D^2AL

A design-based aspect language for distribution control

— Position Paper —
ECOOP '98 Workshop on Aspect-Oriented Programming (W15)

Ulrich Becker
ubecker@informatik.uni-erlangen.de

IMMD IV, University of Erlangen-Nürnberg
Martensstraße 1, D-91058 Erlangen, Germany

Abstract: It is generally accepted that object-oriented programming and distributed computing fit well together, because distribution can be easily integrated into object-oriented systems in a transparent manner. But while transparency is desirable with respect to code readability and reusability, it can cause severe performance problems, because the programmer loses control over distribution.

This paper presents an approach to solve this conflict with Aspect-Oriented Programming: The basic functionality of the program can be formulated completely distribution-transparent, whereas the programmer retains complete control over the distribution through the aspect language D^2AL. D^2AL differs from other aspect languages in that it is based on the design of the application, not on its implementation. We show that using the design as the basis of D^2AL gives access to abstractions that greatly increase the expressiveness, as compared to an implementation-based aspect language.

1 Introduction

It is generally accepted that object-oriented programming and distributed computing fit well together. Since all interactions between objects occur through method invocations, there is only a single mechanism that has to be extended to transparently introduce distribution into an object-oriented programming language. In such a programming language, the programmer can write distributed programs in much the same way as non distributed programs, because local and remote method invocations provide nearly the same syntax and semantics. But complete abstraction from distribution leads to software with extremely poor performance, because there will be many remote invocations, which take several orders of magnitude longer than local invocations. For an efficient distributed application, objects that interact heavily must be located together. Because this may change over time, object migration should be possible. Finally, it should be possible to replicate objects that interact with objects at many different places.

We believe that a detailed understanding of an application is necessary to decide which is the best distribution for it. Therefore, we think that neither automated analysis of the source code nor monitoring the application behaviour at runtime can lead to satisfactory results, but that instead the software developer should have explicit control over the distribution.

Some languages or architectures like Emerald [BHJ+88] and CORBA [OMG98] provide the programmer with explicit means to control migration or replication[1]. The problem with these approaches is that they introduce distribution control by adding special statements or operations to the programming lan-

guage that is used to specify the basic functionality. This leads to a mixture of code that implements the basic functionality with code that controls the distribution. This mixture makes the code harder to understand and to maintain, an effect that can generally be observed when different aspects are tangled [KLM+97]. Furthermore, these approaches limit reuse, because inheriting the *functionality* of a class also inherits the class's *distribution*, which may be inappropriate.

To solve this problem, we developed an aspect language for distribution specification, which we call *D²AL* (Design-based Distribution Aspect Language). Our approach enables the software developer to specify the desired distribution separately from the basic functionality, which is implemented in Java. What distinguishes *D²AL* from other approaches to distribution specification (and, more generally, from other aspect languages) is that it refers to the design instead of to the implementation of the basic functionality. This has several advantages:

First of all, modelling languages for object-oriented analysis and design can themselves be viewed as being aspect-oriented. They provide specialized notations to explicitly express aspects like object interactions, abstract states, or the static structure of the system. Some of these aspects, like object relations and abstract states, prove very useful and convenient for the specification of distribution requirements. But since the separate and explicit notation of these aspects gets lost during the implementation, they are accessible only in the design model.

Furthermore, single aspects like abstract states or object relations can be conveniently expressed through a single model element or a single diagram in the design, while they are spread over several operations or even over several classes in the implementation. Thus, an aspect language that is based on the design can refer to a single element where several elements in the implementation would be needed.

Using the design as the basis of an aspect language also helps to relax a problem generally found in AOP: Several specifications that describe different aspects of an application have to be kept consistent. Because the design abstracts from many implementation details, it is generally more stable then the implementation, which in turn makes changes to referring aspect specifications less often necessary.

The rest of this paper is organized as follows: Section 2 describes the basic features of *D²AL*. Section 3 outlines some issues in the construction of a weaver for *D²AL*. Section 4 gives the current status of the project, and Section 5 gives our conclusions.

2 The *D²AL* Aspect Language

The focus of *D²AL* is to provide means of minimizing the number of remote interactions by giving control over object distribution and replication. *D²AL* does not offer control over marshalling, like e.g. *D* does [LoKi97]. Our choice for the underlying design representation is the UML. At the moment, however, we do not use any features that are unique to the UML—other mainstream methodologies like the Booch method or OMT would also provide sufficient expressiveness.

An important feature of *D²AL* is that the programmer describes the desired distribution *relatively*, i.e. the distribution specification does not determine where an object should reside, but whether two objects should be located on the same node. This notion of *relative distribution* is based on the work of Fäustle ([Fäus92a],[Fäus92b]). The design concept that we use as the basis for the specification of relative dis-

1. CORBA defines only the interfaces for migration and replication, while the mechanisms have to be implemented by the application- or ORB implementor

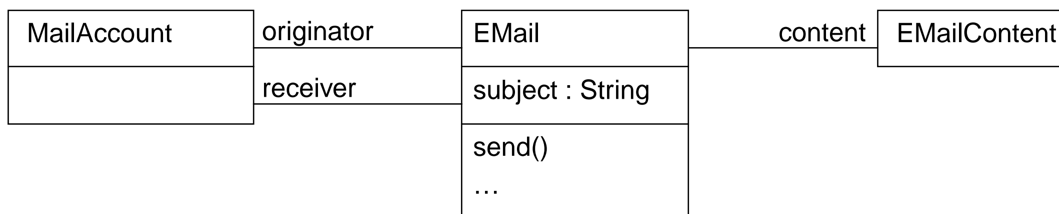

Fig. 2.1 Class diagram of an electronic mail system

tribution requirements is the *collaboration*. In accordance with the UML, we define a collaboration as a group of objects that interact to accomplish a particular task, such as a use case or an operation. Since there is typically a lot of interaction among the members of a collaboration, they are an obvious choice for objects that should be considered for collocation. D^2AL provides direct support for the collocation of collaboration members: The programmer can describe collaborations in D^2AL, and state for every collaboration whether its members should be collocated or not.

In D^2AL, the specification of collaborations is based on either associations or other, more transient relations between objects. Associations are used to model object relations with a semantic background, like the relation between an e-mail and its sender, instead of just a uses relationship. Since interacting objects are often semantically related, associations are well suited for the description of collaborations. Actually, the availability of associations in the design model is one of the primary reasons why we have chosen the design as the basis for the distribution specification: Although object relations are a central aspect of object-oriented software, none of the programming languages that are widely used today offers any support for them. The reason for this is that object relations cross the border of a single object, which is the single abstraction provided by these programming languages. As a consequence, the programmer has to circumvent this shortcoming by implementing associations with primitive language elements like pointers, references, or container-objects in one or both of the related objects. In the implementation, it is therefore not immediately visible anymore what relations exist between objects. Furthermore, how an association is implemented is an implementation detail that is likely to change. For this reason, the implementation is not a good basis to describe collaborations.

To see how collaborations are specified and used in D^2AL, consider the class diagram for a simple e-mail system given in Fig. 2.1. In this design, an EMail object is separated from the EMailContent, which makes it easy to send different kinds of content by simply using specific subclasses of EMailContent. While this separation makes sense from a design perspective, it usually makes little sense to handle an EMail and the accompanying content-object as separate distribution entities. Instead, they should always be collocated. In D^2AL, the following expression can be used to achieve this behaviour:

```
collaboration EMailCluster {
    link Content;
    participants EMail, EMailContent;
    distribution collocation;
}
```

The expression describes that the objects participating in this collaboration are related via an association named “content“, and that the objects have the types EMail and EMailContent, respectively. Finally, it is said that these objects should be collocated.

Whether two objects should be collocated depends not only on their relation, but also on their type. Consider an e-mail that contains large amounts of data, like a sound recording: In that case, one probably would not want to transfer the content in advance, but only when it is needed. For that purpose, the programmer can specify another collaboration that specifies `SoundRecording` as type of the second participant and `indifferent` as distribution, effectively neutralizing the above specification for sound recordings.

While there is often a semantic relationship between interacting objects, this is not necessarily the case. Consider an object that provides a name service for a distributed system: Clearly, many objects will require this service and therefore interact with the name service object, but one would not normally model this with an association. For this reason, D^2AL also supports collaborations that are based on object references that were passed as method arguments or that are stored in local variables or instance variables. The user specifies these collaborations simply by supplying a different `link`-form.

With the D^2AL-expressions described so far, the programmer can only specify relatively static collaborations, because begin and end of a collaboration are always bound to the begin and end of a relation. But an object may collaborate with different objects even though its relations remain unchanged. Consider the e-mail example: The relations between an e-mail and its originator and receiver will persist for the whole lifetime of the e-mail message. But until the e-mail is sent, most interactions will be with objects at the originator's site, while after sending the e-mail, most interactions will be with objects at the receivers site. Therefore, the e-mail should be migrated to the receiver after sending it.

D^2AL provides the ability to specify such dynamic distribution requirements by including the *states* of the participating objects into the specification of a collaboration. What we mean here with state is not the implementation-related notion of an object's state but rather the conceptual model of an object's state that the designer or programmer has. Consider the example given in Fig. 2.2, which shows a state diagram for the class EMail: `Draft` is the state where the EMail has not yet been sent, `Transferring` where the send operation has been called but is not finished yet, and `Delivered` and `Undeliverable` where the execution of send has completed with or without success, respectively. The programmer can now use these states in a collaboration specification by adding constraints on the abstract state for the members of a collaboration. Thus, the desired migration of an e-mail could be specified through two collaborations that are constrained on the EMail's state: One collaboration with the originator's `MailAccount` that is constrained on the states `Draft` and `Undeliverable`, and another collaboration with the receiver's `MailAccount` that is constrained on the state `Delivered`.

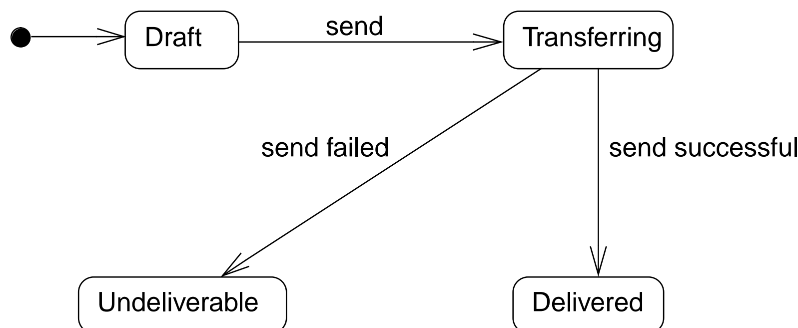

Fig. 2.2 State diagram for the class EMail

As an alternative to migrating the e-mail to the receiver, the runtime system could also replicate the e-mail at the receiver's site. In the case of several receivers at different sites, this would in fact be the only way to satisfy the required distribution. Since no changes are made to an e-mail once it has been sent, replication does not require any special mechanisms for keeping the replicas consistent.

D^2AL handles replication by distinguishing between replicable and non-replicable states: By declaring a state as `replicable`, the programmer guarantees that replicating an object that has reached this state does not lead to any inconsistencies. Obviously, only other replicable states may be reached from such a state. The replication of an `EMail` could then be enabled through the following *D^2AL*-expression:

```
states EMail {
    replicable {Transferring, Undeliverable, Delivered};
}
```

It should be noted that the programmer does not actively replicate the object, as he would do e.g. in CORBA with the `copy`-operation, but that he only states that it can be safely replicated. The decision whether an object is replicated or migrated to satisfy a collocation is made by the runtime system, which knows all current collocation requirements and thus has the information that is necessary to make the right decision.

Typically, a state is replicable if the object is not changed anymore, or if any changes occurring in this state do not impose any consistency problem. Replication of mutable objects that require consistency protocols is currently out of the scope of *D^2AL*. Nevertheless, we expect that the way in which we support replication can significantly improve the performance of distributed applications.

3 The aspect weaver for *D^2AL*

While using the design as the basis for *D^2AL* greatly contributes to its expressiveness, it also means that the weaver has to bridge the gap between design and implementation, since the basic functionality is specified using a normal programming language. In this section, we will outline how this can be achieved.

Our weaver works as a preprocessor that transforms distribution transparent Java code into Java code that, when compiled and executed, is distributed as described in the *D^2AL*-specification. To achieve this, the runtime system must be able to keep track of events at the design level that might be relevant to the distribution, like a state change. At every place in the source code where such an event might be caused, the preprocessor inserts instructions that pass the relevant information to the runtime system. The runtime system can then in turn react to this event, e.g. by migrating an object. The problem is that some of these design elements, e.g. associations and abstract states, have no direct mapping to the implementation. Therefore, the weaver needs not only the Java code and the *D^2AL*-specification, but also a representation of the design and a mapping that specifies how these design elements have been implemented, so that it can identify the relevant places in the source code.

Associations are one of these design elements for which no direct mapping exists. But keeping track of associations is relatively simple, because they are usually implemented with references that are stored in instance variables or container objects, depending on the multiplicity of the association. In that case, the programmer must only provide a mapping between associations and the instance variables that are used

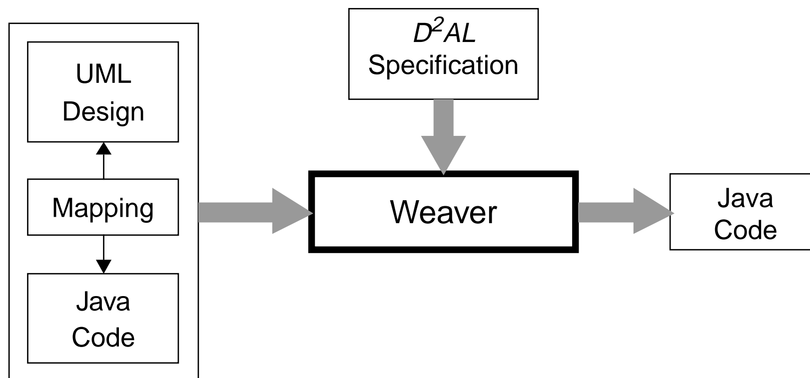

Fig. 3.1 Architecture of the D^2AL weaver

to store the references. Because most design tools can automatically generate code, the use of a design tool can eliminate the need for a manual mapping.

The abstract state of an object also lacks a direct mapping to the implementation. Our approach to providing such a mapping is based on the assumption that every transition between two abstract states must be accompanied by some event in the implementation, such as an operation invocation or an exception. Since the state machine is an abstraction of the implementation, this is a reasonable assumption. For every transition in the abstract state machine, the programmer provides one or more implementation-level events that trigger this transition, thus giving a mapping from abstract events to events in the implementation. In the state diagram for an `EMail` given in Fig. 2.2, the event *send* would be mapped to the invocation of send(), while *send failed* would be mapped to some exceptions that can occur during send().

Every state diagram that is used in D^2AL is implemented through a Java class that is generated by the weaver. This class is independent from the mapping. The weaver uses the mapping to insert the appropriate instructions into the source code that pass the respective abstract event to the abstract state machine. Our approach allows the programmer to implement the class in the same way that he would normally do it, instead of forcing him to explicitly represent the abstract state in his implementation.

4 Status of the project

We are currently implementing a prototype of the weaver that can handle the D^2AL-expressions described in this paper. The purpose of this prototype is to study the problems that are caused by weaving languages at different abstraction levels, and to validate the suitability of D^2AL for practical applications. One area for future research is to explore whether interaction diagrams could be used to describe highly dynamic distribution requirements.

5 Conclusion

We have shown how aspect-oriented programming can be used to give the programmer detailed control over the distribution of an application's distribution, while at the same time retaining the advantages of distribution-transparent programming, such as reusable and understandable code. We presented the aspect language D^2AL, which enables the programmer to specify the desired distribution of an application

based on its design. We could show that the use of the design increases the expressiveness of D^2AL, because the design contains abstractions that are well suited for the specification of distribution requirements, but that get lost during the implementation phase.

6 References

BHJ+88 A. Black, N. Hutchinson, E. Jul, H. Levy, *Fine-Grained Mobility in the Emerald System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988

BJR97 G. Booch, I. Jacobson, and J. Rumbaugh: *UML Notation Guide*, version 1.1, September 1997

Fäus92a M. Fäustle: *An orthogonal distribution language for uniform object-oriented languages*. Techn. Report TR-I4-92-06, Univ. of Erlangen-Nürnberg, IMMD IV, October 1992.

Fäus92b M. Fäustle: *Beschreibung der Verteilung in objektorientierten Systemen,* Dissertation, Univ. of Erlangen-Nürnberg, IMMD IV, 1992.

KLM+97 G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin: *Aspect-oriented programming*. Techn. Report SPL97-008 P9710042, Xerox Palo Alto Research Center, February 1997

LoKi97 C. Lopes, G. Kiczales, *D: A Language Framework for Distributed Programming*, Techn. report SPL97-010 P9710047, Xerox Palo Alto Research Center, February 1997

OMG98 Object Management Group: *The Common Object Request Broker Architecture*, Version 2.2, February 1998