Distribution as a Set of Cooperating Aspects

Position paper submitted to the ECOOP'2000 workshop
on Distributed Objects Programming Paradigms

Johan Fabry (Johan.Fabry@vub.ac.be)

March 23, 2000

1 Introduction

This paper describes work done in the Aspect-oriented Programming (AOP) community, whose goal is to simplify development of programs by enabling the application of the principle of separation of concerns where this was previously not possible.

The AOP community has performed work relating to a number of aspects of particular significance to the field of distributed computing: coordination, replication, and the handling of remote methods. The combination of these three aspects would allow distribution to be treated as a separate concern, which would significantly simplify the implementation and maintenance of distributed systems. Furthermore, as work is performed on more aspects relevant to distribution, this work could also be integrated, which would further simplify the implementation of distributed systems.

However, note that treating distribution as a separate concern does not mean that the programmer is totally unaware of the fact that the system is a distributed system. The program must still be structured accordingly, but the implementation and maintenance is simplified significantly by using aspects. This is due to the extension of the conceptual separation of the distribution concern into a separation at an implementation level.

2 Aspect-Oriented Programming

Almost all techniques aimed at simplifying software development share the concept of "divide and conquer": the software is decomposed into smaller pieces, which individually are easier to comprehend and manage, and these pieces are later combined into the full system. This *separation of concerns*[HVL95], when performed not only conceptually, but also at an implementation level, greatly enhances coding and maintenance.

However, an ideal separation of concerns is hard to achieve. This is because a number of 'special-purpose' concerns, such as concurrency, distribution, persistence, etc. ... can not be decomposed using the existing modularization techniques.

In *Aspect-Oriented Programming*[K$^+$96], the concerns which cross-cut these modules are called aspects. AOP allows the programmer not only to reason separately about the aspects, but also to implement them separate from the normal modules. This is achieved by specifying the code pertaining to an aspect in a separate aspect file, in a special aspect language. Once all modules and aspects are defined, a special tool, called an Aspect Weaver, combines these into executable code. The weaver is able to do this because it knows not only how each aspect can be transformed into code, but it also knows the relationships between the different aspects, and the correct way to combine them.

Of the aspects which have already been addressed, some are extremely relevant to the field of distributed systems. In the following section we will discuss concurrency, data replication and remote method invocations. The subsequent section will describe how these three aspects can be combined by an Aspect Weaver to produce correct code.

3 Relevant Aspects

Some AOP papers, when enumerating possible aspects, consider distribution as being an aspect. We feel however, that the concern of distribution is too broad to describe as one aspect, and therefore should be split up into a number of more concrete aspects. These aspects can later be composed in such a manner that the distribution concern is indeed handled separately, by these aspects.

We shall now discuss three aspects; concurrency, replication and remote methods, which we consider as being of prime importance to the distribution concern. For each aspect we shall give an overview of their respective weaver. This will allow us, in the following section, to envision a weaver which will combine these aspects.

3.1 Concurrency

The Concurrency aspect is one of the first aspects addressed by the AOP community. An early release of the well-known *AspectJ* package [Cor], an AOP extension to Java, included explicit support for concurrency, using the *Cool* language. In current versions of AspectJ, this explicit support has been dropped in favor of a more general aspect language. For the sake of clarity we will however discuss the early versions, i.e. using Cool.

A Cool program describes a set of Coordinator modules or *Coordinators*. Coordinators are 'helpers' for a class: they take care of thread synchronization over the methods of objects of that class, and have read-only knowledge of the state of those objects.

When a method invocation is requested on a class C, the coordinator of C will first check the exclusion constraints for the method. If any of these constraints are not met, the request is suspended until all constraints are met. When all constraints are met, the method is executed by the object. Note that, because the coordinator only coordinates method invocations, the smallest units of synchronization are methods.

Describing a coordinator in Cool implies describing its coordination strategy. A Coordinator declaration includes an number of *selfex* and *mutex* sets and a number of *MethodManagers*.

Methods included in a selfex set are self-exclusive: each method in a selfex set can only be executed by at most one thread at a time. A self-exclusive, directly or indirectly, recursive method, however, will not deadlock.

Methods included in a mutex set are mutually exclusive: if a method in a mutex set is executed by a thread, the other methods in this set cannot be executed by another thread. Mutual exclusion of a method M does not imply self-exclusion: while M is executed by a thread, other threads are also allowed to execute M.

MethodManagers allow further coordination between methods, using guarded suspension of threads. These guards are expressed as a boolean expression of the internal state of the coordinator. Not only must the selfex and mutex constraints be met, but also the given boolean expression must return true. If not, the thread will be suspended until all constraints are met.

Weaving this aspect with the base code is conceptually very straightforward. As the smallest units of synchronization are methods, it suffices to prepend and append a generated body of code to the method. This body of code is determined by the given Cool code and will ensure that the specified coordination strategy is used.

3.2 Replication

Our previous work included a framework for replication using AOP [Fab98a, Fab98b]. In this work we wanted to obtain a significantly higher degree of replication transparency, by treating replication as an aspect.

By the term replication we refer to a method of sharing data between different computers in a distributed system. A number of servers contain copies of the data, and clients can access this data through the network. Whenever changes are made to the data on one server, this server will ensure that these changes are propagated to the data on the other servers, keeping the data in a consistent state.

As in AspectJ we made an AOP extension to Java. The base algorithm is written in Java, the replication aspect is specified in a separate aspect language: *Dupe*, and errors specific to replication are handled in a second language: *Fix*.

We have chosen to only replicate the instance variables of given objects, not ruling out that the objects' behavior is also replicated in later work. We can consider cases where not all variables in an object need to be replicated. In these cases, replicating only the fields which need to be replicated will speed up the system. To provide this control over replicated objects, the Dupe aspect language allows the programmer to specify which fields of a class must be replicated, by simply enumerating the fields which must be replicated and which fields must not be.

Replication implies that accesses to the data are now performed on the network, and that they may therefore fail and throw an exception. Clearly, catching the exceptions should not be done in the base algorithm, because it should not be aware of the replication aspect. Therefore the programmer will want to specify different exception handlers for different kinds of replicated data, so appropriate action is taken.

A separate aspect language, Fix was created to allow exception handlers to be specified by the programmer. Using Fix, the programmer can optionally specify two exception handlers for each replicated variable, one for exceptions thrown when reading the data, and one for writing the data. Also exception handlers can be specified for when the client fails to create the network link to the replicated data upon instantiation of the object.

Again, the work of the aspect weaver is conceptually quite straightforward. What needs to be done is a redirection of the variable accesses of replicated variables. This is a four-step weaving process for every class of replicated objects. First all reads and writes to these variables are modified to calls to accessor and mutator methods, respectively. Second these accessors and mutators are added to the class, with as body a remote method invocation to the class which contains the replicated data. Third the class containing the replicated data is generated. Fourth and last, the code to set up the connection between the replicated object and the replicated data is inserted in the constructors.

Note that the exception handlers specified in Fix are contained in the generated accessor and mutator methods, and in the connection setup code.

Experiments with this system have shown that it greatly simplifies use of replication in distributed systems. To state that the data of a certain object has to be replicated, all that is needed are some specifications in a separate Dupe file, and possibly the specifications of exception handlers in a separate Fix file. It is clear that the concern of replication has been separated out, to a very large extent, by treating it as an aspect. Although we did not achieve full replication transparency, the large separation of concerns has resulted in a significantly higher ease of use.

In subsequent work we performed on AOP and distribution, we have chosen to address remote method invocations as a candidate for an aspect, which is discussed next.

3.3 Remote methods

Our more recent work [Fab99] attempts to simplify the use of remote method invocations in Java, without having to lose a degree of control which characterizes other methods.

Other solutions, of which we consider JavaParty [PZ97] to be a prime example, allow for an extremely large degree of distribution transparency, by making remote method invocations almost completely transparent. In other words, in the code of the program there is no distinction between a local method call and a remote method call.

However, this happens at a significant cost; some assumptions are made about remote method calls which restrict the applicability of the toolkit. A recurring, and extremely significant assumption which is usually made, is that no exceptions are thrown due to failures in the remote method calls (which we shall call remote exceptions from now on). However, as is argued in [GF99], remote exceptions, due to failures between caller and callee, are one of the defining problems of remote method invocations and may not be casually ignored.

Therefore, we have constructed a system, called *Distra* which significantly simplifies the use of remote method invocations by making them more transparent, while still allowing fine-grained control of these invocations. This is achieved by using AOP to define which classes are to be made remote, and to specify exception handlers for remote method invocations.

Note that similar work has since been reported detailing the use of AOP for exception handling [LVL99]. It includes handling exceptions due to remote method invocations, but because the scope of this work is much broader than ours, the handling of remote exceptions is performed in a more limited way.

Distra is an AOP extension to Java, it contains three extra aspect languages: *Repl*, *Fix* and *Serv*. The programmer specifies which class is remote, i.e. which objects are accessed through remote method invocations, using Repl. Fix is used to specify the exception handlers for remote exceptions (hence the reuse of the Fix name). Serv is used to specify on what remote classes are contained on which server machines.We shall now briefly discuss these languages.

Dist allows the programmer to easily specify that any Java class, for which the source is available, is to be made a remote class. Instances of remote classes can be created remotely (i.e. on a different computer) if a remote host is specified, either by a string, or a block of code returning a string.

Fix is designed to easy specification of exception handlers for remote exceptions. Exception handlers can be specified for errors occurring when trying to first contact the remote host, when trying to instantiate an object remotely, and when remote methods are being invoked.

Serv is used to define a number of servers. These servers will contain a number of remote classes of which instances can be created remotely by other machines. A Serv program consists of simple declarations of the remote classes contained by a number of remote hosts.

Weaving these aspects in the base code is a quite complicated process, but it can conceptually be simplified to the following three phases.

First, as the code generated by Distra uses RMI, remote interfaces are generated for the remote classes. These remote interfaces contain all non-private, non-static methods of the class.

Second, the contents of the remote class is moved into a 'server' class. In the original, 'client' class, the methods defined in the remote interface are replaced with methods which redirect the call to the 'server' class, making the 'client' class, essentially, a proxy. This proxy will be responsible for all extra processing required due to the remote method invocations. The 'server' class will perform the actual processing, and return the result, if any, to the proxy, which will return it to the caller. Note that the exception handlers declared in Fix, are included in the proxy class, more specifically in their respective redirecting methods.

Third, the server classes which will contain objects which have been instantiated remotely are generated from the Serv specification.

Using Distra is very straightforward. As in our work on replication, all that is needed to specify that an object is remote, is a simple declaration in separate Repl and Serv files, and possibly some exception handlers in a separate Fix file. Experiments with Distra have shown that this greatly simplifies the use of remote method invocations, while still allowing the programmer to specify the needed exception handlers. We feel we may state that it is indeed possible to achieve a large degree of distribution transparency, while still being in control of the crucial element which is exception handling.

Having discussed these three aspects, we can now reason about an aspect weaver which combines these three aspects, such that a large part of the concern of distribution may be handled separately from the other concerns in the program.

4 Combining These Aspects

In this section we will discuss how an aspect weaver which combines these three aspects may work. To determine how the aspect weaver must weave the code, we must first establish the interactions between the different aspects. When this is determined we can specify an outline of how the weaver should manipulate the code.

First, let us consider the interactions between the concurrency and the replication aspects. We feel we can state that replication is not affected by the concurrency aspect. Replication treats variable access, while concurrency handles method calls. Weaving the replication aspect in the code does indeed produce extra methods, but these will simply be ignored by the concurrency aspect. Similarly, we can state that the concurrency aspect is not affected by replication. While the method guards may use replicated variables of the object, this will happen transparently if replication is weaved after concurrency has been weaved into the code.

Second, let us study the concurrency and remote methods aspects. It is clear that concurrency is not affected by remote methods, as the concurrency control does not introduce extra methods, or make any method calls. However, the remote methods aspect is affected by the concurrency aspect. The issue here is where to place the concurrency control code which is inserted at the beginning and at the end of the methods. The most sensible option seems to place this code on the 'server' part of the object, so the concurrency control affects the concurrency which has arisen due to the distributed nature of the application, i.e. different clients using that remote object. Note that the concurrency code will automatically be placed on the 'server' object if the remote methods aspect is weaved after the concurrency aspect.

Third and last, we must look at the replication and remote methods aspects. Replication is affected by the remote method aspect, and vice-versa, as the methods of the objects are moved to the 'server' object, the accesses to the replicated variables must also be moved to the 'server' object. This can be achieved by weaving the remote methods aspect after the replication aspect, which ensures that the accessor and mutator methods used by the replication aspect are moved to the 'server' object. Note that error-handling of the replication aspect remains separate from error-handling of the remote methods aspect at implementation level which ensures that no unforseen interactions take place.

Having considered the interactions between the three aspects, we can now conceive an aspect weaver which would correctly weave these three aspects. These three aspects can be weaved by a three-pass weaver, whereby every pass consists of a, possibly slightly modified, version of a weaver for one of the three aspects. More concretely, we would first weave the concurrency into the code, second weave the replication, and third weave the remote methods.

By using these three aspects in conjunction, the programmer can now easily handle these three important elements of the distributed system separately. It is clear that the ease of use of the three aspects being combined will result in significantly simpler implementation of the distributed system.

We feel we can state that it should be possible to extend this combination even further, by adding other aspects relevant to the field of distribution. By carefully considering the interactions between the different aspects, as done above, it should be possible to create an even more advanced aspect weaver. As more aspects are incorporated, this weaver will simplify even more the implementation of distributed systems.

5 Conclusion

In this paper we discussed how the principle of separation of concerns can be applied to simplify development of distributed systems. By treating distribution as a separate concern, and implementing it by means of different aspects, it can be reasoned about separately both at design time and at implementation time, significantly easing design and development.

We have discussed three aspects which play a significant role in the concern of distribution; concurrency, replication, and remote methods, and have described the matching AOP languages and their weavers.

Subsequently, we investigated the interactions amongst these three aspects, and sketched an outline of a possible aspect weaver, based on the existing weavers, which would be able to integrate these three aspects into the system. This would significantly simplify the implementation of distributed systems by virtue of combining the increased ease of use of each aspect separately. Additionally, this weaver could be extended to manage extra aspects, which would further simplify the creation of a distributed system.

References

[Cor] XEROX Corporation. Aspectj.org web site. http://www.aspectj.org.

[Fab98a] Johan Fabry. A framework for replication using Aspect-oriented Programming. Licentiaatsthesis, Vrije Universiteit Brussel, Faculteit Wetenschappen - Departement Informatica, 1998.

[Fab98b] Johan Fabry. Replication as an aspect - the naming problem. In *Ecoop '98 Workshop Reader*, number 1543 in LNCS. Springer-Verlag, 1998.

[Fab99] Johan Fabry. Full distribution transparency, without losing control - AOP to the rescue. Master's thesis, Vrije Universiteit Brussel - Belgium, Faculty of Sciences, In Collaboration with Ecole des Mines de Nantes - France and Universidad De Chile - Chile, 1999.

[GF99] Rachid Guerraoi and Mohamed E. Fayad. OO Distributed programming is not Distributed OO Programming. *Communications of the ACM*, 42(4), April 1999.

[HVL95] Walter L. Hürsh and Cristina Videira Lopes. Separation of concerns, February 1995. College of Computer Science, Northeastern University.

[K$^+$96] Gregor Kiczales et al. Aspect-oriented programming, a position paper, 1996. Xerox Palo Alto Research Center.

[LVL99] Martin Lippert and Christina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. Technical report, XEROC PARC, Dec 1999.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11), 1997.