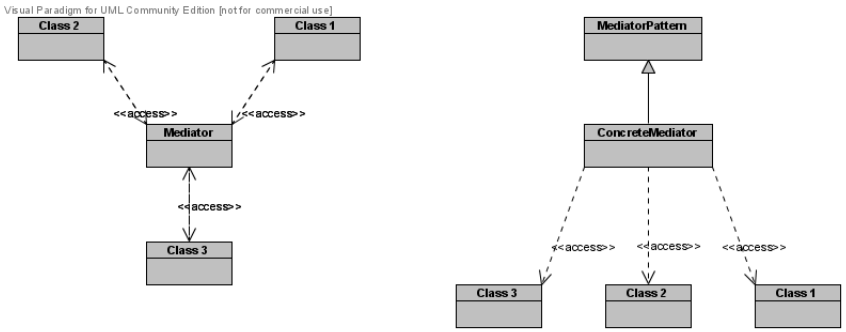


Figure 2.2: Dependencies and code distribution, mediator pattern.

```
1
3  public abstract aspect MediatorProtocol {
       /**
5       * Declares the Colleague role.
        * Roles are modeled as (empty) interfaces.
7       */
       protected interface Colleague { }
9
       /**
11      * Declares the <code>Mediator</code> role.
        * Roles are modeled as (empty) interfaces.
13      */
       protected interface Mediator { }
15
       /**
17      * Stores the mapping between <i>Colleagues</i>s and <i>
        * Mediator</i>s. For each <i>Colleague</i>, its <i>Mediator</i>
19      * is stored.
        */
21     private WeakHashMap mappingColleagueToMediator = new WeakHashMap();

23     /**
        * Returns the <i>Mediator</i> of
25      * a particular <i>Colleague</i>. Used internally.
        *
27      * @param colleague the <i>Colleague</i> for which to return the mediator
        * @return the <i>Mediator</i> of the <i>Colleague</i> in question
29      */
       private Mediator getMediator(Colleague colleague) {
31         Mediator mediator =
               (Mediator) mappingColleagueToMediator.get(colleague);
33         return mediator;
       }
35     /**
        * Sets the <i>Mediator</i> for a <i>Colleague</i>. This is a method
37      * on the pattern aspect, not on any of the participants.
        *
39      * @param colleague the <i>Colleague</i> to set a new <i>Mediator</i> for
        * @param mediator the new <i>Mediator</i> to set
41      */
       public void setMediator(Colleague colleague, Mediator mediator) {
43         mappingColleagueToMediator.put(colleague, mediator);
       }
45
       /**
47      * Defines what changes on <i>Colleague</i>s cause their <i>Mediator</i>
        * to be notified
49      *
        * @param colleague the <i>Colleague</i> on which the change occured
51      */
       protected abstract pointcut change(Colleague colleague);
53
       /**
55      * Call updateObserver to update each observer.
        */
57
```

```
    after(Colleague c): change(c) {
59      notifyMediator(c, getMediator(c));
    }
61  /**
     * Defines how the <i>Mediator</i> is to be updated when a change
63   * to a <code>Colleague</code> occurs. To be concretized by sub-aspects.
     *
65   * @param c the <i>Colleague</i> on which a change of interest occured
     * @param m the <i>Mediator</i> to be notified of the change
67   */
    protected abstract void notifyMediator(Colleague c, Mediator m);
69 }
```

The following concrete code of the general definition of the Mediator pattern.

```
1

3 public aspect MediatorImplementation extends MediatorProtocol {
    /**
5    * Assigns the <i>Colleague</i> role to the <code>Button</code>
     * class. Roles are modeled as (empty) interfaces.
7    */
    declare parents: Button implements Colleague;
9   /**
     * Assigns the <i>Mediator</i> role to the <code>Label</code>
11   * class. Roles are modeled as (empty) interfaces.
     */
13  declare parents: Label implements Mediator;
    /**
15   * Defines what changes on Colleagues cause their <i>Mediator</i> to be
     * notified (here: Button clicks)
17   *
     * @param cs the colleague on which the change occured
19   */
    protected pointcut change(Colleague c):
21      (call(void Button.clicked()) && target(c));

23  /**
      * Defines how the <i>Mediator</i> is to be updated when a change
25    * to a <i>Colleague</i> occurs. Here, the label's text is set
      * depending on which button was clicked. The appropriate button's label
27    * is also updated.
      *
29    * @param c the colleague on which a change of interest occured
      * @param m the mediator to be notified of the change
31    */
    protected void notifyMediator(Colleague c, Mediator m) {
33      Button button = (Button) c;
        Label label = (Label) m;
35      if (button == Main.button1) {
            label.setText("Button1 clicked");
37      } else if (button == Main.button2) {
            label.setText("Button2 clicked");
39      }
        button.setText("(Done)");
41  }
}
```

The previous code examples are taken from the study made by Jan Hannemann and Gregor Kiczales [19]. It shows the reusability of the pattern, its aspect definition can be extended in order to be applied as needed in any other case.

The concrete example of the Mediator pattern has two colleagues: two buttons and a label that is the mediator, the Mediator aspect has the task to communicate changes between these buttons and the label, whenever a button is clicked the label changes, the concrete aspect resolves the needs of this example, but the general Mediator aspect can be used in any case.

**The results** of improvements were defined in terms of locality, reusability, dependency inversion and transparent (un)pluggability. **Locality** defines how well localized a design

pattern implementation is, this is how easy is to find it and how modularized is, the result showed that 17 of the 23 GoF patterns were localized. **Reusability** defines if there is the possibility of using part or all the code for different specific implementation of the design pattern, the research gave as a result that 12 of the 17 localized GoF patterns were reusable [19], reusability is achieve by means of defining a global aspect where general behaviour of the pattern is specified and then a concrete aspect to the specific behaviour where the concrete case is implemented.

**Dependency inversion** states that the pattern code depends on the participants and not the other way around; the result is such that all the dependencies between pattern and participants were localized at the pattern implementation, now that aspects depend on the base code and not the other way around, this characteristic is achieved implicitly by means of AOP specification. On the other hand, **(un)plugability** establish that there is no need to have code from the pattern implementation at the base code, in such a way that it can be added or removed , this was possible on the participants that have did not have any other responsibility inside the system.

The counterpart of the aspectization of design patterns was studied in *Modularizing Design Patterns with Aspects: A Quantitative Study* [18] , they have done a study of the 23 GoF patterns LOC(Lines of Code), WOC(Weighted Operations per Component) between other, they have found some interesting results: the use of generic abstract aspect did not always achieve reusability, they mention the cases for *The Flyweight, Command, CoR, Memento, Prototype, Singleton*, those were ranked as reusable in the study made by *Jan Hannemann and Gregor Kiczalest*(HK), contrary to defined by HK on this study the LOC and WOC on the AO implementations were higher than in the OO implementations, on the other hand, the Memento pattern implementation showed some bad results were found on the AO implementation like higher complexity in terms of coupling , inheritance complexity and also more lines of code.

### 2.1.2 Concurrent patterns

Concurrent programming addresses the following problems: several accesses to resource from different programs or processes of the same program and partial calculation in several machines. This section presents the definition of concurrent patterns, AOP study on them and results that show how AOP has achieved modularity and reusability on their implementation. The presented examples are: The shared resource pattern, which is one of the first proposed patterns for concurrency; Active object patterns, a non-trivial pattern; Waiting guards pattern, widely used.

*A concurrent design pattern is a small grouping of tasks and protected units that is useful in many applications* [36]. Whenever someone wants to define a design pattern, there is the need to express with words the pattern, then translate it to the language specific

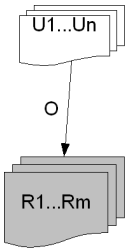


Figure 2.3: Shared-resource pattern

support for concurrency, diagrams are not good enough to express concurrency due to its two-dimensional limitation, sequence diagrams in UML do not achieve to express the full context of the concurrent design pattern, its participants and their collaboration, so in order to overcome this limitation a mix between the description and figures is used.

**Ex**: *The Shared-Resource pattern* [36], this pattern is applicable in all concurrent programs where multiple tasks or threads need exclusive access to one or more resources, Figure 2.3 shows this patern.

**Participants**

- U1 - Un are resource users
- RI - Rm are resources

**Interactions**

- Ui operates on Rj as indicated by the direction of the arrow. For each resource Rj, the exclusive access can be either hidden or public.
- With hidden exclusive access, the operation, Op, contains the critical section where Rj is manipulated, i.e., each caller Ui has exclusive access to Rj while executing Op. The exclusive access is then hidden inside Op.
- With public exclusive access,Op0 stands for one operation to acquire exclusive access to the resource and one to release the exclusive access. An Acquire operation returns control to Ui with Ui having exclusive access to Rj. The exclusive access now occurs in Ui. More complex interactions between resource users and resources than Acquire. Ui may acquire exclusive access to Rj then proceed to acquire Rk before releasing Rj. This is simultaneous exclusive access.

**The Results** show that when implementing patterns for concurrent programming it is crucial to have some support to the concurrent operations by the language(as synchronized in Java) or in other cases libraries(C++), there are also extensions for languages to support concurrency $\mu$ C++ [35] for C++, Concurrent Smalltalk [49] for Smalltalk [38] , so it is true to admit that even when the design patterns are defined as text and meta algorithms their implementations are dependant to the language chosen and their support for concurrency. Concurrent programming is not only dependant to the context of the language or the libraries available, but its code implementation is scattered around the objects that need to have this concurrent behaviour. [36]

### AOP and concurrent patterns

Concurrent programming has been relegated to experts [12] due to its complexity, the concurrency concern is not an object-oriented composition, then the implementation of concurrent patterns also suffer from the code tangling and scattering phenomena [23]. AOP was proposed to achieve modularization of cross-cutting concerns. On the other hand design patterns in concurrent programming were defined in order of reuse practices that have proven to achieve some improvement on the code in terms of flexibility, but in many cases they increase complexity and make the understanding more complex. Design pattern implementation in concurrent programming is studied in *"Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms"* [12].

**Ex**: *The Active Object pattern,* showed in Figure 2.4 was created to invoke a thread per every object, the normal implementation was divided into *3 levels.The first lever* is the object that makes the call, *the second level* is the behaviour that sends the call to the specific object and on the *third level*, there is a thread per running object that waits for the method calls. The aspect oriented implementation of the Active Object pattern moves the *level 2 and 3* to the aspect, so the original method of the running object can be called as normal, without any pollution of synchronization and with no coupling with the aspect, so this object is oblivious to the aspect that implements the pattern. The following is the implementation of this pattern.

```
public aspect ActiveObjectA extends ActiveObjectProtocol {
    declare parents :A implements ActiveObject;
}
```

The implementation of the ActiveObjectProtocol [12] as follows:

```
1
public privileged abstract aspect ActiveObjectProtocol {
3
    protected WeakHashMap<ActiveObject,MQScheduler> hash = new WeakHashMap<ActiveObject,MQScheduler>();
5
    protected interface ActiveObject{};
7   declare parents : @concurlib.annotations.ActiveObject * implements ActiveObject;

9   /**
     * Defined in aspect subclasses
11   */
    protected pointcut create(ActiveObject s) : execution(ActiveObject+.new(..)) && this(s);
```

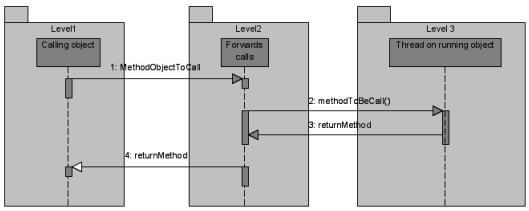


Figure 2.4: Active object OO levels.

```
13      protected pointcut callMeth(ActiveObject s) : call(public * ActiveObject+.*(..)) && target(s);

15      /**
         * Servant thread creation for each object instantiated
17       */
        before(ActiveObject s) : create(s){
19          MQScheduler mqs = new MQScheduler(50);
            synchronized(this){ hash.put(s, mqs); }
21          (new Thread(mqs)).start();
        }
23
       /**
25      * Methods that return values
        */
27     Object around(final ActiveObject s): callMeth(s){
              Message ms = new Message();
29          MethodRequest mr = new MethodRequest(new Callable(){
                private Object msg = null;
31              public void call(){ msg = proceed(s); }
                public Object getValue(){ return msg; }
33              },ms);

35          sendToQueue(mr,s);
            return waitForValue(mr, s);
37      }

39     private Object waitForValue(MethodRequest mr, ActiveObject s)
       {
41          Object res = null;
            try{
43              res = mr.getResult().getValue();
            }catch(Throwable t){
45              throw new ConcernRuntimeException(t);
            }
47          return res;
       }
49
      /**
51     * Send to ActivationList through Scheduler
       */
53     protected synchronized void sendToQueue(MethodRequest mr, ActiveObject s){
                hash.get(s).insert(mr);
55     }
}
```

*Level 2 and 3* of the pattern were modularized into the aspect that implements the active object concurrent pattern, the aspect can be **fully reusable** just declaring that a class implements the ActiveObject interface or if the class has the ActiveObject **annotation**. So, Modularization is achieved, reusability is also achieved.

**Ex**: *Waiting guards pattern,* the execution of a method depends on the state of the object, if a precondition is not satisfied the waiting guards puts the invocation on waiting until the condition is satisfied, other threads can change the state of the object so the condition can be achieve or a time-out can be given to the waiting guard. The Waiting Guard [12] concrete code and general protocol as follows.

```
public aspect aspect_name extends WaitingGuardsProtocol {

    protected pointcut deblockingOperation(Object targetObject) :<pointcut definition>;
    protected pointcut blockingOperation(Object targetObject): <pointcut definition>;

    protected boolean preCondition(Object ob, Object[] args) {
        return <precondition validity>;
    }

    //and optionally override method getWaitingTime
    protected long getWaitingTime(){
        return <time in milliseconds>;
    }
}


**********************************************************************

public privileged abstract aspect WaitingGuardsProtocol{
// private boolean blocked = false;
    protected abstract pointcut deblockingOperations(Object targetObject);
    protected abstract pointcut blockingOperations(Object targetObject);

    protected abstract boolean preCondition(Object ob, Object[] arguments);

    protected long getWaitingTime(){ return 0; }

    after(Object targetObject) : deblockingOperations(targetObject) {
        synchronized(targetObject){
//    state.blocked = false;
            targetObject.notifyAll();
        }
    }

    before(Object targetObject) : blockingOperations(targetObject){

        try{
            synchronized(targetObject){
                while(! preCondition(targetObject, thisJoinPoint.getArgs())){
                    //blocked = true;
                    targetObject.wait(getWaitingTime());
                }
            }
        }catch(InterruptedException c){
            throw new ConcernRuntimeException(c);
        }
    }
}
```

The **modularization** of this concurrent implementation is also achieved, there is also **partial reusability**, only preconditions, blocking and deblocking conditions must be specified for the concrete application of this pattern, then reusability is also achieved, the complexity of dealing with concurrency is moved into the aspect.

**Results** Some of the results of the study made by Carlos Cunha, Joao Sobral and Miguel Monteiro [12] are divided into the four following categories:

*Modularization* is achieved when implementing a single concurrent design pattern, the complexity is moved to the aspect and located on it, Java language synchronization is needed to implement these patterns but there could also implemented by aspects that

define monitors, semaphores and so on, it could be really interesting if concurrency support is implemented with aspects.

*Annotations* can also be used to mark a class, a method or an instance variable in order to be able to declare that will be applied to more methods, but then there can be annotation scattering and also tangling.

*Composition*, concurrent pattern composition was partially done, but more complex compositions can appear and then it would be fair to state that there would be the same problems that composing design patterns for sequential programming plus concurrent complexity.

*Reusability*, the concurrent design pattern implementations made in the *Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms [12]* paper were all partially reusable, using the methodology of: aspectize a general design pattern implementation and then an aspect that extends it to achieve the concrete needs.

### 2.1.3 Distributed patterns

Distributed computing refers to the process in which different computations take place and run simultaneously on two or more computers that are connected over a network, then design patterns for distributed programming are definitions of solutions to problems that occurs on applications that have different computational parts over a network. This subsection presents two examples of OO distributed patterns and AOP on distributed patterns. These design patterns have commonly been defined for specific technology by the research group of an organization let say JavaEE [29] by SUN, CORBA [34] by OMG or .NET [28] by Microsoft, so it is not hard to find that a defined pattern is dependant of the technology that uses. Let it be illustrated by two examples: The Busines delegate, a J2EE pattern, and Distributed Callback, a CORBA pattern.

**Ex**: *Business delegate, Java EE [5] The problem* a client needs to have access to business services, but a tied API specific access will give the client more problems than benefits, problems like error handling, object binding, API evolution, high coupling from the client application to the Business service application, there is no caching mechanism and high network traffic and as a consequence low performance.

*The solution,* a Business service abstraction is made and renamed Business delegate(BD), it is in charge of doing the object binding over the network for the Business service, now the client access the Business delegate, that hides the implementation of the Business service and as a consequence the evolution has less impact on the client's side, the business delegate also intercepts business service exceptions as java.rmi.Remote exceptions and encapsulates

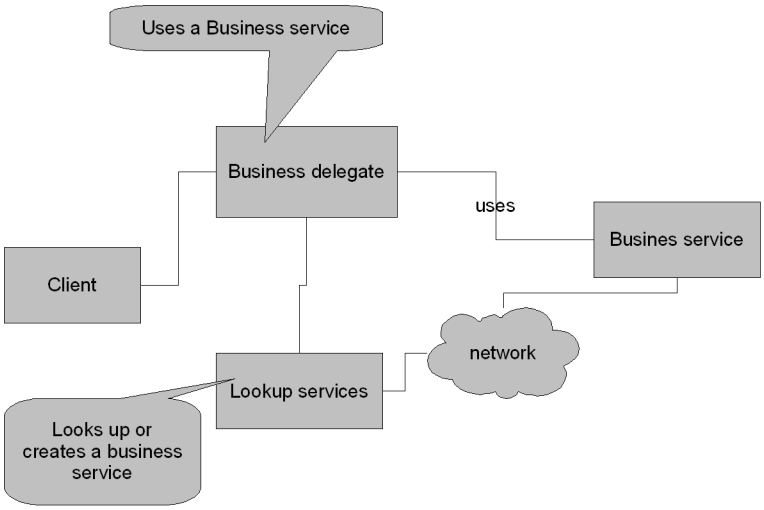


Figure 2.5: Business delegate pattern architecture.

them in application level exceptions, that are user friendly to handle. Figure Figure 2.5 shows the architecture of the business delegate pattern.

*The benefits,* some of the benefits that this pattern introduces are: application level error handling, object binding over network, lower coupling with Business service and Caching.

Another approach to manage distribution is defined by the Object Management Group in CORBA [34], the implementation is commonly known as a midleware that communicates objects definitions in an Interface Definition Language(IDL) between networks and/or compilers. Because it has been applied many times, there are patterns for this definition as well, the following is one of them.

**Ex**: *Distributed Callback, CORBA pattern[30]* The distributed callback pattern is likely one of the most widely used patterns in the CORBA Design Patterns book [30] . Using a callback, a client can request information from the server and continue processing while the server handles the request. In a normal call the client is reduced to "busy waiting" while the server processes the request. To facilitate the callback, the client passes itself in as one of the arguments in the call. When the server has finished processing it calls a method on its client reference passing the results to the client. This CORBA design pattern is one of the simplest ways to achieve a measure of parallel processing in your distributed object application. This pattern could be seen as a future value for a client, the distributed version of the future call concurrent pattern.

The previous examples have some **dependencies** on: *Frameworks*, they give many non-functional requirements such as caching, persistence, error handling, objects binding among others, such concerns are often removed from the design pattern because they are given by the framework, so the definition of a distributed design pattern is more or less the same than a sequential local design pattern, there is no need to specify or mention the behaviour that the framework has. *Language,* in many cases language dependency was found now that on the specification of a pattern the uses of a language specific functionality was needed or desired, such as the IDL of the *OMG's CORBA technology*, or in some other cases the problems they intent to solve came out of the technology itself, in example the *Business delegate* pattern which between one of the problem that solves there is the error handling of the EJB [29] classes.

### AOP and distributed patterns

Research to support distributed patterns have been done by the computer science department of "L'Ecole de Mines de Nantes", an approach based on the AWED(Aspect With Explicit Distribution) language and its implementation the DJasCo language. They define the concept of *invasive patterns*, what means that in order to apply one pattern some data or tasks have to be done before.

This research [7] has achieved a definition of a pattern language, the foundation of this language are well known distributed patterns like gather, farm and pipeline and sequences of patterns, the pointcuts and advices can be applied to one or many hosts and the sequences of the patterns refers to the synchronization of the aspectized patterns. In that approach, some advices are executed at source pointcuts first and their synchronization done before applying a target advice in a target pointcut

The example used in this study was JBoss cache [21], the Cache is done in order to increase performance in distributed applications, the JBoss Cache framework architecture can be defined in patterns like pipeline, gather and farmer. JBoss Cache abstract architecture is shown in 2.6, its substructures are: a pipeline pattern for transaction control (whose parts are represented by the dashed circles) and farm patterns (dotted circles) for replication actions [7]. Their analysis of the JBoss Cache code showed that the TreeChache Class(the main class for the Cache concern) has 2802 LOC and 280 of those related to transactions and 137 LOC of 5099 of package that is related to transaction. The scattered is later on, modularized in an implementation of this new pattern sequence definition.

Some results of this research are: the definition of the grammar for the pattern language, the definition of "invasive patterns", also with the study done called "Patterns-based grip programming with aspects" the evaluation of the language vs the NAS Grid language gave AWED a better and simpler form of constructing groups of host and running sequences intentions(Synchronization of aspects). The AWED implementation showed a small overhead of the performance, but comparing the 93 LOC of AWED against the 3993 of the

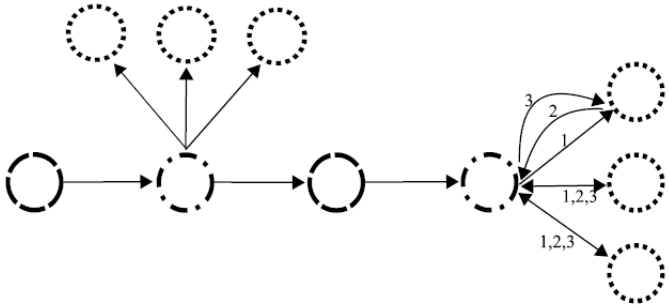


Figure 2.6: JBoss Cache, architectural pattern [7].

experiment realized and futures optimizations to the AWED language performance, it is fair to say that AWED is a really good option to start the research of AOP support for distributed patterns.

### 2.1.4 AOP and pattern composition

This subsections first presents the problems encountered when composing aspects and then describes pattern composition on AOP approaches.

When many aspects are applied to a large scale software application, there is need for composing aspects, this composition can be done by means of precedence, functional transformations [26] and kernel operators and visibility of structural changes [42] . The precedence mechanism is one of the most used, because it comes with the AspectJ implementation, in order to use it in AspectJ it is just necessary to declare an aspect that declares the precedence. The following code exemplifies this case.

```
1 public aspect Precedences {
      declare precedence : Aspect1, Aspect2.Aspect3, Aspect4;
3 }
```

What means that whenever there is a pointcut collision the aspects are going to be applied in the defined order. This solution can get us to a circularity problem or to not being able to express all composition orders, due to just order precedence is going to be applied to all contexts, limiting the composition.

Functional composition [26], declares that the result of weaving a sequence of aspects into a program equals the weaving of advice in weaving order into the program, this approach gives us quantification into unbounded base code, i.e. using wildcard patterns this approach will still be applicable.

On the other hand, some other solution to solve this conflict between aspects was proposed defining kernel operators and the visibility of structural changes [42]. On the first approach used in Reflex [44], operators are defined in order to deal with the composition of aspects. In the study done Erick Tanter defines sequence and wrap operators as low level operators that can be seen as a precedence operator and a cflow mechanism, then, this mechanism is extended to proceed high level operators that depend upon the execution context of the program. As an example, he defines a DWrap that is only applied if the extended Wrap is not active, so it depends on it.

The second approach, visibility of structural changes, describes the code that is viewed by the cuts, which means that if any changes are done the cuts are still applied by default to the original code, or in the case of a declarative visibility definition, the visibility of the code will include the changes made by other aspects.

Whenever multiple patterns have to be related to be implemented there is a pattern composition, AOP could have some good results on such composition due to locality, reusability and less dependency inversion of the pattern implementation code. Simon Dernier, Herv Albin Amiot and Pierre Cointe [13] did a research on this expression and composition of patterns with aspects.

Their study was based on design properties to evaluate aspectization of design patterns, locality, decoupling, traceability and understanding [13]. A result on these properties is "*Locality and decoupling promote traceability and evolution and partially easy understanding*" [13], meaning that whenever an aspectization of a design pattern can be done without coupling with the base code and in a localized part, their code is more easily understand and by means of consequence more easily maintain, traced and evolved.

Problems composing patterns can emerge if there is a strong coupling between patterns [10]. There are needs for composition configuration, ordering aspects and stronger pointcut language. The first need will make the aspects dependent on an order, the role of deciding the order and the constraints to take into account are still need to be defined, the second need is because of fragile pointcut definition on the AspectJ actual implementation, there is not enough expressiveness, the expression of the poincut is based on the name of the classes and methods and not in the functionality where an aspect wants to be inserted. Other results from composition of design patterns showed that the results from composing aspectized design patterns depends upon the patterns involved and also the requirements of the application, It has to be globally reasoned in order to take into account the context of the whole system implementation, its dependencies and the design options when implementing the design patterns [9].

### 2.1.5 Lessons learned

The full 23 GoF design pattern have been successfully implemented using AOP, the results from this research depend on the metrics and mechanism used to measure, there are some differences between results of reusability and base code obliviousness. When implementing concurrent pattern, the AOP language has to have mechanism to manage the concurrency, like the java reserved work **synchronized** or concurrent libraries, but the need for managing the concurrent has to be taken part of the approach, by extending a language or using a library. On the other hand, the research done using AOP to implement distributed patterns have shown the benefits of its use. AOP has been research in order to implement patterns, these implementation have encountered a critical need for pattern composition, this composition has to take into account dependencies and coupling between patterns.

Looking at the pattern implementations, many of the pattern protocols(the reusable part of the pattern implementations) are basically aspects that define mechanism to communicate the roles of the patterns using interfaces, they also define abstract methods that are implemented at the application of the aspect that introduces the pattern protocol to the specific application(this is not reusable), 75% of those "introductions" are inter type declaration over objects, so it is proven that those type of declarations support pattern desinition in an AOP way.

AOP has not been used to implement architectural patterns, neither for patterns defined by the the technologies presented before, it appears that current research on AOP on distributed patterns is not enough to measure whether AOP can or not support them, and if it supports, which support it would be.

## 2.2 AOP and distribution

This subsection presents AOP, how AOP has been used in frameworks for distribution and distributed AOP. First AOP is briefly explained, subsection 2.2.1 describes parallel and distributed programming, it is the distributed AOP context. Section 2.2.2 presents sequential AOP and frameworks for distribution and section 2.2.3 describes distributed AOP and current language approaches.

Aspect Oriented Programming(AOP) aims to improve code modularization by doing separation of concerns, concerns like login, persistence, and security, just to to mention some. The Figure 2.7 shows one example of how an aspect oriented application could be structured.

The AOP architecture defined in Figure 2.7 shows in the middle a base OO(object Oriented) code and two concerns, security(in example authentication) and persistence, the interception of the OO model and the concern implementation are oblivious from the base code[23].

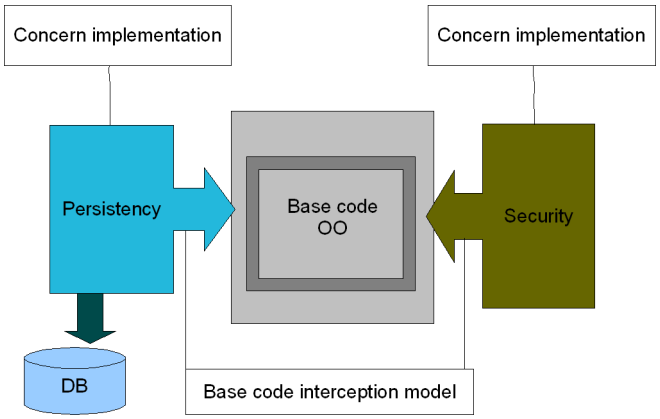


Figure 2.7: Aspect oriented program architecture example

The base code is intercepted by pointcuts, in other words the pointcuts are localizations of code or behaviour to be intercepted in order to perform some required concern at that point(s), the introduced code or behaviour is called advice, this is the implemented behaviour of the required concern at the pointcut.

### 2.2.1 Parallel and distributed programming

Parallel programming refers to the application programmed in order to conduct computation in different processors, this computation can take place into different processor of a single machine or different machines over a network(Distributed computing). Whether the computation is done on a single or many machines the program have to take into account the synchronization between the processors, the communication and the complexity of handling the computation of different processors or machines.

In order to handle the communication of different machines over a network, there are several proposes, some of these are: remote method invocation(RMI) and the .NET framework. Because of the need to make remote available objects, there are some performance implications. A study of performance implications of the implementation of design pattern using RMI and the counterpart dotnet showed an overhead of up to 40 percent, that study was made of the implementation of three design patterns over the framework before mentioned, the patterns implementred were Facade, Command[17] and Combined Command.

The table of the Figure 2.8 is taken from the paper *Performance Implications of Design Pattern Usage in Distributed Applications Case Studies in J2EE and .NET* [17]. The performance is not only decreased by means of the network delay by also the conversion and deconversion of the objects in order to made them available over the network.

**Table 1: Object Sizes After Serialization**

| Design | Object Type | Size in bytes | | |
|---|---|---|---|---|
| | | RMI | Binary | SOAP |
| Façade | Small | 13 | 30 | 479 |
| | Medium | 241 | 386 | 1463 |
| | Large | 43613 | 65714 | 406773 |
| Command | Small | 125 | 169 | 702 |
| | Medium | 326 | 474 | 1712 |
| | Large | 43698 | 65802 | 407032 |
| Combined | Small | 179 | 263 | 987 |
| Command | Medium | 343 | 512 | 2001 |
| | Large | 43715 | 65840 | 407328 |

Figure 2.8: Object size after serialization

This overhead of the performance can be reduced using some mechanism of code mobility[2]. Two of these mechanisms are : First mechanism, called *code aggregation*, encapsulates the call of many methods into one entity that is transferred into the server or client side of the application. The second mechanism, called *server forwarding*[2] , it relies in faster connections that server to server have. The previous two mechanisms presented have some problems to solve, the first is found if passing an object that has a large hierarchy it is possible to introduce overhead instead of solving it and the second problem refers to mutual calls and object sharing, if the code of a method is optimized to be passed to the server it is also possible that the server can have the need to call the client, then this kind of optimization is not applicable.

Those approaches attempt to solve problems of communication overhead due to distribution, but as stated before there are problems arising from the complexity of synchronizing different computations, for those problems some approaches like libraries and code generators have been proposed. SkeTo[27] is a C++ library based on constructed algorithmic(these are algorithmic derived from specification by calculation)[47] [27]. The previous attempts to deal with some of the numerous obstacles of distributed or parallel programming can be defined only in a static way, it definitely reduces the complexity by code generation, but it exposes the programmer to the generated code and relies on the programmer knowledge in order to include change to improve performance. Nevertheless the foundations of this approach are used worldwide, a basic conceptual structure used to solve a complex issue or better know as **frameworks**.

### 2.2.2 Sequential AOP and Frameworks for distribution

A software framework is a re-usable design for a software system/subsystem. A software framework may include support programs, code libraries, a scripting language, or other

software to help develop and glue together the different components of a application. This subsection presents the current impact of AOP in frameworks, how it has been used on them, how it is integrated with them, JBoss AOP proposal, differences with distributed AOP and use cases. First case, the IBM Websphere[20] application server is refactored using AspectJ. Second case, JBoss AOP[22] is an AOP approach that is tightly with JBoss application server.

### JBoss AOP

JBoss application is a J2EE platform for developing and deploying enterprise Java applications, Web applications, and Portals, it provides server clustering, caching, and persistence. JBoss AOP[22] is a 100% Pure Java aspected oriented framework usable in any programming environment or as tightly integrated with JBoss application server.

JBoss AOP is not only a framework, but also a prepackaged set of aspects that are applied via annotations, pointcut expressions, or dynamically at runtime. Some of these include caching, asynchronous communication, transactions and security. Links between aspects and pointcuts are defined in extended Markup Language (XML). Aspects are defined in two flavours, interceptors and normal aspects. The interceptors are aspects with a single method(advice) called invoke, the normal aspects are defined as a normal class and the defined methods are the advices, then the link is made from the method of the pointcut to the method of the aspect. Poincuts are defined in regular expressions of wildcards over methods, class names and packages. JBoss AOP permits to do hot deployment of aspects, introductions of interfaces to classes where the behaviour is later implemented in mixing classes. The Visualization of an aspect is the first code box and the second is the binding between the aspects and the base code, the concurrent **OneWay concurrent pattern implemented using JBossAOP:**

```
public OnewayAspect
{
    private static class Task implements Runnable
    {
        private MethodInvocation invocation;

        public Task(MethodInvocation invocation)
        {
            this.invocation = invocation;
        }

        public void run()
        {
            try { invocation.invokeNext(); }
            catch (Throwable ignore) { }
        }
    }

    public Object oneway(MethodInvocation invocation) throws Throwable
    {
        MethodInvocation copy = invocation.copy();
        Thread t = new Thread(new Task(copy));
        t.setDaemon(false);
        t.start();
        return null;
    }
}
```

```
<aop>
    <aspect class="org.jboss.aspects.OnewayAspect"/>
    <bind pointcut="execution(void *->@org.jboss.Oneway(..))">
```

```
        <advice name="oneway" aspect="org.jboss.aspects.OnewayAspect"/>
    </bind>
</aop>
```

The previous code shows the way of declaring and binding an aspect, but it leaves outside the instantiation model of JBoss AOP, it is possible to create a bind between the aspect and the pointcuts, it is possible to declare instantiation defining the scope of the aspect, it can be per java virtual machine, per jointpoint and per instance. Even when this AOP approach is one of the most useful, used and well defined that I could found it still not suffice the need for distributed AOP, let me explain. First all mechanism previously defined are applied into the base code in a single machine, even when applied in the JBoss application server its domain still a local domain, there are no remote pointcuts, no remote advices and not even the notion of distributed aspects, then it is fair to state that only when those mechanism are included in the aspect model of JBoss AOP this approach can be included in the Distributed Aspect Oriented Languages definition, the following subsection presents some of the distributed AOP language approaches.

#### Websphere case

One of the biggest use of AOP in a framework was done by Adrian Colyer and Andrew Clement, the task was to separate the support for Enterprise JavaBeans(EJB) from the rest of the websphere application server[11], this application contains approximately 15.500 source files and many concerns are implemented, one of them is the EJB container. The research, first conducted refactoring in the OO way, then the AOP counterpart was also implemented, the result of this approach was a smaller solution for the AOP approach, but they also encountered need for visibility of private state or methods to the aspect, then they could refactored or monitored, this is probably why AspectJ[6] have now privileged aspects included. The refactoring achieved better modularity but also found problems as for example using reflection in the EJB container original code would result in an invisible call for the AspectJ pointcut model.

An interesting result of this large scale refactorization to the websphere framework is that 74% of the declarations inside the refactorization were inter-type declarations, and only 26% advice [11]. Each piece of advice affected only a single joinpoint, this is a common result to the one achieve in the implemetation of sequential design patterns [19] by Jan Hannemann and Gregor Kiczales.

### 2.2.3 Distributed AOP

Distributed Aspect Oriented Programming has to take into account three main concepts [44], *distributed cut* that describes execution of methods on one or many hosts. *Distributed action*, the behaviour could be executed at a remote host, it can be done at the same host that the cut was match or any other wanted host. *Distributed binding*: the specification

of the binding between the cut and the action of an aspect may be done in any host. The following language approaches match the previous concepts.

### DjCutter [33]

DjCutter is a distributed aspect oriented language, that extends AspectJ [6], it introduces the concept of host in the pointcuts, this language uses an aspect server in order to run the advices, it is in this aspect server host where the executions take place, than all the predefined fields can be accessed by many advices at the aspect server, the *hosts* pointcut predicate can define several hosts to match by separating them with commas, there is also the *gethost* predicate in order to do reflection of identifier of the matched host, it is also possible to call aspect's methods by doing a Casting with a logical name and a the library of DJCutter.

### AWED(Aspects With Explicit Distribution)

AWED was proposed by The Ecole des Mines de Nantes in conjunction with the Vrije Universiteit Brussel, this language is design to make AOP explicitly distributed not only on the remote pointcuts but also remote advices and remote aspects. The language can locate one or many hosts( by grouping them), the execution of the advices can be synchronous or asynchronous between some other features that will be explained. AWED language is going to be explained and extended(Chapter 4) for that reason the focus on the implementation level called *DJAsCO* on this section.

AWED's implementation was made over Java Aspects and Components(JAsCo)[41] , this last is an aspect oriented programming language, it has a high reusability of aspects and a strong definition for aspects composition. The reusability is achieved by separating the implementation of the aspect and the definition of the pointcut, a connector is defined in order to define where the aspect has to be introduced, it is also possible to define a method of an aspect as *refinable* what means that the implementation has to be done on the connector code, achieving partial reusability of the aspect. On the other hand, combination strategies or precedence can be used as mechanism to achieve aspect compositions.

*AWED and remote pointcuts,* pointcuts are evaluated on every host but then pointcuts conditions filter where pointcuts occur, the filters separate groups, localhost or defines a specific host. On the pointcuts it is also possible to get information or to pass it to the matched objects, the pointcuts can be combined using logical operators.

```
class Logs {

    hook log {

        starLog(startmethod(..args1)) {
            execution(startmethod) && joinpointhost("LOG") && executionhost(localhost);
        }
        before {
            System.out.println("logJ: " +
                (String) thisJoinPoint.getArgumentsArray()[0]);
        }
    }
```

```
}
```

The code before defines a Log aspect that logs(prints out) the first argument of the matched method on the localhost before the method is executed on the hosts declared to belong to the LOG group., the following code defines the connector that defines the specific pointcut case(that matches all methods).

```
static connector LogCon {

    test.Logs.log hook0 =
     new test.Logs.log(* *.*(*));

}
```

The separation of connector and aspects makes also possible to dynamically deploy or undeploy aspects, this is achieved using the Jutta (Just-in-time combined aspect compilation) system, that is a real time weaver, whenever a new connector is deployed the Jutta system weaves again the code and the aspects are applied. In other words there is the possibility of deployment of aspects at run time.

*Advices,* the advices are those of AspectJ plus some extra needed for distribution. For example the **proceed** statement (that continues with the original call) has a counterpart **localproceed** that continues with execution on the matched joinpoint host rather than in the advice host. The advice execution can be also synchronous or asynchronous, giving the possibility for parallel execution of computation also, so futures can be used also, as an example an extension of the previous code.

```
class Logs {
    @DistributedAdvice (executionType = DistributedAdvice.Type.ASYNCEX)
    hook log {

        starLog(startmethod(..args1)) {
            call(startmethod) && joinpointhost(localhost);
        }
         around() {
            String ilog= "logJ: " +
                (String) thisJoinPoint.getArgumentsArray()[0]);
            return localproceed ilog;
        }
    }
}
```

The code showed before returns as String the first argument of the intercepted method in an asynchronous way, the returned object is updated when the execution of the code in the joinpoint has finished then the String is returned locally to the executing code

The AWED language benefits from the JAsCo implementation. When several aspects want to be applied, several advance composition techniques can be applied. For example, a precedence of aspects or defining combination strategies between aspects.

```
public class AdminCombinationStrategy implements CombinationStrategy
 {
 private Object validationAspect, loggin;

 public AdminCombinationStrategy(Object a,Object b) {
      validationAspect = a;
      loggin = b;
```

```
8   }

10  public HookList validateCombinations(HookList hlist) {

12          if (hlist.contains(loggin)&&!hlist.contains(validationAspect)) {
                hlist.remove(loggin);
14          }
                return hlist;
16      }
    }
```

```
    static connector ShoppingProccessControl {
2
        perobject aspects.Validation.ProductOrder validation =
4           gunew aspects.Validation.ProductOrder(* *.ShoppingBasket.*ProductOrder(*))
        {
6       ...
          };
8         log.around();
          validation.around();
10        logInside.around();
          addCombinationStrategy(new AdminCombinationStrategy( validation,logInside));
12  }
```

This code defines a precedence between the aspects and also a combination strategy, the first code box defines a combination strategy and the second defines an execution precedence and then adds the combinationStrategy previously defined. This approach clearly solves composition of aspects made by one connector and having a completed view of the aspects and their implication, but there are now problems on the composition of connectors and/or in aspects deployed in different machines. For example, mutual exclusion of aspects in different machines could cancel their behaviour.

AWED's has been applied to mane use cases, I briefly describe two of them, the first Aspect Oriented Toll System(ATOLL)[31] , the implementation of a charging system for motorways usage using Global position System(GPS) as the input, the AWED implementation prove to be more flexible to variability of the implementation and also more modularized, it managed the distribution using AWED's distributed definition.

The second case of use was Web Service Management Layer(WSML)[32] , this manager is in charge of *error handling*(i.e. need for rollback in an unfinished transaction), *performance optimization*(i.e. redirect service calls to avoid bottlenecks) and *business evolution*(i.e. new or improved requirements of service). The separation of the clients core system and web services decrease the dependency, the AWED language was extended with chains of a/synchronous executed remote advices, in order the composition of Web Services and error handling. The result of this application of AWED was increase modularization of crosscutting web services compositions.

### ReflexD [44]

ReflexD is an approach by the Universidad de Chile, it is build over Reflex [43] and use of Remote Consistency Framework to handle the distributed calls. Reflex can be briefly described as a library for structural and behavioural reflection in Java[44]. It also implies that changes on metaobjects affect the actual state of the objects that it describes.

ReflexD defines a **hookset** that is in charge of selecting the class type and method to be intercepted, then a **link** is used to determine what action to take, this link is used to associate metaobjects created in ReflexD with the methods matched by the hookset extended, this last contains the notion of host introduced by a RHost.

```
public class Logger{
    public void log (Object aThis , RHost aHost ){
    i f ( "log".equals(aHost.getProperties().get("type")))
// log
    else
// do not log
    }
}...

    RHost host = RHosts.get ( "172.1.2.3:5555", "log");
    Link linklog = Links.get( methodOne,
    new MODefinition.Class( "ClassA", new ExecHost(host));
    trace.setCall("Logger" , "log" , this, Parameter.HOST) ;
```

The previous code is an example of how to use ReflexD, it creates an aspect Logger that logs whenever the host of execution is done at the log type host, the last part of the code creates a alink with the execution of the method methodOne at class ClassA in host 172.1.2.3:5555 that belongs to the type log. ReflexD permits to define remote cuts by introducing the Rhost class, it also allows to describe where to make the advice is going to be executed; ExecHost.THIS, at the link host. ExecHost.APP at the pointcut host or new ExecHost(host) any arbitrary hosts as showed in the previous example.

Distributed AOP in ReflexD is achieved over the Remote Consistency framework(RCF), this framework maintains consistency between remote calls and remote objects. The consistent declaration of a class is done by implementing the Consistent interface, then a Distributed policy has to be declared. The principal components of the RCF are: (1) the consistency metaobject and (2) the consistency manager. The consistency metaobject is the one in charge of intercepting the method calls in a consistent object. Once it has taken the control of the execution, it connects to a consistency manager and through RMI invokes a method to notify the method call occurrence. The role of the consistency manager is to be a remote access point to the local host and therefore, a net bridge for notifications of method calls. Notice that this bridge is bi-directional as each host has its own consistency manager.

ReflexD was build over Reflex, so it has the open facilities to extend as reflex does, the openness refers to how ReflexD can be extended o changed in order to introduce new functionality or improve them. Reflex uses a declarative composition in order to be extended via a Syntax Definition Formalism(SDF), the new statements can be defined as also new operators.

**Case of use,** The N-Queens problem[45] consists in finding the number of ways N queens can be positioned on a $NxN$ board without attacking each other, the complexity of this problems is $O(n^n)$ making it impossible to solve. The problem was then divided into subproblems(tasks) performed by workers and the results of those subproblems are joint

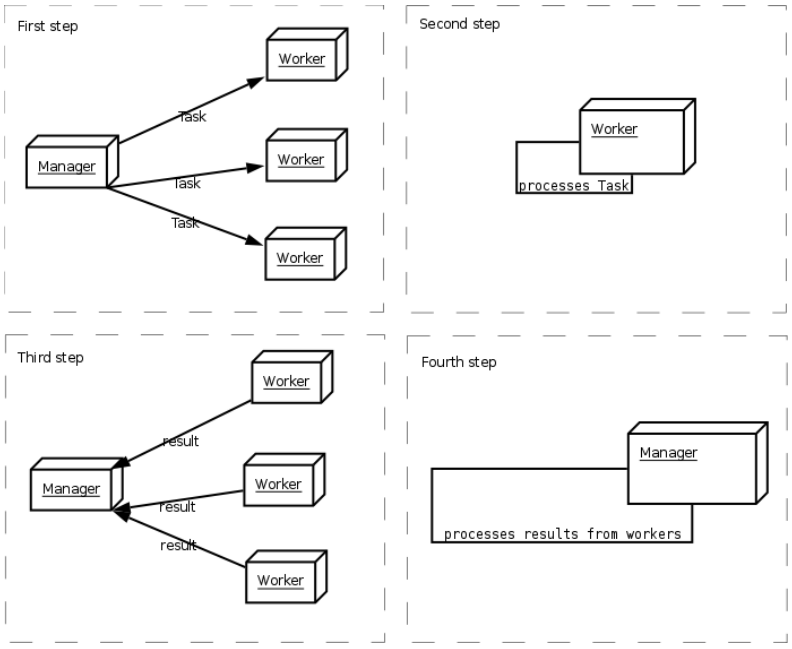


Figure 2.9: Gather, Farmer pattern applied to N-Queens problem.

by a single manager, this is a composition of the farmer and gather distributed patterns, the architecture is showed in Figure 2.9 .

The Figure 2.9 describes the necessary steps in order to solve this problem, ReflexD was used in order to create an independent module to manage the distribution of tasks in different workers across the network. The RFC of ReflexD is used to manage the consistency and the communication between the workers and the manager, the result is a better transparent modularized version of the algorithmic with increased performance than the sequential counterpart.

### 2.2.4 Lessons learned

The context of distribution and concurrent programming is more complex than sequential one, for this reason, frameworks were created in order to manage this complexity, AOP showed improvements in the implementation of concerns in frameworks also. An aspect oriented language applied to a framework for distribution is not distributed aspect oriented. The performance is a critical part of distributed and concurrent programming, it increases complexity because it has to be inside the implementation of distributed aspect oriented languages, the management of communication between distributed objects and remote binding, have to be taken into account, irregular topologies have not been taken into

account on any of the distributed AOP approaches.

## 2.3 Conclusions

AOP has showed improvements of modularization, in sequential and concurrent pattern implementations. The modularization and reusability of pattern implementations are at least, partially achieved. On the context of concurrency, the synchronization mechanism is fundamental, no matter if it comes from the base language of the AOP language, or from libraries. Composition of patterns is a difficult task, it has to be carefully managed, in order not to have aspect interference. Distributed AOP has not managed to implement reusable distributed architectural patterns, neither it has taken into account the irregular topologies in which they are deployed and used.

The refactorization of large middleware as Websphere, has proven the power of AOP of solving real life crosscutting concerns. The research done in concurrent and distributed application of AOP, has shown improvements to the modularization of crosscutting concerns, motivating this research to continue.

The result of applying AOP to frameworks for distribution is not distributed AOP. The complexity of distributed programming is due to communication mechanisms, irregular topologies, and synchronization of the processes on different machines, so distributed AOP is proposed to manage at language level previous mention difficulties, then developers are focused on the business problem and not on the distributed concern. One of the aims of AOP is to make code more understandable. In order to make distributed patterns implementation more comprehensible distributed AOP provides support at language level for those needs. AWED and ReflexD are the languages that manage better distribution concerns.

# Chapter 3

# Pattern Implementation with distributed AOP

This chapter presents the implementation of concurrent and distributed patterns using AWED's system/implementation DJAsCo, the section 3.1 presents the concurrent pattern implementations, section 3.2 the distributed pattern implementation and in section 3.3 the conclusions.

## 3.1 Concurrent Pattern implementation

Concurrency occurs at the systems where various computational processes execute at the same time, these processes can be interacting with each other. The concurrent pattens are well defined programming mechanism that have proven to solve efficiently needs for concurrency, i.e. better performance in distributed programming and parallel programing. This section shows the implementation of some concurrent patterns, their descriptions, the problems found, some possible solutions and the leasons learned from their implementation.

### 3.1.1 Synchronization mechanism

#### Pattern description

This mechanism better known as monitor is done in order to restrict the access of two or more processes to a single resource, a lock is used in order to exclude access from multiple processes, only one process has the lock at a given moment in time and when the lock is released the processes compete to adquire the lock.

#### Solution using DJAsCo

```
class Synchronization {
    HashMap annotationLocks = new HashMap(); //<String, Object>

    hook synchronizedUsingSharedLock {
        synchronizedUsingSharedLock(method(..args)) {
            execution(method) && joinpointhost("synchronized");
```

```
        }
8       around() {
            synchronized(this){
10              return proceed();
            }
12      }
    }
14
    hook synchronizedUsingCapturedLock {
16      synchronizedUsingCapturedLock(method(..args)) {
            execution(method) && joinpointhost("synchronized");
18      }
        around() {
20          synchronized(thisJoinPoint.getCalledObject()){
                return proceed();
22          }
        }
24  }

26  hook synchronizedUsingAnnotations {
        synchronizedUsingAnnotations(method(..args)) {
28          execution(method) && joinpointhost("synchronized");
        }
30      around() {
            Object lock = thisJoinPoint.getCalledObject();
32          Annotation[] annotations=thisJoinPoint.getAnnotations();
            Synchronized parameters=(Synchronized)annotations[0];
34          if(! parameters.id().equals("default"))
                lock = global.mapId2Lock(parameters.id());
36
            synchronized(lock){
38              return proceed();
            }
40      }
    }
42
    protected synchronized Object mapId2Lock(String id){
44      Object ob;
        ob = annotationLocks.get(id);
46      if(ob == null) annotationLocks.put(id, new Object());
        return ob;
48  }
}
```

This aspect extends the synchronization mechanism of java in order to define new rules to synchronize, these rules refer to the lock used to synchronize, the first is the aspect object itself, the second is the matched object at the pointcut and the last is an object created from an annotation defined in the matched method. The implementation of this mechanism was straight forward due to Java synchronization level support, no problems were found.

### 3.1.2 One way pattern

#### Pattern description

This pattern improves performance by running method that does not return a value in a different thread. A thread is created in order to run a single void method and then it is destroyed or not used any more. The application of this method using aspect can be done by extending or instantiating a reusable One Way Aspect or by using metadata(i.e. annotations) in order to mark the methods where the pattern should be applied.

One example of what this pattern use is described in the following diagram, where there is a class that Class that contains three void messages, the user calls the three messages

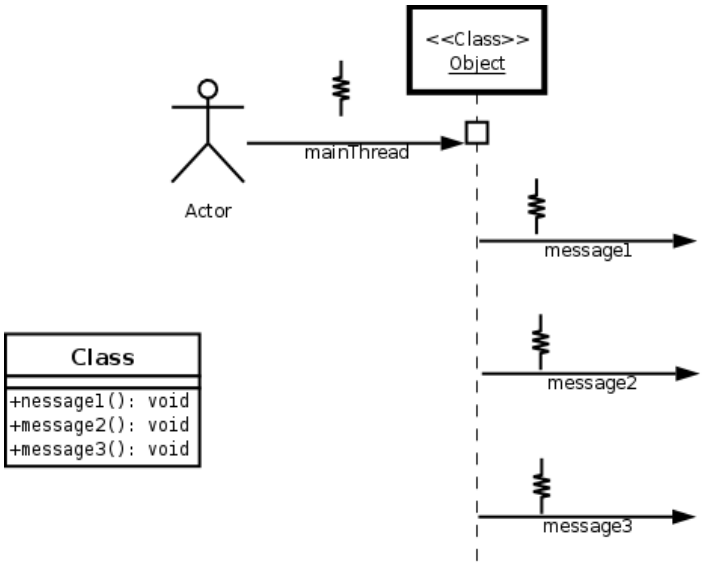


Figure 3.1: One Way pattern example

on and object Object of type Class, then a thread is created on order to run each of the methods' calls. Figure 3.1 shows this example.

### Solution using DJAsCo

```
1  class OneWay {
       protected WeakHashMap threads = new WeakHashMap();
3
       hook onewayMethodExecution {
5
           onewayMethodExecution(method(..args), callable(..args)) {
7              execution(method)&&!cflow(callable) && joinpointhost(localhost)&& executionhost(localhost);
           }
9
           around(){
11             //Setting the call Builder Object
               RunnableMethodExecution methodThread = new RunnableMethodExecution();
13             methodThread.setObject(thisJoinPoint.getCalledObject());
               methodThread.setParameters(thisJoinPoint.getArgumentsArray());
15             methodThread.setMethodSignature(thisJoinPoint.getName());
               methodThread.setParameterTypes(thisJoinPoint.getFormalArgumentTypes());
17             Thread t = new Thread(methodThread);
               t.start();
19             return null;
           }
21     }
   ...
23 }
```

The previous code defines a single joinpoint with two input methods, the callable method is used as wrapper in order not to fall into an infinite loop because of introspective calls and the pointcut of the aspect. All necessary parameters in order to execute the call are set in the RunnableMethodExecution and then called by introspection, the original behaviour is canceled using the return null statement.

### Problems encountered

The main problem encountered here was that in order to execute the pointcut method in a new thread class specification, the aspect language has to be fully integrated with the base code language, in this particular example, the proceed() call cannot be executed in java statements defined inside the advice body like at an inner class, but it has to be call from the first level of the advice body.

### Possible solutions

I used an instrospective thread call, it is stored in a object called RunnableMethodExecution and executed when needed, the initial behaviour of the pointcut is canceled in order not to duplicate behaviour.

Complete language integration of JAsCo [40] with Java would solve this problem, as AspectJ has with Java, the problems behind this proposal in JasCo are not part of the domain of this research, it is just mentioned to donate ideas and motivation to implement this complete language integration.

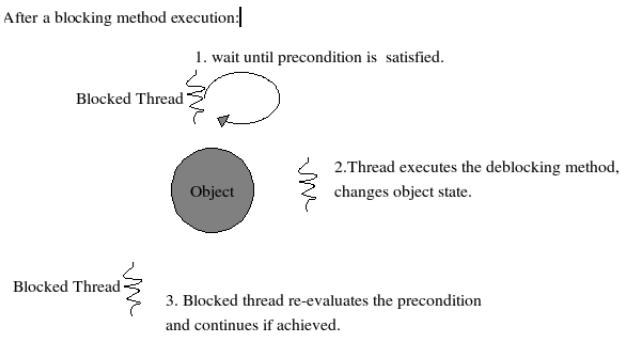


Figure 3.2: Waiting Guards pattern.

### 3.1.3 Waiting Guards

#### Pattern description

In the case that method executions depend on preconditions, in many cases this precondition is link to the state of the object, two different path can be taken, first is to rise an exception or to wait until the preconditions are satisfied. In concurrent programming the state of the object can be changed by another thread accessing the object, the Waiting Guard pattern defines that a thread that executes methods with not satisfied preconditions has to block until some action changes the state of the object and triggers the precondition evaluation again or at least a given amount of time to re-evaluate.

In order to apply this pattern three things have to be defined for each pattern application, a blocking method that could send the object to a non-satisfying state, a deblocking method that could return the object to a satisfying state and the definition of the precondition to be satisfied. The Figure 3.2 presents the pattern after the execution of a blocking method.

#### Solution using DJAsCo

```
class WaitingGuard {

    hook deblockingOperations {
        deblockingOperations(method(..args)) {
            execution(method) && joinpointhost(localhost) && executionhost(localhost);
        }
        after() {
            Object jpObject=thisJoinPoint.getCalledObject();
            synchronized(jpObject){
                jpObject.notifyAll();
            }
        }
    }

    hook blockingOperations {

        blockingOperations(method(..args)) {
            execution(method) && joinpointhost(localhost)&& executionhost(localhost) ;
        }
```

```
20
        before() {
22          Object jpObject=thisJoinPoint.getCalledObject();
            Object[] params=thisJoinPoint.getArgumentsArray();
24          try{
                synchronized(jpObject){
26                  while(! preCondition(jpObject, params)){
                        jpObject.wait(getWaitingTime());
28                  }
                }
30          }catch(InterruptedException c){
                throw new Exception(c);
32          }

34      }
        public refinable synchronized boolean preCondition(Object ob, Object params ){
36          return false;
        };
38
        public refinable synchronized long getWaitingTime(){
40          return 1000;
        };
42  }
}
44 ...
//Apllication of the pattern to a Water tank, where the overflow is not permited.
46 static connector OverFlowWaitingGuardConnector {

48      waitingguards.WaitingGuard.deblockingOperations hookOverFlowDebloking =
            new waitingguards.WaitingGuard.deblockingOperations(* waitingguards.example.WaterTank.removeWater(*));
50
        waitingguards.WaitingGuard.blockingOperations hookOverFlowblock =
52          new waitingguards.WaitingGuard.blockingOperations(* waitingguards.example.WaterTank.addWater(*)){

54          public refinable synchronized long getWaitingTime(){
                return 1000;
56          }

58          public refinable synchronized boolean preCondition(Object targetObject, Object params){
                waitingguards.example.WaterTank wt = (waitingguards.example.WaterTank) targetObject;
60              Object[] parameters=(Object[]) params;
                Float amount = (Float) parameters[0];
62              return (wt.getCurrentVolume() + amount) <= wt.getCapacity();
            }
64
        };
66
        hookOverFlowDebloking.after();
68      hookOverFlowblock.before();
}
```

This code defines the two method advices, one to be executed before a blocking method is executed and the second when a deblocking method is executed, because of the "refinable" declaration of the last two methods, those can be implemented as needed by the specific connector application where these are to be defined(in example the second part of the code). The *precondition(..)* defines when a blocking method can be executed and the getWaitingTime() method for how long the thread should be waiting to re-evaluate the condition.

### Problems encountered

The use of Java support for synchronization and thread management was helpful for the implementation of this pattern, without those mechanisms the implementation of this pattern would have been a lot more complex, because the *synchronization* and *waiting* mechanism of the thread had to be implemented.

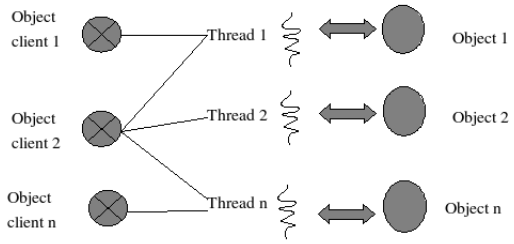


Figure 3.3: Active Object pattern

### Possible solutions

In order not to depend on the Java language concurrent features, a library could be implemented and reuse if wanted, but in my case I decide to reuse what was available.

## 3.1.4 Active object

### Pattern description

This pattern improves performance by running every object instance in a different thread, for every instance of a class there is a thread in charge of executing the calls to its methods.

The invocation and execution of methods are decoupled, in order to achieved this behaviour an association between a new Thread and an instance is created, then a Method scheduler retrieves every call to objects and dispatch then to the respective associated thread. Figure 3.3 represents the Active Object pattern.

### Solution using DJAsCo

```
1  class ActiveObject {
       protected WeakHashMap hash = new WeakHashMap();//<ActiveObject,MQScheduler>();
3
       hook create {
5          create(method(..args),callable(..args2) ) {
               execution(method) && !cflow(callable) && joinpointhost(localhost) && executionhostlocalhost) ;
7          }

9          boolean appliedAlready=false;

11         isApplicable() {
               return !appliedAlready;
13          }

15         before() {
               appliedAlready=true;
17             MQScheduler mqs = new MQScheduler(50);
               System.out.println("Instantiation of object: " + thisJoinPoint.getCalledObject().toString());
19             synchronized(this){
                   global.hash.put(thisJoinPoint.getCalledObject(), mqs);
21                 }
               (new Thread(mqs)).start();
23         }
       }
```

```
25
    hook callMeth {
27      callMeth(method(..args), callable(..args)) {
            execution(method)&&!cflow(callable) && joinpointhost(localhost)&& executionhost(localhost) ;
29      }

31      around(){
            Message ms = new Message();
33          Object msg;
            CallableMethodExecution callable = new CallableMethodExecution();
35          callable.setObject(thisJoinPoint.getCalledObject());
            callable.setParameters(thisJoinPoint.getArgumentsArray());
37          callable.setMethodSignature(thisJoinPoint.getName());
            callable.setParameterTypes(thisJoinPoint.getFormalArgumentTypes());
39          MethodRequest mr = new MethodRequest(callable,ms);
            sendToQueue(mr,thisJoinPoint.getCalledObject());
41          return waitForValue(mr, thisJoinPoint.getCalledObject());
        }
43
        public Object waitForValue(MethodRequest mr, Object activeObject)
45      {
            Object res = null;
47          try{
                res = mr.getResult().getValue();
49
            }catch(Throwable t){
51              t.printStackTrace();
            }
53          return res;
        }
55
        public synchronized void sendToQueue(MethodRequest mr, Object activeObject ){
57          ((MQScheduler) global.hash.get(activeObject)).insert(mr);
        }
59  }
}
```

The complexity of this pattern is the biggest of all implemented patterns, it has the need for a new introspective call executor that returns a value, also a data structure that associates the object with a thread, also a queue for the calls for each object and the concurrency of these executions.

The first part is the pointcut definition to associate the object with a thread, it is instantiated *perobject* but only once. The second pointcut is used whenever a call is done to the method of the previously associated objects, it is wrapped into a CallableMethodExecution(introspective method execution that returns the value) object and send to the queue of the object, finally it is returned when the result of that execution is done, the executions are done in the First In First Out(FIFO) way. The whole code is not included because of size limitations.

### Problems encountered

Two problems I found implementing this pattern, the first, in order to execute the pointcut method in a the thread of the object the arguments needed to perform an introspective execution had to be store in the wrapper, it is the same problem encountered in the One Way pattern implementation.

The second and more complex problem is that JAsCo pointcuts are not defined to match object creation, the following code illustrates the wanted pointcut definition that it is not yet available at the current version.

```
1 protected pointcut create(ActiveObject s) : execution(ActiveObject+.new(..)) && this(s);
```

**Possible solutions**

The first problem was solved defining an object that can be set to perform introspective calls and return the value, this is the wrapper for the method executions, that is linked to the object thread. Like I said in One Way section, this could be overcome by having cmplete integration of the aspect language and the base code language.

I found my way on the second problem by doing a *perobject* instantiation of the pointcut that associates the object with a thread, this association is done at the first call to a method of an object, but the pointcut is done to every call to every object, as seen this is not an optimized solution, because for every call to any method of the class to introduce this pattern there is a pointcut method that checks whether this object has been associated before or not. The obvious optimized solution is to take into account that a new(..) message is a class message that has to be part of the pointcut language definition.

### 3.1.5 Lessons Learned

In order to implement concurrent patterns using AOP there is the need for synchronization mechanism, concurrent library or base language support as in the case of the previous implementations, if Java language support was not present the implementation complexity of these patterns would have been increased, because of he need to manage concurrency.

It is good to have full integration with the language in order not to introduce overhead in a pattern that is meant to increase performance. The solution I defined in order to overcome this limitation may have introduce overhear by doing introspective executions, future performance tests have yet to be done in order to state whether this is true or not. On the other hand object instantiation is important, pointcut languages need to support it, the new(..) message is a class message and it possible definition as pointcut is needed.

## 3.2 Distributed Pattern implementation

This section shows the implementation of some distributed patterns, their descriptions, the problems found, some possible solutions and the leasons learned from this implementation.

### 3.2.1 Model Viewer Controller

**Pattern description**

Application presents content to users in numerous interfaces containing various data. The Model Viewer Controller(MVC)[5] is composed by the model, controller and viewer roles.

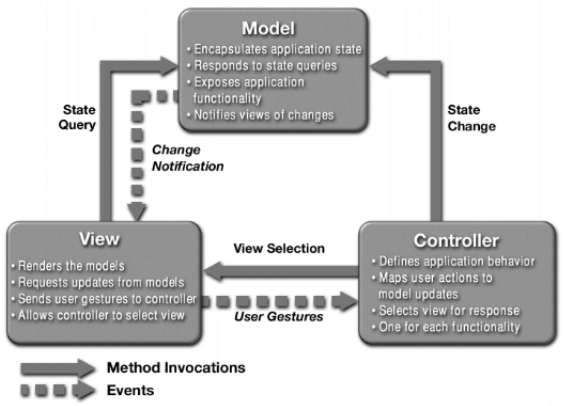


Figure 3.4: Model Viewer Controller Pattern [5]

The model represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.

The view renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes. This can be achieved by using a push model, where the view registers itself with the model for change notifications, or a pull model, where the view is responsible for calling the model when it needs to retrieve the most current data.

The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

### Solution using DJAsCo

```
1  class MVC {

3      hook MVCModel implements Model {
           MVCModel(method(..args)) {
5              execution(method);
           }
7          private Vector <Viewer> viewers = new Vector<Viewer>();

9          public void addViewer(Viewer view) {
               viewers.addElement(view);
11             view.setModel(this);
           }
```

```
13          public void removeViewer(Viewer view) {
                viewers.removeElement(view);
15              view.setModel(null);
            }
17          public Vector getViewers() {
                return viewers;
19          }
            public Object getData(){
21              return null;
            }
23          public Object performeTask(Task task){
                return null;
25          }
        }
27
        hook MVCViewer implements Viewer {
29
            MVCViewer(method(..args)) {
31              execution(method);
            }
33
            private Model model = null;
35
            public void setModel(Model mod)
37          { model = mod;
            }
39          public Model getModel(){
                return model;
41          }
            public void update(){
43
            }
45          public void eventPerformed(Event event){

47          }
        }
49  }

51  ...

53  perobject mvc.aspect.ToDoMVC.TODOMVCModel mixmodel = new mvc.aspect.ToDoMVC.TODOMVCModel(* *.ToDoList.*(*));
    perobject mvc.aspect.ToDoMVC.TODOMVCViewer mixViewer = new mvc.aspect.ToDoMVC.TODOMVCViewer(* *.ToDoDelegator.*(*));
55  ..
```

The previous code defines a reusable aspect that mix base code with the partial implemented and reusable Model and Viewer roles of the MVC, a second specific aspect extends this partially reusable aspect and implements the application specific methods, the last two lines are the wildcards needed in order to introduce the mixing with the base code classes.

### Problems encountered

The cross-cutting of model and viewer concerns could possible impact every graphical interface that interacts human clients and business objects that represent important data on the system. Without means to group these different models and viewers, the aspect implementation of MVC patterns becomes heterogeneous problem that needs one on one implementation and specific application aspect definition, in others words for every model/viewer association there is an aspect that implements the model viewer specific needs.

On the other hand the controller roll is not reusable at all, the events(interactions) and tasks(takss) to perform are different for every viewer/model type of association. Another possible problem arises from the composition of viewers and models, there would be no way to express to with hosts the association belongs to.

### Possible solutions

The declarative group statement of AWED's pointcuts(host(..)) and advicesexecution(on(..)) can be extended in order to be more expressive, in example if two different groups of viewers are applied to the same model, the pointcut and advice declarations can be defined as the interception between the specific viewer and model groups, the following code illustrates this example.

```
1 //pointcut 1 execution(* *method(..))&& host("ModelA) && executionhost("ModelA $ \cap$ JSPViewer);

3 //pointcut 2 execution(* *method(..))&& host("ModelA) && executionhost("ModelA $ \cap$ JavaClient);
```

On the composition problem the order in which the pattern task are implemented can be defined in a graph, each node is a task to be made and the edges are the events, the transition path of tasks to perform and events performed can be solved. Research done on dynamic crosscuts are commonly understood and defined in terms of an event-based models [48, 24] but they leave open the composition of large systems because the whole system has to be taken in mind when defining this kind of event base rules, so the applicability of this solutions is limited to small systems or at least small parts of a large system.

## 3.2.2 Data Access Object

### Pattern description

In computer software, a Data Access Object (DAO) is a software component that provides a common interface between the application and one or more data storage devices, such as a database or file. The term is most frequently applied to the Object design pattern. The purpose of DAOs is to uncouple the database logic so that any changes in database related activities (like change in database) will become simpler. The logic to access the database is abstracted and changes in data access logic will be easier to implement. Figure 3.5 shows the relationships between the DAO objects.

### Solution using DJAsCo

```
1  class DBDAO {
       String login;
3      String password;
       String dbpath;
5      String dridb;
       java.sql.Connection dbcon;
7
       hook initiate {
9          initiate(method(..args)) {
               execution(method)&& joinpointhost("neigh:DAO") && executionhost(localhost);
11         }
           before() {
13             Properties defaultProps = new Properties();
               InputStream in;
15             try {
                   in = getClass().getClassLoader().getResourceAsStream(
17                         "dao.properties");
                   defaultProps.load(in);
19                 ...
       }
21 }
```

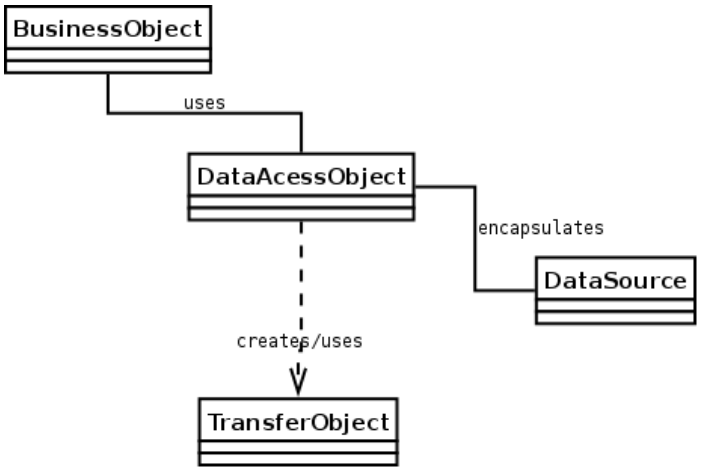


Figure 3.5: Data Access Object pattern

```
1  class ToDoDAO extends DBDAO{

3      public Collection getToDos() {
           Vector<ToDo> todos = new Vector<ToDo>();
5          try {
                   dbcon = DriverManager.getConnection(dbpath, login, password);
7              Statement state = dbcon.createStatement();
               ResultSet rs = state
9                      .executeQuery("select priority,description,date from todos");
               ToDo temp;
11             while (rs.next()) {
                   temp = new ToDo(rs.getString(2), rs.getInt(1), rs.getDate(3)
13                         .toString());
                   todos.add(temp);
15             }
   ...
17         return todos;
       }
19 ...

21     hook getToDos {

23         getToDos(method(..args)) {
               execution(method)&& joinpointhost("neigh:DAO") && executionhost(localhost);
25         }

27         around() {
               return global.getToDos();
29         }
       }
31 }
```

The first piece of code gets the database properties from a property file, then creates the connection, the application specific DAOs aspect extend from this DBDAO aspect the connection, then it implements the specific application needs as in example the second piece of code, an application that manages "to do" tasks that uses this aspect to gather the tasks of the database.

#### Problems encountered

First problem encountered was the lack of expression to determine if a node was suppose to use a database or any other type of source storage, so for every type of source storage a new DAO specific implementation has to be done. The simples way to solve this limitation was to asume only data base source storage and in order to have an aspect more reusable I separated the properties to a file and use pure Statement Query Language(SQL) statements without any Database Management System(DBMS) dependant queries, so the properties file and database driver can be change and reuse the aspect with other databases.

The static definition of groups of hosts at the aspect do not resemble the pattern independence of source storage or business objects, as in example if a new source storage is created and it is used by a single business object, there is no way to express this modification at run time, the only solution would be to re-implement the aspect and include this modifications on it.

#### Possible solutions

The association of a machine process(Node) with some properties like type of storage, localization of the source storage or other information needed, the definition of host as an entity is used in the ReflexD[44] approach to get important data out of the system where a virtual machine is running, it is used to enable or disable hosts pointcuts at runtime.

### 3.2.3 Lessons Learned

Current declaration of AWED's group declaration do not have enough expressiveness to abstract group interactions, as found in the model viewer aspect pattern implementation, this limitation do not enable to define well reusable aspects pattern implementation, that is extended by another aspect in order to add the application specific behaviour. like in the Data Access Object aspect implementation,

The topology of a distributed system can be used in order to know where to apply patterns, it can also be used in order to compose event between hosts and actions as seen at the implementation of the model viewer pattern, this example is just one case of the topology use, this extension would work fine in other cases as remote binding, database replication topology, grid definitions and in any distributed application where the topology management takes an important roll.

## 3.3 Conclusions

In order to implement concurrent patterns using AOP, it is necessary to have synchronization mechanisms, like concurrent libraries or base language support. In absense of concurrent mechanisms, the implementation complexity of these patterns would have been

increased, because of the need to manage concurrency, turning aside the attention from the main idea: implement and support the concurrent patterns.

It is good to have full integration with the language in order not to introduce overhead in a pattern that is meant to increase performance. The solution I defined in order to overcome this limitation may have introduce overhear by doing introspective executions, future performance tests have yet to be done in order to state whether this is true or not. On the other hand object instantiation is important, pointcut languages need to support it, the new(..) message is a class message and it possible definition as pointcut is needed.

AWED's system implementation DJAsCo has helpful features, like separation of the aspect from connectors, stateful aspects, reifinable methods, mixin classes and definition of host groups at pointcut and advice level, just to mention some of them. But when it comes to define or describe irregular topologies or application communication structures, the group declaration is not enough. The distributed AOP support for distribution needs, to be general enough to support the dynamic distributed context, but explicit enough to be useful.

Current AWED syntax defines the body of an advice as a Java statement, because of this the **proceed()** aspect statement is not integrated inside all java statements(in example an inner class definitions) and this is why the implementation of the concurrent patterns at DJAsCo may introduce overhead because of the introspective calls. At the DJAsCo level it is necessary to take into account the distributed implementation, because some thread are created in order to replicate behaviour around hosts and the concurrent pattern or mechanism introduced could not be valid anymore.

# Chapter 4

# AWED Prime language

This chapter presents the extension of AWED language, to improve support for concurrent and distributed patterns implementation, this is achieved by introducing topology concepts and improving the expressiveness of group predicates. Section 4.1 presents two examples to motivate this extension, at section 4.2 the current AWED is briefly described, including its current implementation system DJAsCo, some important features and section 4.3 presents the extension of the language, called AWED Prime.

## 4.1 Motivation for AWED language extensions

In this section I present two possible uses of the group and topology extensions to the AWED language, this two examples and the pattern implementations showed on chapter 3 motivated this extension, those are briefly described in order to show the need and the utility of the extensions.

### 4.1.1 Master Slave pattern

This subsection describes the master slave pattern, the possible use of AWED Prime on it and how it is used to improve the pattern implementation.

The Master Slave pattern [14] declares a computational resource to be the principal entity that processes tasks to one or many clients, called the Master, there are other entities that caches the functionality and data of the Master, called the Slave(s), if the Master system resource is not available, one of the slave resources can replace it, in the ideal case this replace must be as short as possible.

The data base replication is one application of this pattern, some databases(Slaves) that replicate the data stored in the principal one(Master). In case of failure, a topology is defined to replace the database access at the clients and server side of the application. In order to implement this data base replication two alternatives are common, the first is to

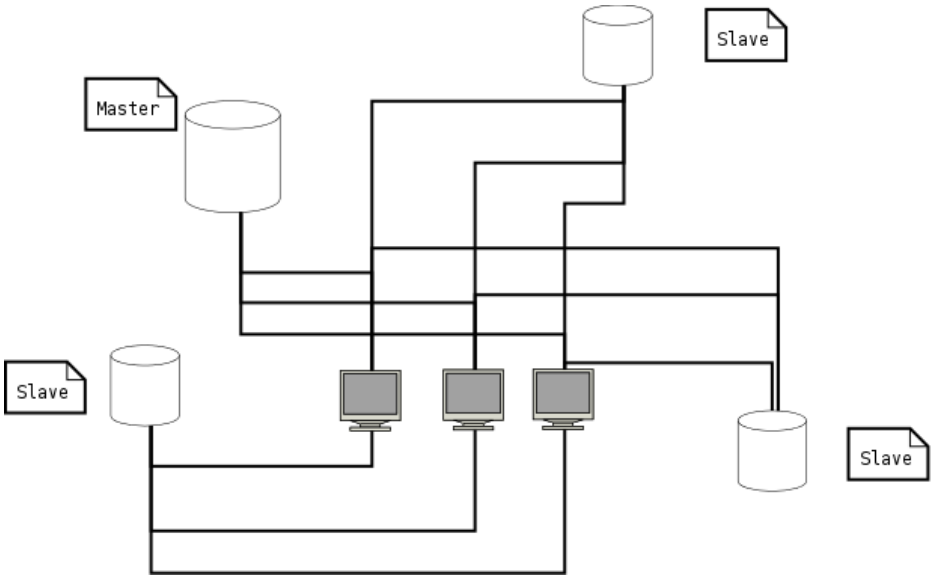


Figure 4.1: Master Slave pattern application.

program in every client of the databases to be used, and the order in which they are used, in other words the auto failover recovery system, as shown on the Figure 4.1, the second is to use a layer that has the previous information and the clients connect to it.

The Figure 4.1 shows one way to achieve the Master Slave pattern in data replication, every client has the implementation of the failover recovery and an external agent(not showed) conducts the replication, this is a clear cross-cutting of the data replication concern and the auto failover concern. Heterogeneous implementation of this failover and replication spread code around host, classes and methods, the functionality is not centralized and hardly maintainable.

Topologies between data replication machines are used, defined and needed, but current AWED does not support them, neither it supports detailed group management, this is why the AWED Prime language is presented to overcome these previous needs by including topology definition and management at language level.

### 4.1.2 Publish Subscribe pattern

This subsection describes the publish subscribe pattern, a pattern used in order to redirect messages across the network. It also shows the possible uses of AWED Prime on it.

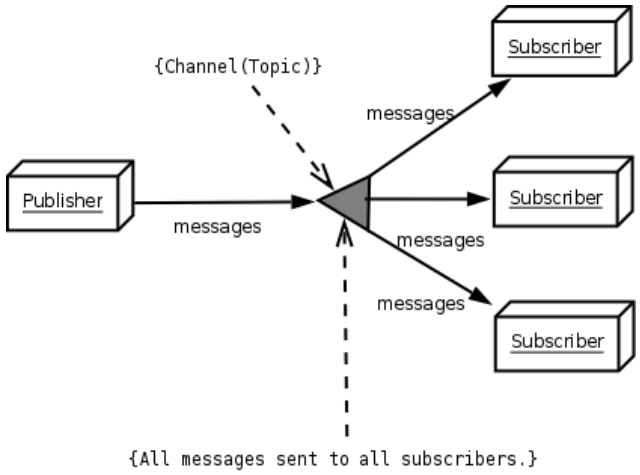


Figure 4.2: Publish Subscribe Pattern.

The messages are created by publishers(producers) and received by subscribers(consumers), this mechanism is called the Publish subscribe pattern. The main idea here is that an agent, the publisher sends messages to a channel, that later all messages are received by subscribers(if any), subscribers express interest in one or more channels, and only receive messages of that channel, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow greater scalability and a more dynamic network topology. The Figure 4.2 shows the representation of this pattern.

Message brokers like ActiveMQ implement this pattern, he current version of ActiveMQ supports many other features also, as compatibility with many languages and protocols among others, but the implementation of those features are tangled in many classes and packages, to give some numbers the total lines of code are 64218 in 61 packages, this crosscutting of features increases complexity at maintenance, document and support level of the system.

AWED Prime is used in chapter 5 to encapsulate message groups feature and use the topology management and group deployment in order to express the pattern and decrease the complexity of the implementation.

## 4.2 Overview of AWED

This section presents features important for the extension of the current AWED and its implementation system DJAsCo.

The current AWED was created at the École des Mines de Nantes and the Vrije Universiteit Brussels, the AWED language supports remote pointcuts constructors, distributed advices ,distributed aspects, state sharing and distributed deployment. The subsection 4.2.1 describes some important parts of the AWED language and subsection 4.2.2 describes other features of its system/implementation DJAsCo .

### 4.2.1 AWED language

The full syntax can be found at Appendix A, but some AWED definitions of a/synchronous executions, remote pointcuts and advices and statefull aspects are important to the extension so they are explained in this subsection.

#### Synchronous and asynchronous executions

Advice executions can be declared as a/synchronous, the **asyncex** reserved word is added in front of the advice declaration meaning that the execution at the joinpoint host continues and the result of the advice execution is done at the advice host, if the **syncex** is used at the advice declaration the execution at the joinpoint host waits until the advice execution is done then the result is send to the joinpoint host and continues there.

This definition and the sharing of arguments and data among hosts is important to pattern definition because some patterns like the future object pattern [30] need an asynchronous task execution and some other like the MVC [5] need to be executed in order and synchronously.

#### Remote pointcuts and advices

Pointcut and advices can be defined to occur at the **localhost**, **jphost**(joinpoint host) or in a **group**(string identifier of the group) of hosts, this definition is for deployment of aspects and sharing. The remote poincuts use the **host(..)** predicate, it means that the pointcut is going to be matched at specific host, group or locally. Remote advices use the **on(..)** predicate, that means that the advice execution take place at the defined host, locally or a group of hosts. The following code exemplifies the remote poincuts and advices.

```
2 aspect Log{
  pointcut LogAllinLocalHost:
4     execution(* *.*.*(..) && host(localhost);
      }
6     before LogAllinLocalHost() && on(Log){
          util.Logger.log((thisjoingpoint.getMethodSignature() );
8     }
}
```

The current AWED definition allows to add or remove the current host to a group or from a group, this is done with the addGroup(..) and removeGroup(..) predicates that can be defined in the body of the advice.

```
1
aspect Log{
3 pointcut ThreeHostExecution:
    seq(execution(* *.*.methodA(..)) && host("172.16.2.3:2121"),
5       execution(* *.*.methodB(..))&& host("172.16.2.4:2121") ,
        syncex execution(* *.*.methodC(..))&& host("172.16.2.5:2121") )
7   }
    after ThreeHostExecution() && on(localhost){
9       util.Logger.getInstance().logImportant("***2.2*** The methods a, b and c of the componentX where called" );
    }
11 }
```

Figure 4.3: AWED aspect example.

In AWED Prime this distributed pointcuts and advices predicates are extended in order to support topology definitions and group predicates as *union* or *interception*.

#### Statefull aspects

Statefull aspects in AWED are defined as the ordered execution of defined pointcuts, aspects that are defined to be statefull are only applied when the execution of pointcuts is done as declared, the following code illustrates this functionality, the aspect pointcut Statefull is only matched after the execution of methods a, b and c are done in that order and finally the advice is executed at the advice host.

The Figure 4.3 presents an example of how an AWED aspect is defined, this example uses this sequence definition, pointcut sequence notion is extended and used at the transformation done from AWED Prime to current AWED, the sequence hosts identifiers are taken from the topology definition.

### 4.2.2 AWED language implementation, DJAsCo

AWED system implementation is called DJAsCo, it is a distributed version of JAsCo [40],some of the important features of DJAsCo system are a/synchronous executions, two performance improvements Jutta and Hotswap and the distributed Cflow are described in this subsection.

#### Synchronous and asynchronous executions

DJAsCo allow advice execution to be done in a synchronous and asynchronous way, then the concurrent future pattern is implemented implicitly at language level. This patterns defines data objects that are blocked if a client tries to use their values before they are fully completed [25]. The UML showing the 3 most important classes that implement this feature in DJAsCo.

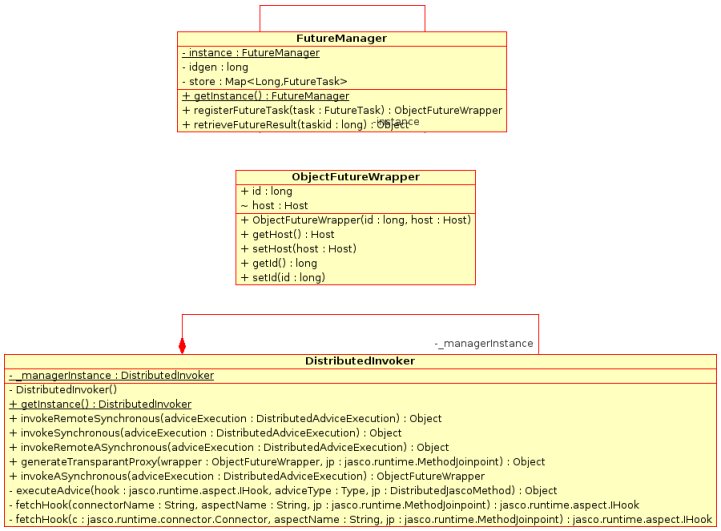


Figure shows the three JAsCo classes.

The current implementation of asynchronous execution is implemented at the the invokeRemoteASynchronous method of the jasco.runtime.distribution.DistributedInvoker class, in order to call this method the hook of the aspect must be annotated with @DistributedAdvice (executionType = DistributedAdvice.Type.ASYNCEX). then the execution is done by creating a wrapper object that later on is transformed in a dynamic Proxy class, the execution of this wrapped Object is managed by the Future Manager class at the jasco.runtime.distribution package. The method invokeRemoteSynchronous to invoke synchronous executions is located at the jasco.runtime.distribution.DistributedInvoker class, this last method is executed by default.

### Hotswap

The current DJAsCo implementation allows to introduce dynamically Aspects by adding connectors to the registry, then a dynamic weaving of the aspect is done and introduced to the affected classes dynamically, this mechanism is called JAsCo HotSwap [46] .

After Java 1.4 the HotSwap is introduced since Java 1.4 and allows to dynamically replace the byte code of a loaded class. Whenever an aspect is applied to a Class the byte code of the class is changed to include traps to the affected methods, this mechanism can be done doing a prepossessing if possible to classes that are definitely affected by aspects, but in case that is not the case the mechanism is done at run time. The implementation methodology of DJAsCo is to deploy every aspects in each of the nodes, so it can be locally introduced and registered in the connector registry on each JVM, afterwards Jgroups is used to communicate aspects and objects across the network.

This mechanism has to be used in order to dynamically deploy AWED aspects and AWED Prime aspects. Two drawback has HotSwap when dealing with distributed AOP,

first it can only change the byte code of a loaded class, so in order to do a remote hotswap the classes that have to be matched with pontcuts, these classes have to be load in every host, second, in order to know that a new class was loaded the Java Virtual Machine(JVM) is running in debuggin mode and the Java Debugging Interface (JDI) is used in order to be notified of these new loaded classes. Because of this use of the JVM is done in JDI there is performance decrease of about 40%.

#### Jutta

Just-in-time combined aspect compilation(Jutta)[46] is proposed in order to overcome the overhead that dynamically deployed mechanism causes, this system allows to generate and cache code fragment for a given joinpoint. This code fragment directly executes the appropriate advices on the applicable hooks in the sequence defined in the connector, the order of sequences of all applicable hooks for different advice types in order to implement precedence strategies is done just once, JAsCo performance is better than AspectJ, according to performance tests[16], this is achieved because of the combination of the hotswap and Jutta systems.

On the other hand, due to dynamic variables like the cflow condition, deployment cannot be applicable, because dynamic conditions have to be re-evaluated for every execution of a given joinpoint. This dynamic condition apply to the topology definitions, the topology can change at runtime and this caching would not be achievable.

#### Cflow

Each distributed Cflow is managed in a Stack data structure, this stacks are encapsulated in a list inside an RemoteInfo Object, this serializable Object is distributed to remote hosts using Remote Method Invocation(RMI). A single thread called JascoThreadCflow manages in each host the information of the executions. This cflow use in AWED Prime is necessary, this RemoteInfo definition has to be extended in order to deal with the new topology definition and group predicates.

## 4.3 AWED Prime

In the distributed heterogeneous context, in which distributed AOP is, the need for a better definition of poincuts, deployment and reuse of topology definitions arises as shown in the Motivation of AWED language extension section. The extension to the current AWED [8] language approach is defined in order to support those needs. the main objectives of this extension are:

- To have a better deployment information and declaration. The **single** and **all** current deployment mechanism define an aspect to be deployed only locally or in every host, group deployment is proposed to overcome this limitation.

- To have a more declarative group declaration. Groups predicates are defined in order to have better expressiveness.
- Reuse topologies definitions, distributed application have topologies predefined, AWED should be able to manage and use those definitions, introducing the concept of topology, node and edge at language level this objective is achieved.

This section presents the AWED Prime language, subsection 4.1 shows the syntax, section 4.2 the semantics and section 4.3 the overview of AWED Prime implementation.

### 4.3.1 Syntax

The following syntax definition aims to give a more flexible group management, definition and use. This part of the extension suits the irregular topologies and heterogeneous communication requirements, as an example the groups can be identifiers of a set of computers that have some functional similarity, like a database service, or hardware capacity as broadband, or even a relation can be established between roles of the nodes and the defined groups, but most important, the hosts that interact on the application topology can change over time.

*// Aspects*

| | | |
|---|---|---|
| Asp | ::= | [Depl] [Inst] [Shar] `aspect` Id '{' {Decl} '}' |
| Depl | ::= | `single` \| `all` \| `Group` |
| Inst | ::= | `perthread` \| `perobject` \| `perclass` \| `perbinding` |
| Shar | ::= | `local` \| `global` \| `inst` \| `Group` |
| Decl | ::= | [Shar] JVarD \| PcDecl \| Ad \| graph |
| PcDecl | ::= | `pointcut` Id({Par}) : Pc |

*// Pointcuts*

| | | |
|---|---|---|
| Pc | ::= | `call`(MSig) \| `execution`(MSig) |
| | \| | `get`(FSig) \| `set`(FSig) |
| | \| | `cflow`(Pc) \| Seq |
| | \| | `host`(Group) \| `on`(Group[, Select]) |
| | \| | `args`({Arg}) \| `args`({Arg,Group }) |
| | \| | `host`(TopoHostPred) \| `on`(TopoHostPred) |
| | \| | `eq`(JExp, JExp) \| `if`(JExp) |
| | \| | `within`(Type) \| `target`({Type}) |
| | \| | Pc ‖ Pc \| Pc && Pc \| !Pc |
| Seq | ::= | [Id:] `seq`({Step}) \| `step`(Id,Id) |
| Step | ::= | [Id:] Pc [→ Target ] |
| Target | ::= | Id \| Id Target |
| Host | ::= | `localhost` \| `jphost` \| "ip:port" |
| GroupId | ::= | String |
| Select | ::= | Jclass |

| | | |
|---|---|---|
| Group | ::= | `group`(Group BinGroupOp Group) \| `group`(UnGroupOp Group) |
| | \| | GroupId \| Host |
| BinGroupOp | ::= | `-` \| ∪ \| ∩ |
| UnGroupOp | ::= | `complement` \| `groupAnd` \| `anyHost` |

*// Advice*

| | | |
|---|---|---|
| Ad | ::= | [syncex] \| Pos({Par}) : PcAppl '{' {Body} '}' |
| Pos | ::= | `before` \| `after` \| `around` |
| PcAppl | ::= | Id({Par}) |
| Body | ::= | JStmt \| `proceed`({Arg}) \| TopoPred |
| | \| | `addGroup`(Group) \| `removeGroup`(Group) |

*// Topology syntax and predicates*

| | | |
|---|---|---|
| Topology | ::= | `Id` \| `graph` \| (TId, TopoPred) |
| TopoPred | ::= | `defaultGraph`() |
| | \| | `addNode`(node_stmt) \| `removeNode`(node_stmt) |
| | \| | `addEdge`(edge_stmt) \| `removeEdge`(edge_stmt) |
| TopoHostPred | ::= | `predecessors`(Host) \| `successors`(Host) |
| | | |
| graph | ::= | 'graph' `Id` \| '{' {stmt_list} '}' |
| stmt_list | ::= | {attr_list} \| {node_stmt} \| {edge_stmt} |
| attr_list | ::= | '[' {a_list} ']' |
| a_list | ::= | `Id` '=' `Value` ';' |
| node_stmt | ::= | 'node' `NodeId` attr_list ';' |
| edge_stmt | ::= | 'edge' `EdgeId` attr_list `NodeId` → `NodeId` ';' |

*// Auxiliary functions*

| | | |
|---|---|---|
| *Type* | ::= | *// type expressions* |
| *Arg*, *Par* | ::= | *// argument, parameter expressions (AspectJ-style)* |
| *Id* | ::= | *// identifier* |
| *Ip*, *Port* | ::= | *// integer expressions* |
| *JClass* | ::= | *// Java class name* |
| *JExp* | ::= | *// Java expressions* |
| *JStmt* | ::= | *// Java statement* |
| *JVarD* | ::= | *// Java variable declaration* |

### 4.3.2 Semantics

This subsection presents the definition of the topology predicates and the necessary group transformation to get an AWED code from the just defined AWED Prime. First, the informal semantics of the topology and group extension, and second, a transformation into plain AWED from the AWED Prime group expressions.

Two important things have to be taken into account when defining the semantics of AWED Prime: *First* the deployment group of an aspect defines its domain, it should be interpreted as the Universe group of the aspect being deployed, anything outside this group is not in the scope of the aspect. *Second,* irregular topology supports requires an entity, to store, update and retrieve the current Topology at a given time, this new entity, called TopologyManager, supports the previous defined requirements, and can be accessed in order to be use by the topology or group predicates, the implementation of this TopologyManager is proposed in the subsection Overview of AWED Prime implementation.

### Group specifications, informally

The binary predicate " $-$ " receives two groups as parameters, and the result are the host that belong to the group one but not to the group two. The predicate " $\cup$ " refers to the hosts that belong to group one or to the group two and the predicate " $\cap$ " refers to the host that belong to the group one and to the group two.

The unary predicate **complement** refers to the host that are in the deployment group of the aspect but not on the groupid passed as argument, the **groupAnd**, matches all the host that belong to a group in a sequence of pointcuts and the **anyHost** resolves to one and only one host that belongs to the given group.

### Topology specifications, informally

On the topology syntax, the graph definition is straightforward, what must be explained are its predicates named at the syntax as TopoPred, this predicates can be used at the advice, the **defaultGraph()** returns the id of a just created empty graph, the **removeNode(NodeId)** removes the given node, the **addNode(NodeId)** adds the given node, the **removeEdge(EdgeId)** removes the given edge, the **removeEdge(EdgeId)** removes the given edge.

TopoHostPred defines the predicates that can be used at the poincut, the **predecessors(Host)**, resolves a pointcut sequence that contains all the predecessors of the given Host and the **successors(Host)** a pointcut sequence that contains all the successors of the given host.

### Group specifications, formally

The transformation from AWED Prime to AWED as follows:

$$
\begin{aligned}
\mathcal{T}_G|[Group, type|] \quad = \quad & \underline{\texttt{if}}\ isGroupid(Group, type) \\
& \quad generateGroupPredicate(Group, type) \\
& \underline{\texttt{if}}\ isHost(Group) \\
& \quad generateHostPredicate(Group, type) \\
& \underline{\texttt{if}}\ isBinaryOP(Group) \\
& \quad \mathcal{T}_{BOp}|[g1(Group), g2(Group), gOp(Group), type|] \\
& \underline{\texttt{if}}\ isUnaryOp(Group) \\
& \quad \mathcal{T}_{UOp}|[g1(Group), gOp(Group), type|]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_{BOp}|[g1, g2, op, type|] \quad = \quad & \underline{\texttt{if}}\ \mathcal{T}_{BOp}|[g1 \cap g2, type|] \\
& \quad \mathcal{T}_G|[g1, type|]\ \&\&\ \mathcal{T}_G|[g2, type|] \\
& \underline{\texttt{if}}\ \mathcal{T}_{BOp}|[g1 \cup g2, type|] \\
& \quad \mathcal{T}_G|[g1, type|]\ ||\ \mathcal{T}_G|[g2, type|] \\
& \underline{\texttt{if}}\ \mathcal{T}_{BOp}|[g1 - g2, type|] \\
& \quad \mathcal{T}_G|[g1, type|]\ \&\&\ !\ \mathcal{T}_G|[g2, type|]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_{UOp}|[Group1, op, type|] \quad = \quad & \underline{\texttt{if}}\ isComplement(op) \\
& \quad !\ \ \mathcal{T}_G|[g1, type|]\ \&\&\ \mathcal{T}_G|[deploymentGroup(), type|]
\end{aligned}
$$

- $generateGroupPredicate(g, type) = if(type == on)\ then\ on(g.id)\ else\ host(g.id)$

- $generateHostPredicate(h, type) = if(type == on)\ then\ on(h.id)\ else\ host(h.id)$

*// Auxiliary functions*

| | | |
|---|---|---|
| *type* | ::= | *// can be on(..) or host(..).* |
| *isGroupid* | ::= | *// determines whether group is a group id or not.* |
| *isHost* | ::= | *// determines whether group is a host or not.* |
| *isBinaryOP* | ::= | *// determines whether group definition has a binary operation.* |
| *isUnaryOp* | ::= | *// determines whether group definition has an unary operation.* |
| *isInterception* | ::= | *// determines whether group binary operator is interception.* |
| *isUnion* | ::= | *// determines whether group binary operator is union.* |
| *isDifference* | ::= | *// determines whether group binary operator is a difference.* |
| *isComplement* | ::= | *// determines whether group unary operator is a complement.* |
| *deploymentGroup* | ::= | *// resolves the deployment group of the aspect.* |

The definition of the predecessors and successors functions:

- $predecessors(n) = \{\ n' \in Node \mid n' \in \{TogologyManager.getHosts()\} \wedge$
  $n' \rightarrow n \in \{TogologyManager.getEdges()\ \}\ \}$
- $successors(n) = \{\ n' \in Node \mid n' \in \{TogologyManager.getHosts()\} \wedge$
  $n \rightarrow n' \in \{TogologyManager.getEdges()\ \}\ \}$

Many more topology predicates can be useful depending on the application context, in example inmediateSuccesors, inmediatePredecessors, allPredecessors and allPredecessor-Successors.

**Constraints,** In the transformation these applied:

- The definition of node contains at least the "ip:port" and "group" as attributes of the node.
- The definition of edge describes a directional relationship from the first to the second node and an attribute list.
- Topology predicates con only be used when a Topology deployment is used or when a Topology is later created.
- The body statements addNode(node) and removeNode(node) can only be used when a topology is defined.

### 4.3.3 Overview of AWED Prime implementation

This subsection presents an overview of the implementation of the AWED Prime over current AWED implementation system DJAsCo. The first the group extension and second the topology extension.

#### Group extension

The group extension aims to improve group declaration but not to emulate set theory, in example the pointcut declaration $on(Administrators \cap Personnel \cup Clients)$ is valid, but there is no precedence delcaration using parenthesis(), the declaration $on((Administrators \cap Personnel) \cup Clients)$ is not valid. Group extension has been defined in order to allow group predicates, these predicates can be unary or binary operators. Three operators are defined to be binary group operators: *union* $\cup$, *intersection* $\cap$ and *difference* -, the unary operators are *complement*, *groupAnd* and *anyHost*. My proposal to implement this group extension is to parser the group definition using the transformation defined in the syntax subsection, Figure 4.4 shows the class diagram of this proposed solution.

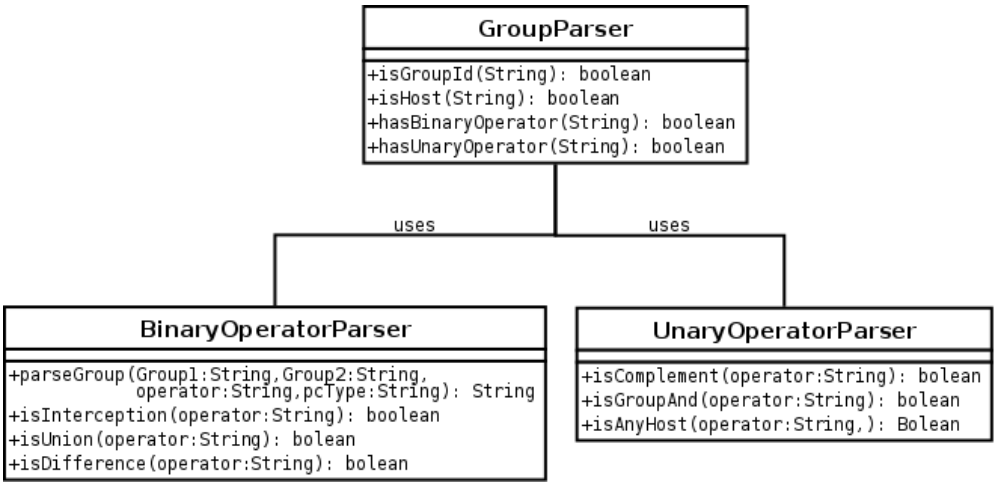


Figure 4.4: Group extension class diagram.

The propose is to pass the string from the on(..) or host(..) declarations and return the complete string transformation, as example is the input pointcut group declaration is $on(Administrators \cap Personnel \cup Clients)$ the output will be: on(Administrators) && on(Personnel) || on(Clients).

### Topology extension

This extension is defined in order to give management support for distributed application topologies at language level. the figure Figure 4.5 shows the class diagram of the proposal. The TopologyManager class is a singleton facade class, it can add and remove groups and/or hosts to groups, create edges between nodes, adding attributes to nodes and edges, it also has the responsibility of of managing(storing and returning) the deployment group of every aspect.

Current AWED implementation system DJAsCo uses a class call the RemoteInfo(found the jasco.util.distribution) in order to manage the cflow executions, this class is the best place to add the TopologyManager because it is already distributed to the hosts. The impact of this modification can be located in classes: RuntimeContext(jasco.runtime package), DistributedStandarMessage(jasco.runtime.distribution package), JascoInputStream( jasco.util.distribution), JascoOutputStream( jasco.util.distribution) and JascoThreadCflow( jasco.util.distribution).

In case of difficulties introducing this extension to the current DJAsCo system, a second implementation proposed of this extension is presented, messages are transferred

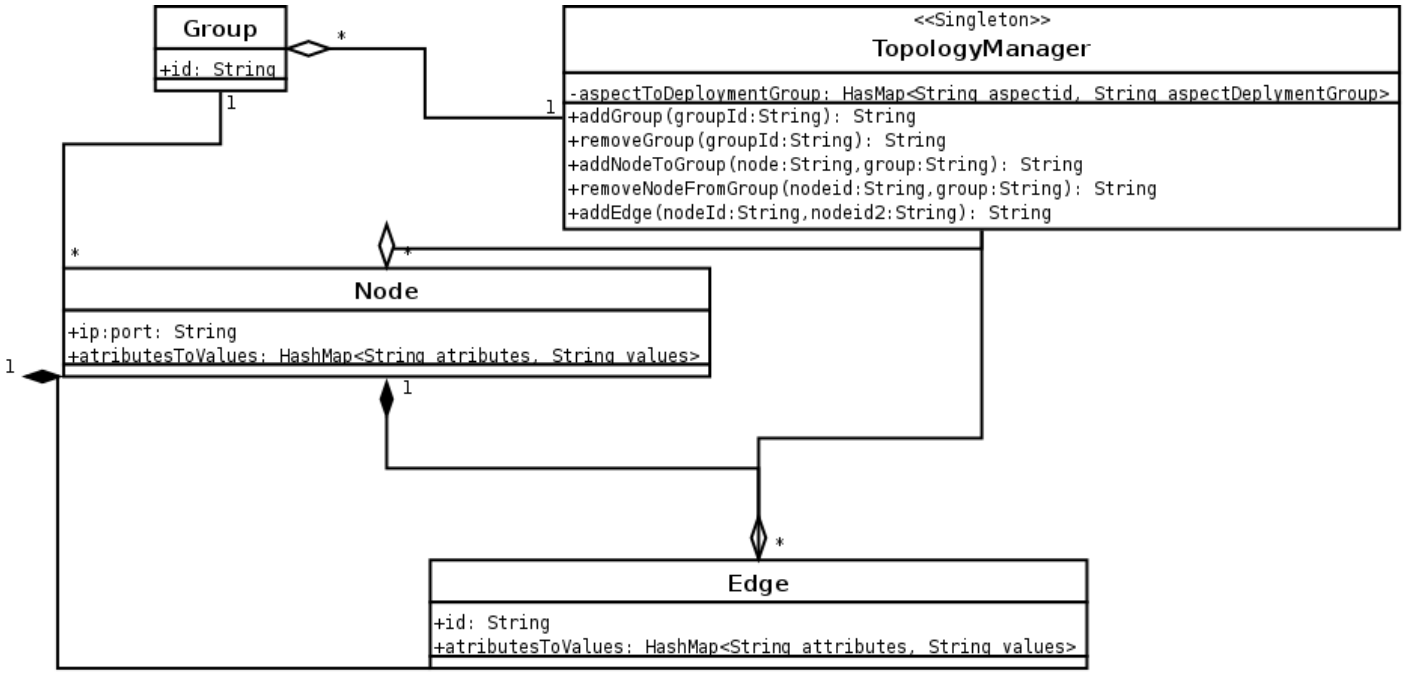


Figure 4.5: Topology extension class diagram

between remote or local JVMs, this message distribution could be done by an external agent(Message broker as ActiveMQ), and the messages are parsered and topology is managed locally in each JVM, the same functionally showed in Figure 4.5 is used and the same responsibilities apply. The message distribution system is a Star system that is in charge of distributing the messages across the registered JVM clients.

The first message that just registered JVM will receive would be the current topology message( using the defined graph of the topology extension syntax), from that moment the local TopologyManager will get the messages of changes of the topology, so it is locally replicated by sending messages. This approach reduces to tangled behaviour with the current RemoteInfo class but increases the complexity of the implementation because a parser of the messages would be needed.

## 4.4 Conclusions

This extension chapter, motivates with two examples, the need for Topology management and more expressive group definitions. This examples showed, the utility for the extension. On the other hand, current important AWED features were presented, also the implication of AWED Prime on them, the AWED system implementation features were also presented and the precautions to have in mind, when implementing the extension on it.

On this chapter, we presented: Syntax, the topology management and better group definition language extensions. Semantics, the definitions for each extension and the transformation to be used for the groups new declaration, with their constraints. Language extension implementation proposals, one for the group extension and two proposals for the topology management, defining its impact on current AWED implementation DJAsCo and possible modules to introduce it.

# Chapter 5

# AWED Prime application examples

This chapter presents two possible uses of the AWED Prime language, section 5.1 presents the first use case Master Slave pattern, this pattern has been applied in many areas, like data bases replication and high availability systems(where continuity is crucial to the system clients). The second use case shown in section 5.2 is the Publish subscribe pattern, this distributed pattern has been used in the implementation of message brokers like ActiveMQ.

## 5.1 Master Slave pattern

The Master Slave pattern defines one computational resource to be the principal entity to process tasks to one or many clients called the Master, there is a second entity that replicates the functionality and data of the Master called the Slave, whenever necessary the slave resource can replace the Master, in example the replacement can be force because of Master's failure. This section presents the AWED Prime definition of the Master Slave pattern applied to Database replication.

The data base replication is one application of this pattern, there are some Database Management Systems(DBMS) that have database replication feature implemented, it replicates a principal database and the other(s) databases copy data from the Master, in case of Master's failure there is a structure(topology) defined to replace the database access at the clients and server side of the application.

Awed Prime manages the groups and the topology defined in order to replace the Master's data and behaviour in case of failure. In this use case databases hosts are associated to groups, the Master to the **"Master"** group and the Slave(s) to the **"Replicate"** and their topology defined and used in the same aspect, realizing modularization of an commonly scaterred distributed pattern. The access from the clients to the Master can be done using AOP[39] also, but it is not part of the pattern.

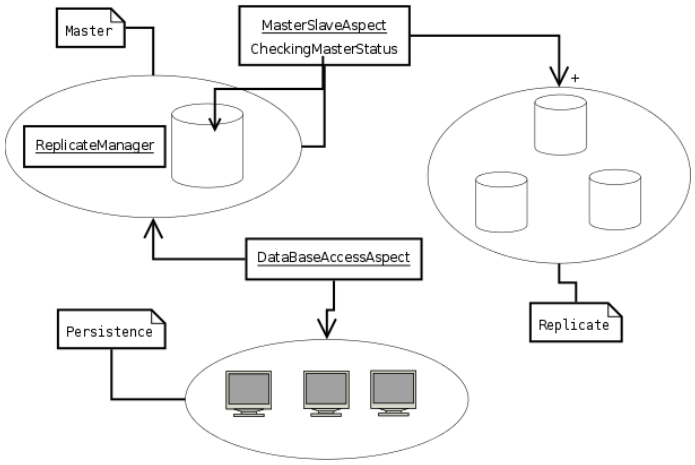


Figure 5.1: Master Slave pattern AOP application

The Figure 5.1 shows two aspects, one is in charge of the persistence concern, it uses the group Persistence and Master in order to maintain persistency and the second of the Topology concern, the MasterSlaveAspect monitors the state of the Master data base and replicates the data to the Replicate group, in case of failure of the Master host, one of the Replicate databases is to be used to replace it, then it will be deleted from the Replicate group and added to the Master group.

Another important task to be done at the Master Slave is to continue using the Master resource if possible, then the Topology of the nodes can be defined in order to give some nodes precedence. The MasterSlaveAspect uses topology to define the auto fail over recovery and ReplicateManager object to manage the replication in this application of the Master Slave Pattern as shown in the following code:

```
1  group("Master" U "Replicate" ) aspect MasterSlaveAspect{

3      pointcut masterUpdated(ReplicateManager repl):
           execution(* MasterCommitMethod(*)) &&
5                  host("Master") && on(group("Master" )) ;

7
       pointcut masterDown(ReplicateManager repl):
9          execution(* MasterDownMethod(*)) &&
                   host("Master") && on(group("Master" U "Replicate" )) ;
11
       pointcut masterUp(ReplicateManager repl):
13         execution(* MasterUpMethod(*)) &&
                   host("Master") && on(group("Master" U "Replicate" )) ;
15
       around() : masterUpdated(ReplicateManager repl) {
17                 repl.updateSlaves();
                   proceed();
19             }

21     around() : masterDown(ReplicateManager repl) {
                   TopologyManager.addToGroup(TopologyManager.sucessor(repl.getCurrentMaster()), "Master");
23                 TopologyManager.removeFromGroup(repl.getCurrentMaster(), "Master");
                   cach.setCurrentMaster(sucessor(repl.getCurrentMaster()));
25                 proceed();
```

```
        }
27
    around() : masterUp(ReplicateManager repl) {
29          TopologyManager.removeFromGroup(TopologyManager.sucessor(repl.getCurrentMaster()), "Master");
            TopologyManager.addToGroup(repl.getMaster(), "Master");
31          proceed();
        }
33  }
```

As showed in the previous code, the implementation of the Master Slave pattern is modularized and simplified in a single aspect, this is done at language level, with the help of the new concept/entity of Topology. The group and topology are used in the aspect definition. On the other hand, this pattern definition is also reusable on databases replication, the pointcut methods can be replaced by application specific wildcards method and it will be applicable and functional .

## 5.2 Apache ActiveMQ

The Apache ActiveMQ is a Message Broker that supports many Cross Language Clients and Protocols and many advanced features while fully supporting JMS 1.1 and J2EE 1.4, It is written in Java together with a full JMS client. However Apache ActiveMQ is designed to communicate over a number of protocols such as Streaming Text Oriented Message Protocol(Stomp) and OpenWire(Binary message protocol) together with supporting a number of different language specific clients like Java, .NET, C/C++, Perl, Python, PHP and Ruby.

The Publish subscribe pattern( said in chapter 4) achieves scalability and improves the dynamism of the network, by doing asynchronous calls from publishers to unknown subscriber(s), that can or not reply each other depending on the declaration of both. The idea was shown in Figure 4.2, a publisher sends a message to a channel and that message is resend to all channel subscribers if any.

In contrast, when messages are too important to get lost, this pattern has to be extended in order to define new needs, needs like: *Processing*: messsages can get lost without having a subscriber to manage the requirements of the message; *Performance*: network traffic is increase by a number of n, being n the number of subscribers; and *Exclusivity*, it could be necessary that a type of message is only received by a determined subscriber or subscribers.

**Solution objectives,** the Apache solution to the previous problems was to create a queue, inside this queue messages are send to subscribers depending among others of the speed of the subscribers, subscriber's exclusivity or message grouping. This queue stores all messages sent to it and releases messages only when a subscriber gets connected, *processing all messages*, a message is send only once, *decreasing network traffic* and if needed a selection mechanism can be define to choose to which subscriber the message should be send, gaining *exclusivity*. Message Groups is going to be explained and its AWED' implementation documented. Figure 5.2 shows the queue mechanism.

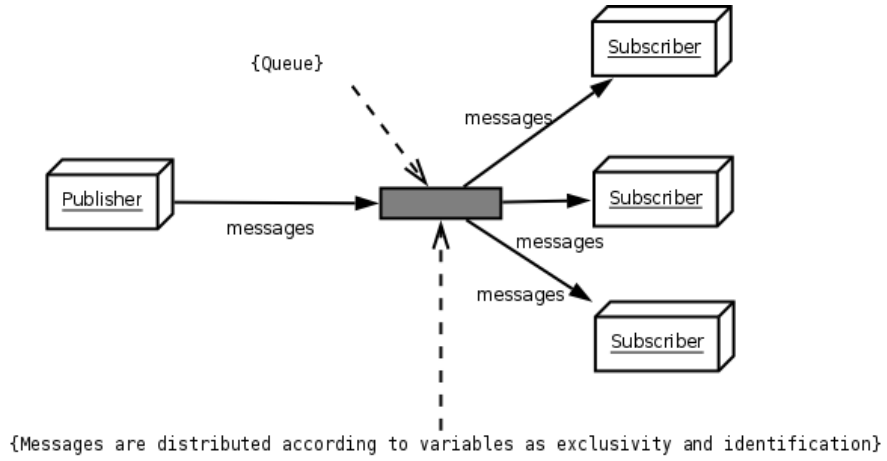


Figure 5.2: ActiceMQ queue: Publish Subscribe pattern application.

**Message groups**

In this extension of the publish subscribe pattern, messages are send to a queue, this messages can be marked to belong to a group, when the message broker finds that a message sent to the queue have a group mark, it creates an association between a group mark and a subscriber for every message declared group, afterwards whenever a message is send to the queue and has a group mark already associated, it sends this message to the associated subscriber. Now that many groups can be defined in a single queue, parallel exclusivity can be declared and used, in case that a associated subscriber goes down, a new association between the group and a subscriber is created. In other words message groups achieve:

- Ordering of the processing of related messages across a single queue, using the association.
- Load balancing of the processing of messages across multiple consumers, message group balance.
- High availability / auto-failover to other consumers if a JVM goes down, group re-association if needed.

### 5.2.1 Motivation for distributed AOP and overview of application of AWED Prime

Message groups features is scaterred around six packages and nine classes at the message broker, and one extra invokation is matched for every new subscriber and publisher. AWED

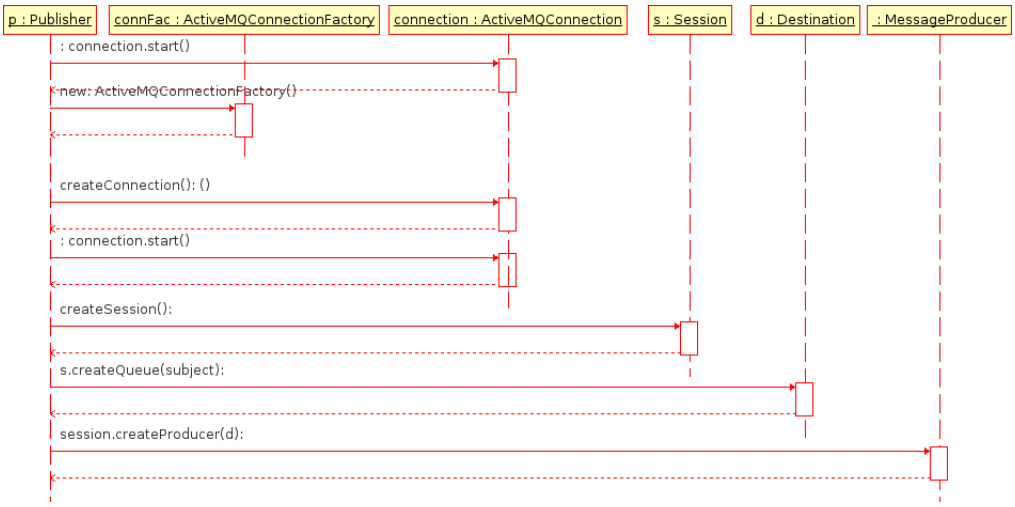


Figure 5.3: Sequence diagram to create a Publisher.

Prime is applied to re-implement message groups feature, its topology predicates are used to map from *group messages* associations to *edges* between publisher and subscriber nodes, the AWED Prime group extension is used to add subscribers to "subscriber" group and publishers to "publishers" group. Messages are managed by the MessageGroupAspect, new nodes are added dynamically to the groups and new edges created every time a message group is defined, in order to maintain the same functionality whenever a subscriber or a publisher goes down, the edges and nodes associated to that subscriber or publisher are removed.

The pointcuts where the group managing have to match is showed in Figure 5.3. The last part of the sequence diagram is the pointcut where the host is added to the publisher group, previously the broker has been created and initiated, a relative similar sequence is done in order to create the subscriber. The Topology of the broker is modeled by the TopologyAspect that uses a TopologyManager( part of the AWED Prime library) in order to add host to groups and remove them form them, the TopologyAspect code is shown:

```
1  all aspect TopologyAspect{
3      pointcut messageBrokerCreation():
           execution(* org.apache.activemq.broker.BrokerService.addConnector(*));
5
       pointcut publisherCreation(javax.jms.Destination destination):
7          execution(* javax.jms.Session.createProducer(*))
           && args(javax.jms.Destination destination);
9
       pointcut subscriberCreation(javax.jms.Destination destination)):
11         execution(* javax.jms.Session.createConsumer(*))
           && args(destination);
13
       pointcut publisherDown(org.apache.activemq.command.ProducerId id):
15         execution(* org.apache.activemq.broker.TransportConnection.processRemoveProducer(*)) && on("Broker")
               && args(id);
17
```

```
    pointcut subscriberDown(org.apache.activemq.command.ConsumerId id):
        execution(* org.apache.activemq.broker.TransportConnection.processRemoveConsumer(*)) && on("Broker")
                && args(id);

    after() : messageBrokerCreation(opub){
        Node nod=TopologyManager.addNode(new node(jpHost));
        TopologyManager.addToGroup(nod, "Broker");
    }

    after(objectType opub) : publisherCreation(opub){
        Node nod=TopologyManager.addNode(new node(jpHost));
        TopologyManager.addToGroup(nod, "Publishers");
    }

    after(objectType opub) : publisherDown(id){

        TopologyManager.removeFromGroup(id, "Publishers");
        TopologyManager.removeNode(id);
    }

    after(objectType opub) : subscriberCreation(opub){
        Node nod=TopologyManager.addNode(new node(jpHost));
        TopologyManager.addToGroup(nod, "Subscribers");
    }

    after(objectType opub) : subscriberDown(id){
        TopologyManager.removeFromGroup(id, "Subscribers");
        TopologyManager.removeNode(id);
    }
}
```

After the Topology has been modeled, the MessageGroupAspect models, the message sending to a Subscriber, there is then a strong precedence strategy needed for these two aspects, the code of the MessageGroupAspect as follows:

```
group("Publishers" U "Subscribers" ) aspect MessageGroupAspect{

    pointcut messageGroupPossibleDeclaration( Message mess):
        execution(* javax.jms.TextMessage.setStringProperty(*)) && host("Publishers");

    pointcut messageSend():
        execution(* javax.jms.MessageProducer.send(*)) && host("Publishers")

    around(objectType opub) : messageGroupPossibleDeclaration(Message mess) {
        if( MessageHandler.isMessageGroup(mess) ){
                Host ahost=anyHost("Subscribers");
                String[] attributes=new String[1];
                attributes[0]="MessageIdGroup";
                String[] attributesValues=new String[1];
                attributesValues[0]=MessageHandler.getGroupId(mess);
                TopologyManager.addEdge(jphost, ahost, attributes, attributesValues);
                TopologyManager.removeFromGroup(ahost, "Subscribers");
                proceed();
            }
    }

    around() : messageSend() {
        if( isMessageGroup(mess) ){
            if(TopologyManager.nodeHasSuccessor(jphost)
                MessageGroupManager.sendMessage(mes, successor(jphost, getMessageIdGroup(mes)));
        }
        else{
                proceed();
        }
    }
}
```

### Strong precedence strategy

On the previous code, the topology is managed dynamically, an edge is created between the publisher and the subscriber in case that a message has a Group declaration as a property. This is an invasive enabled aspect, in order to apply the MessageGroupAspect the topology has to be defined by the TopologyAspect correctness of the implementation,

if the topology is not defined the MessageGroupAspect is not correct. The TopologyAspect has to be applied always before the MessageGroupAspect, and the MessageGroupAspect should not be applied if the TopologyAspect is not applied, in other words there is not only a precedence between the aspects but a dependence.

**Evaluation**

The application becomes more readable and centralized, only two Aspects and one class( TopologyManager) that belong to the AWED Prime are needed to implement the Message Groups functionality of ActiveMQ, compared to the original code that includes code scaterred around six packages and nine classes. The Message Group feature is located in only two aspects that clearly resemble its functionality, in resume it is better defined and better modularized.

## 5.3 Conclusions

AWED Prime has been motivated with two widely used patterns, the Publish Subscribe and the Master Slave pattern, each of them with different requirements for distributed communication, and irregular topology management. AWED Prime extended group definition, is used in the Master Slave pattern application on Database, to monitor changes on the Master Database and inform the Slave Databases, the topology is used to implement the auto-failover recovery, Database hosts moved from the Slave group to the Master group when needed. This pattern application is done in only one aspect and Two Classes, the ReplicateManager class and the TopologyManager(part of the AWED Prime library). It is a reusable distributed architectural pattern definition.

On the other hand, AWED Prime is used to implement the Publish Subscribe pattern application, the Message Groups feature, the topology is first used, to capture the dynamic changes of publishers and subscribers. Later on, it is used to associate message group declarations to subscribers, and finally to redirect messages to the associated subscriber. The AWED Prime implementation uses two aspects that have an important precedence dependency, it was was explained. This implementation is modularized into two aspects and one class, the TopologyManager(part of the AWED Prime library), improving its definition and localization.

AWED Prime has proven to improve distributed architectural patterns definitions and implementations in these two widely used patterns, more research has to be done, in order to fully understand the pattern support implications of topology management at AOP language level.

# Chapter 6

# Conclusions

In order to support concurrent paterns with AOP, It is necessary to have synchronization mechanisms, like concurrent libraries or base language support. In absense of concurrent mechanisms, the implementation complexity of these patterns would have been increased, because of the need to manage concurrency, turning aside the attention from the main idea: implement and support the concurrent patterns.

The Object Oriented implementation of concurrent and distributed patterns is difficult, it is because of the added complexity of heterogeneous communication requirements and irregular topologies. Current pattern implementations have code scaterred and tangled with other concerns, not only in a local way but around many hosts. We propose an extension to AWED in order to improve group expressiveness, communication requirements and provide topology predicates. This report is basically divided in four parts, the research of the state of the art, the implementation of concurrent and distributed patterns, a proposal to solve the problems and needs found at the concurrent and distributed pattern implementation, called AWED Prime. And two examples of the applicability of AWED Prime on widely used patterns.

AWED Prime is defined to have three primary goals: Better deployment information and declaration, the group deployment of an aspect bounds its domain to include only hosts of interest; More expressive group declaration, using *group predicates* on advices and poincuts, group declaration achieved better expressiveness and abstraction of patterns; *Topology support*, distributed patterns are implemented over irregular topologies. AWED Prime language level support for topologies was able, to abstract and manage the irregular topology and communication requirements of the patterns studied. The topology and group support of AWED Prime were used to implement two widely used patterns. First, the Master Slave Pattern, in which AWED Prime express the database replication using **topology predicates** and group declaration, it also achieved partial reusability and modularization of the pattern into a single aspect. The second use case is done on Apache ActiveMQ, AWED Prime separates the extended implementation of the publish subscribe pattern, called Messages Groups, in which the topology extension provided abstraction

and management of the dynamic network, and the association between publishers and subscribers.

AWED Prime has proven to improve distributed architectural patterns definitions and implementations in these two widely used patterns, more research has to be done, in order to fully understand the pattern support implications of topology management at AOP language level. On the other hand, this study leads to several future works: Extend current AWED system implementation DJAsCo in order to provide AWED Prime definitions; Define and implement more concurrent and distributed patterns using the AWED Prime system implementation; Evaluate AWED Prime, over other distributed areas that require topology management and distributed communication requirements, like peer-to-peer applications, grid-base applications and frameworks for distribution.

# Bibliography


[1] *Design patterns explained: a new perspective on object-oriented design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] Alexander Ahern and Nobuko Yoshida. Formalising java rmi with explicit code mobility. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 403–422, New York, NY, USA, 2005. ACM Press.

[3] Jonathan Erik Aldrich. *Using types to enforce architectural structure.* PhD thesis, 2003. Chair-Craig Chambers and Chair-David Notkin.

[4] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1979.

[5] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies.* Sun Microsystems, Inc., Mountain View, CA, USA, 2003.

[6] Eclipse AspectJ. Aspectj. http://www.eclipse.org/aspectj/.

[7] Luis Daniel Benavides Navarro, Mario Südholt, Rémi Douence, and Jean-Marc Menaud. Invasive patterns: aspect-based adaptation of distributed applications. In *4th International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'07) at the 21st European Conference on Object-Oriented Programming ECOOP'07*, July 2007.

[8] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th Int. ACM Conf. on Aspect-Oriented Software Development (AOSD'06).* ACM Press, March 2006.

[9] Nelio Cacho, Claudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Thais Batista, and Carlos Lucena. Composing design patterns: a scalability study of aspect-oriented programming. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 109–121, New York, NY, USA, 2006. ACM Press.

[10] Siobhán Clarke and Robert Walker. Composition patterns: an approach to designing reusable aspects. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 5–14, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Adrian Colyer and Andrew Clement. Large-scale aosd for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65, New York, NY, USA, 2004. ACM Press.

[12] Carlos Cunha, Joao Sobral, and Miguel Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 134–145, New York, NY, USA, 2006. ACM Press.

[13] Simon Denier, Herv Albin-Amiot, and Pierre Cointe. Expression and composition of design patterns with aspects. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, 4, rue du Chteau de lraudire, 44324 Nantes France, 2006.

[14] J. Easton et al. *Patterns: Emerging Patterns for Enterprise Grids*. IBM Redbooks. IBM, June 2006. `http://publib-b.boulder.ibm.com/abstracts/sg246 682.html`.

[15] The Apache Software Foundation. Activemq. http://activemq.apache.org/.

[16] B De Fraine, W Vanderperren, D Suvee, and Johan Brichau. Jumping aspects revisited. In *DAW 2005*, Chicago, USA, 2005.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[18] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá; Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM Press.

[19] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[20] IBM. Ibm websphere application server. http://www.ibm.com/developerworks/websphere.

[21] JBoss. Jboss cache. http://labs.jboss.com/jbosscache/.

[22] JBoss. Jbossaop. http://labs.jboss.com/portal/jbossaop.

[23] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[24] Ralf Lämmel. Declarative aspect-oriented programming. In Olivier Danvy, editor, *Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, January 1999.

[25] Douglas Lea and Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[26] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM Press.

[27] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 13, New York, NY, USA, 2006. ACM Press.

[28] Microsoft. Microsoft dotnet. http://msdn2.microsoft.com/en-us/netframework/default.aspx.

[29] Sun Microsystems. Java platform, enterprise edition (java ee). http://java.sun.com/javaee/.

[30] Thomas J. Mowbray and Raphael C. Malveau. *CORBA design patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[31] Luis Daniel Benavides Navarro, Christa Schwanninger, Robert Sobotzik, and Mario Südholt. ATOLL: aspect-oriented toll system. In *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, page 7, New York, NY, USA, 2007. ACM Press.

[32] Luis Daniel Benavides Navarro, Mario Sdholt, Wim Vanderperren, and Bart Verheecke. Modularization of distributed web services using Aspects With Explicit Distribution (AWED), 2006.

[33] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed aop, 2004.

[34] Object Management Group OMG. Corba common object request broker architecture. www.omg.org/corba/.

[35] University of Waterloo Programming Languages Group. http://plg.uwaterloo.ca/%7Eusystem/uC++.html.

[36] Bo I. Sandén. Concurrent design patterns for resource sharing. In *TRI-Ada '97: Proceedings of the conference on TRI-Ada '97*, pages 173–183, New York, NY, USA, 1997. ACM Press.

[37] Douglas Schmidt. Design patterns, pattern languages, and frameworks, 09 2006.

[38] Smalltalk. Smalltalk. http://www.smalltalk.org.

[39] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190, New York, NY, USA, 2002. ACM Press.

[40] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[41] System and Vrije Universiteit Brussel Software Engineering Lab. Jasco. http://ssel.vub.ac.be/jasco/.

[42] Éric Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089, pages 98–113, Vienna, Austria, March 2006.

[43] Éric Tanter. An extensible kernel language for AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, 2006.

[44] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed aop. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, 2006. Springer-Verlag.

[45] Rodolfo Toledo, Éric Tanter, José Piquer, Denis Caromel, and Mario Leyton. Using ReflexD for a Grid solution to the n-queens problem. In *Proceedings of the CoreGRID Integration Workshop*, pages 37–48, Cracow, Poland, October 2006.

[46] Wim Vanderperren and Davy Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In Robert Filman, Michael Haupt, Katharina Mehner, and Mira Mezini, editors, *DAW: Dynamic Aspects Workshop*, pages 120–134, March 2004.

[47] Eric G. Wagner. From algebras to programming languages. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 214–223, New York, NY, USA, 1973. ACM Press.

[48] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[49] Yasuhiko Yokote and Mario Tokoro. The design and implementation of concurrent smalltalk. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 331–340, New York, NY, USA, 1986. ACM Press.

[50] Uwe Zdun and Paris Avgeriou. Modeling architectural patterns using architectural primitives. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 133–146, New York, NY, USA, 2005. ACM Press.

# Appendix A

# Current AWED syntax

*// Aspects*

| | | |
|---|---|---|
| Asp | ::= | [Depl] [Inst] [Shar] `aspect` Id '{' {Decl} '}' |
| Depl | ::= | `single` \| `all` |
| Inst | ::= | `perthread` \| `perobject` \| `perclass` \| `perbinding` |
| Shar | ::= | `local` \| globalall \| `inst` \| `Group` |
| Decl | ::= | [Shar] JVarD \| PcDecl \| Ad |
| PcDecl | ::= | `pointcut` Id({Par}) : Pc |

*// Pointcuts*

| | | |
|---|---|---|
| Pc | ::= | `call`(MSig) \| `execution`(MSig) |
| | \| | `get`(FSig) \| `set`(FSig) |
| | \| | `cflow`(Pc) \| Seq |
| | \| | `host`(Group) \| `on`(Group[, Select]) |
| | \| | `args`({Arg}) \| `passbyval`({Id}) |
| | \| | `eq`(JExp, JExp) \| `if`(JExp) |
| | \| | `within`(Type) \| `target`({Type}) |
| | \| | Pc ‖ Pc \| Pc && Pc \| !Pc |
| Seq | ::= | [Id:] `seq`({Step}) \| `step`(Id,Id) |
| Step | ::= | [Id:] Pc [→ Target ] |
| Target | ::= | Id \| Id Target |
| Group | ::= | {Hosts} |
| Host | ::= | `localhost` \| `jphost` \| "ip:port" |
| | \| | GroupId |
| GroupId | ::= | String |
| Select | ::= | Jclass |

*// Advice*

| | | |
|---|---|---|
| Ad | ::= | [`syncex`] \| Pos({Par}) : PcAppl '{' {Body} '}' |
| Pos | ::= | `before` \| `after` \| `around` |
| PcAppl | ::= | Id({Par}) |
| Body | ::= | Jstmt \| `proceed`({Arg}) \| \| `localproceed`({Arg}) |
| | \| | `addGroup`(Group) \| `removeGroup`(Group) |

*// Standard rules (intensionally defined)*

| | | |
|---|---|---|
| *MSig, FSig* | ::= | *// method, field signatures (AspectJ-style)* |
| *Type* | ::= | *// type expressions* |
| *Arg, Par* | ::= | *// argument, parameter expressions (AspectJ-style)* |
| *Id* | ::= | *// identifier* |
| *Ip, Port* | ::= | *// integer expressions* |
| *JClass* | ::= | *// Java class name* |
| *JExp* | ::= | *// Java expressions* |
| *JStmt* | ::= | *// Java statement* |
| *JVarD* | ::= | *// Java variable declaration* |