D^2AL - A design-based aspect language for distribution control

U. Becker, F. J. Hauck, J. Kleinöder

University of Erlangen-Nürnberg, Germany
{ubecker, hauck, kleinoeder}@informatik.uni-erlangen.de

Abstract. The D^2AL aspect language gives the programmer control over the distribution of application objects. Unlike other aspect languages, D^2AL is based on the design of the application, not on the implementation. This approach increases the expressiveness of the aspect language as well as the maintainability of the aspect program.

1 Motivation

Distribution transparency is generally desirable with respect to ease of programming, reusability and readability. But due to the high costs of remote interactions, the programmer needs control over the distribution to deliver efficient applications. AOP [2] enables the programmer to combine both needs: The basic functionality can be implemented in a distribution transparent way, while the desired distribution is specified in a separate aspect program. The aspect language D^2AL lets the programmer describe which objects should be collocated, and when objects can be safely replicated. Unlike most other aspect languages, D^2AL is based on the UML [1] model of the application instead of the implementation. One advantage of this approach is increased expressiveness: Not all information in the design can be expressed in current object-oriented programming languages. As an example, neither associations nor state diagrams have a direct counterpart in the implementation, but both are suitable abstractions for the distribution specification. Therefore, enabling their use in the aspect program increases the expressiveness of D^2AL as compared to implementation-based aspect languages. Another advantage is that the design is more abstract than the implementation and is therefore changed less often, which makes it less often necessary to adapt the aspect program to changes in the component program.

2 The D^2AL aspect language

In D^2AL , the collocation of objects is described on the basis of *collaborations*. Every collaboration specification describes when a pair of objects forms an instance of this collaboration, and whether these objects should be collocated or not. Collaborations are primarily based on links between objects, where a link may either be based on an association or on another, more transient relationship between objects. The collaboration specification may contain additional constraints concerning the types and abstract states of the participating objects, which enables the programmer to specificy dynamically changing collocations. Whenever two objects form an instance of a collaboration whose members should be collocated, the runtime system tries to fulfill the distribution requirement by migrating one of these objects to the location of the other one.

As an alternative to migration, objects could also be replicated to fulfill a collocation requirement. While replication requires special consistency protocols in the general case, this is not necessary if an object does not undergo any semantically relevant changes any longer. D^2AL supports replication in the latter case by allowing the programmer to declare states as *replicable*: Once an object has reached a replicable state, the programmer guarantees that it can be safely replicated without any consistency protocol. The runtime system can then choose to replicate the object to fulfill a collocation requirement.

3 The D^2AL weaver

While using a design-based approach increases the expressiveness of D^2AL , it also means that the join points are not as obvious as for an implementation-based aspect language. For every design element used in an aspect program, the weaver has to identify the corresponding join points in the implementation. To achieve this, the programmer has to provide a mapping between these design elements and the corresponding code in the implementation.

For every association, the weaver has to know through which fields it is implemented. At every place where these fields are set, the weaver can then weave-in code that keeps track of the association. We expect that in most cases, the programmer does not have to supply this mapping manually: Since most design tools can automatically generate code in a predictable way, no manual mapping is needed when such a tool is used. If no tool support is available, naming conventions can be used to derive a large part of the mapping from the implementation itself.

For every class whose state diagram is used in the distribution specification, the weaver generates a state-machine class that implements the state diagram. Every instance of a class is then associated with an instance of the corresponding state-machine class that keeps track of the abstract state. The programmer has to provide a mapping between the abstract events from the the state diagram and the corresponding implementation-level events. The implementation-level events that can be used in the mapping are in particular exceptions and the begin and end of method invocations. The weaver can then weave-in code that passes abstract events to the state-machine object at the appropriate points.

References

1. G. Booch, I. Jacobson, and J. Rumbaugh: UML Notation Guide, version 1.1 (1997)
2. 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 (1997)