

# D: A Language Framework for Distributed Programming

Cristina Videira Lopes

PhD Thesis, College of Computer Science, Northeastern University. November 1997.

© Copyright 1997 Xerox Corporation. All rights reserved.

# Chapter 6

## Conclusions

“This is what I mean by "focusing one’s attention upon a certain aspect"; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts that I know of.”

Edsger Dijkstra in “*A discipline of programming*” [18]

**6. Conclusions**

## 6.1. Summary

The design and implementation of distributed systems requires addressing a number of issues that do not arise in non-distributed systems. Two of the most important are the synchronization of concurrent threads and the application-level data transfers between execution spaces. At the design level, addressing these issues typically requires analyzing the components under a different perspective than is required to analyze the functionality. Very often, it also involves analyzing several components at the same time, because of the way those two issues cross-cut the units of functionality. At the implementation level, existing programming languages fail to provide adequate support for programming in terms of these different and cross-cutting perspectives. The result is that the programming of synchronization and remote data transfers ends up being tangled throughout the components code in more or less arbitrary ways.

This thesis presents a language framework called D that effectively untangles the implementation of synchronization schemes and remote data transfers from the implementation of the components. In the D framework there are three kinds of modules: (1) classes, which are used to implement functional components, and are clear of code dealing with the aspects; (2) coordinators, which concentrate the code for dealing with the thread synchronization aspect; and (3) portals which concentrate the code for dealing with the aspect of application-level data transfers over remote method invocations.

To support this separation, D provides two aspect-specific languages: COOL, for programming the coordinators, and RIDL, for programming the portals. COOL and RIDL were designed to address the specific needs of the two kinds of aspects. COOL and RIDL can be integrated with existing object-oriented languages, with little or no modifications to that language. COOL's coordinators and RIDL's portals compose with the classes through the classes' "aspect interfaces." Aspect interfaces are quite different than normal client interfaces but have some of the flavor of specialization interfaces.

D leads to programs whose modules are more focused and where the separation of concerns is more clear than it would be using traditional object-oriented languages. Often, D programs are smaller as well. D programs can be efficient -- the performance penalty of the framework is very low. In alpha-user experiments, programmers reported not only that they understood the aspect interfaces and the aspect languages well, but also that, having classes, coordinators and portals,

helped them to focus on different issues at different times, and that this was of great help in the development of applications.

## 6.2. Contributions

This thesis makes a number of contributions that can have an immediate impact on the design, implementation and documentation of distributed applications. It also makes contributions that, in the longer term, may affect the design of programming languages.

The concrete and most immediately useful contribution is DJ, the integration of D with Java™. This dissertation described one implementation of DJ that can be easily reproduced, either partially or in its entirety, and that performs within acceptable bounds.

But the most important contribution of this work is the design of an enforceable support for programming thread synchronization and application-level remote data transfers in separate from the implementation of the components, while using an ordinary object-oriented language for programming the components. The two new kinds of modules in D, coordinators and portals, are additions with respect to the class modules, and they control the behavior of the classes in concurrent and distributed environments. Coordinators and portals do not pursue the goal of being abstract descriptions that can be used by many different classes; instead, they simply aim for an effective separation of concerns. And, unlike reflective approaches, this separation is enforced by aspect-specific languages.

Systems like CORBA and Java RMI use the object-oriented composition mechanisms (inheritance and type implementation, respectively) to integrate remoteness with the OOPL. RIDL breaks away from the state-of-the art type-based remote interaction for distributed object systems by providing new abstraction and composition mechanisms that support a better division of labor and that capture a lot more of the issues involved in remote interaction than types can ever capture. These new mechanisms are equally well integrated with the OOPL. The declarative nature of the language makes it very easy to write relatively complex data transfer schemes.

Based on the study of the synchronization needs of concurrent object systems, COOL provides a small number of powerful language constructs that address those needs in a succinct way. The selfex and mutex declarations capture, to a large extent, the most common needs of multi-threaded object-oriented programs.

Coordinators and portals also improve the documentation of the applications. When using other languages, the tangling between functionality and aspect code makes it extremely difficult to understand *what* are the concurrency control schemes and data transfer protocols affecting those components. Coordinators and portals isolate and describe those issues. A positive side-effect of this has to do with user-written documentation (i.e. comments) that explains *why* those issues are programmed in a certain way. When using other languages, the documentation about the implementation decisions related to the distribution issues is frankly bad, and, in most cases, it simply does not exist. However, in the applications that were written in DJ, programmers were compelled to explain the decisions in the implementation of the aspect modules at least as well as the decisions in the implementation of the classes.

Another contribution of this thesis is the extensive study of code tangling with respect to the current programming practices and to the existing programming languages. This study provides the background for understanding some of the software engineering problems that programmers are faced with when developing distributed applications, and some of the solutions that they can apply. This study is a methodological research that sets up the motivation for “better” languages that, as Wulf puts it, “permit and even encourage the use of “good” program structures” [75]. Similar studies can be done for issues other than synchronization and remote data transfers.

In the longer term, D and this thesis suggest interesting directions for language design. The aspect languages were designed without having to modify or extend the component language. This is considerably different from all the previous approaches, where either completely new languages or extensions to the existing ones have been proposed. The approach taken here has one major benefit. The classes are programmed without explicit commitments to the aspect modules; therefore a class can be tested for what it does, independently of whether it will be used with D or not. The preliminary alpha-users study showed that, although the programmers had distribution in mind from the beginning, they first used and tested their classes in a non-distributed environment, and only then added coordinators and portals. The non-intrusion of the aspects into the component modules seems to be a useful design decision: programmers can simply “plug-in” or “plug-out” the aspect modules whenever they want.

In object-oriented languages, subclassing already established a relationship between modules that is not of the type client/provider. D went one step further, and showed that it is possible to establish many other kinds of interfaces between modules that are quite different from the normal

client interfaces, but that are, nonetheless, extremely useful for structuring programs. This suggests one language design direction in which aspects are identified and aspect languages are added.

### 6.3. Future Work

A language framework very similar to DJ was implemented at the Xerox Palo Alto Research Center, and this implementation has been in alpha-usage since June of 1997. We plan to pursue the development and improvement of this framework, in many ways.

First, we intend to fix some design problems of D that were already detected. In RIDL, we need to include nested copying directives and clarify the relation between the components' subclassing relations and the types of the parameters that are passed.

Secondly, we intend to improve and extend the existing aspect languages. Some feature requests made by the alpha-users include support for replication, timeouts and new parameter passing semantics. One issue that will be further researched is the connection between types and RIDL. Another issue that will be investigated is the possibility of making the aspect languages more imperative than what they are now. Currently, they are mostly declarative, with the exception of the guarded suspension/notification in COOL. But it may be possible to add imperative features that give more flexibility to the languages, in particular RIDL. A third issue that needs urgent attention is error handling.

We intend to identify other aspects and design new aspect languages following the methodology in this thesis. That is, first we will look at many more Java programs in order to identify other kinds of code tangles; then we need to understand what are the good program structures that minimize those tangles — this gives us hints for what kinds of things an aspect language needs to be able to express; next, we design an aspect language and provide a weaving engine for it.

Finally, at the implementation level, the current weaver is strongly coupled with the two aspect languages. We plan to develop a generic weaver that can easily process the new aspect languages that will be designed.

### 6.4. Conclusion

This thesis has demonstrated that aspect-oriented programming is possible and useful. By separating the program modules into aspects and components, important issues that would otherwise be

diluted in the program texts become visible and with well-localized effects. The particular aspect-oriented framework, D, has proven to be useful for small to medium size programs, and there is good reason to believe that its benefits will be even greater for larger, more complex programs.

Personally, I have learned that modularity means a lot more than dividing designs into components and implementations into classes. Better modularity can be achieved by including new kinds of modules that compose in new kinds of ways with each other and with the components, as long as those modules align well with issues in the design.

As important as the solution presented here, are the questions that this thesis raises. What other issues can be thought of as aspects? Are there other domains in which aspects can be useful? Is there a systematic way of defining aspect interfaces? How can we debug aspect-oriented programs and what kinds of visual programming interfaces would be appropriate? How do aspect modules scale? Would it be of any use to divide a software development team into component and aspect experts? The experience gained during the design and implementation of D and the many discussions it generated, gave me and my two groups, the AOP group at PARC and the Demeter group at Northeastern University, precious insights that will allow us, and the rest of the research community, to investigate this idea even further.

