Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming

Gianpaolo Cugola, Carlo Ghezzi, and Mattia Monga

Politecnico di Milano – Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32
I 20133 Milano – Italy
{cugola, ghezzi, monga}@elet.polimi.it

Abstract. In this paper we discuss how Aspect-Oriented Programming (AOP) may be useful in the design of distributed applications. Different design paradigms might be singled out in separate aspects, thus separating the "functional" code of an application from the interchangeable parts describing the distribution.
After introducing an Aspect-Oriented Language (AspectJ), we illustrate its use in a simple toy example. This will allow us to identify how research should progress to make AOP a useful approach in the design of a distributed application

1 Motivations

Since the beginning of software engineering, separation of concerns [1] was recognized as a fundamental tool to manage the complexity of systems. A complex system can be partitioned into simpler, tractable subsystems. Sub-problems are addressed relatively independently and the complete solution is built by gluing together the sub-solutions. The parts that compose the whole system are often modular units of functionality and existing programming languages work well in isolating modules, thanks to object orientation, functional decomposition, etc.
However, sometimes a concern is not easily factored out in a functional unit, because it *cross-cuts* the entire system, or parts of it. Synchronization, memory management, network distribution, load balancing, error checking, profiling, security are all *aspects* of computer problems that are unlikely to be separated in functional units.
As an example, suppose that a Java class is used to describe the pure functionality of certain objects. Separate aspects, not easily encapsulable in a module, may include:

- the definition of how objects are distributed on the nodes of a network and which pattern they use to communicate;
- the definition of synchronization operations to constrain concurrent access to the object (e.g., a consumer trying to read a datum from a queue must be suspended if the queue is empty);

– the definition of security or accounting policies (e.g., to get information from an object one must first ask some permission).

A language supporting the separate definition of pure functionality from the various different aspects is called an Aspect-Oriented Language (AOL). An AOL should satisfy a number of desirable properties. For example, each aspect should be clearly identifiable; it should be self-contained and easily changeable. Moreover, the various aspects should not interfere with one another. They should not interfere with the features used to define and evolve functionality, such as inheritance.

This report is organized as follows: Section 2 introduces the current stage of Aspect-Oriented Programming. Section 3 uses Aspect-Oriented Programming to applying different paradigms to a distributed application, highlighting problems and pitfalls. Section 4 provides some conclusions and motivates future briefly sets a research agenda.

2 A Brief Introduction to Aspect-Oriented Programming

The central idea of Aspect-Oriented Programming (AOP) [2] is to separate the code that expresses an aspect (i.e., a property of the system not cleanly separable in a functional unit) from the code that expresses functional units. A weaver braids (not necessarily at compile-time) aspects with functional units to obtain the final system. Aspects are expressed by the means of an Aspect-Oriented Language, while functional units are defined with a Component Language (CL). There can be a different AOL for each kind of aspect one wants to cope with.

In the study reported here we analyzed AspectJ, which can be considered as a representative of current AOLs. AspectJ 0.1 [3] is an environment for aspect programming developed at Xerox PARC. In the environment, the CL is Java and there is an AOL for synchronization (COOL) and an AOL for expressing remote invocation (RIDL). In the new version (0.3) of AspectJ [4], there is a unique general-purpose AOL that captures the cross-cutting nature of aspects, independent of what those aspects are.

3 Problems and Pitfalls

In this section, we describe an experiment in using AOP to encode different paradigms of distribution. Examples are implemented with AspectJ 0.3, Java 1.2 and RMI; we discuss a number of drawbacks and pitfalls of the solutions. More generally, our remarks enlighten the weaknesses of the current state of the art in AOLs and indicate directions of future investigation.

When implementing a distributed application, the concept of distribution of modular components among locations needs to be taken explicitly into account [5]. In conventional Object-Oriented Programming (OOP), changing the design decision about distribution is never easy, since the effects of such decision are tangled all around the code. On the other

hand, AOP offers an appealing approach since it allows the design decisions regarding different distributions policies to be specified separately, making it easy to design them and to switch from one to another.

In our research group, previous work identified and analyzed a number of design paradigms that can be chosen to develop a distributed application [6]. In particular, [6] discusses the design paradigms through the metaphor of two friends - Luise and Christine - cooperating in the task of making a cake. In order to make the cake (service), a recipe (know-how) is needed, as well as the ingredients (*light* resources) and an oven (*heavy* resource) to bake the cake. In a *client-server* (Figure 3) situation we may want that Louise asks Christine to bake the cake for her. Suppose we want change the environment towards *remote-evaluation* (Figure 4); now Louise owns the know-how and communicates it to Christine to get the cake baked. A third option (*Code on Demand*, Figure 5) is that Louise asks Christine for the recipe and bakes the cake herself. By using an AOL, our goal is to 'isolate' the code that implements the above different situations, in a way that makes further changes easy.

The core code (see Figure 1) knows about the use of RMI, but it is independent of the design decision regarding distribution. Louise (see Figure 2) is the initiator of the interaction with Christine and the one interested in the final result, the cake.

```
public class Person
    extends UnicastRemoteObject
    implements PersonI{

    protected Ingredients ing;
    protected Oven ov;
    protected CookBook book;

    public Person()
        throws RemoteException {}

    public CookBook getBook()
        throws RemoteException{
        return book;
    }

    public void setBook(CookBook cb)
        throws RemoteException{
        book = cb;
    }

    public Cake getCake()
        throws RemoteException{
        return makeCake();
    }

    private Cake makeCake(){
        return (Cake)
            book.prepare("Cake",ing,ov); }
    }
}
```

```
interface PersonI
    extends Remote{
    CookBook getBook()
        throws RemoteException;
    void setBook(CookBook cb)
        throws RemoteException;
    Cake getCake() throws
        RemoteException;
}

class Food {/* ... */}

class Ingredients{/* ... */}

class Cake extends Food {/* ... */}

class Oven{
    public Food bake(Food ww){
        /* ... */
    }
}

interface CookBook{
    public Food prepare(String foodName,
                        Ingredients ii,
                        Oven oo);
```

Fig. 1. Core classes without any distribution issues

```
class Louise extends Person{
    private PersonI p;
    public Louise()
        throws RemoteException{
        try {
            p = (PersonI)
                Naming.lookup("Christine");
        }
        catch(Exception e) {}
    }

    public static void main(String[] arg){
        try {
            Louise l = new Louise();
            l.act();
        }catch(Exception e) {}
    }

    private void act() {}
}
```

```
class Christine extends Person{
    public Christine()
        throws RemoteException{}

    public static
        void main(String[] arg){
        try {
            Christine me =
                new Christine();
            Naming.bind("Christine",me);
        }
        catch(Exception e) {}
    }
}
```

Fig. 2. Louise and Christine

```
aspect ClientServer{
    advise void Louise.act(){
        static before{
            try {
                Cake c = p.getCake();
            }
            catch(Exception e) {}
        }
    }

    advise Christine(){
        static after{
            ing = new Ingredients();
            ov = new Oven();

            class Artusi implements CookBook{
                public Food prepare(String foodName,
                                    Ingredients ii,
                                    Oven o){
                    /*...*/
                }
            }
            book = new Artusi();
        }
    }
}
```

Fig. 3. Client-Server

In our analysis of current AOLs, we found three main drawbacks:

```
aspect RemoteEvaluation{
    advise Louise(){
        static after{
            class Artusi implements CookBook{
                public Food prepare(String foodName,
                                    Ingredients ii,
                                    Oven o){
                    /*...*/
                }
            }
            book = new Artusi();
        }
    }

    advise void Louise.act(){
        static before{
            try {
                p.setBook(book);
                Cake c = p.getCake();
            }
            catch(Exception e) {}
        }
    }

    advise Christine(){
        static after{
            ing = new Ingredients();
            ov = new Oven();
        }
    }
}
```

Fig. 4. Remote Evaluation

1. Possible clashes between functional code (expressed using a CL) and other aspects (expressed using one or more AOLs). Usually such clashes result from the need of breaking encapsulation of functional units to implement a different aspect. As an example, the `RemoteEvaluation` aspect (see Figure 4), modifies the value of the protected variables `ing`, `ov` and `book` of `Person`. This results in a potentially dangerous violation of class encapsulation. As a consequence, in general, it is not possible to change the internals of a functional unit without affecting aspects.
2. Possible clashes between different aspects. Suppose we developed an aspect `TraceBefore` to trace the start of execution of methods of class `Person` and an aspect `TraceAfter` to trace the end of execution of the same methods. The two aspects work perfectly when applied individually (for example, to trace the start of execution or to trace the end of it). Unfortunately, if each aspect introduces the same method (e.g., method `print`) with different definitions, they fail when applied together.
3. Possible clashes between aspect code and specific language mechanisms. One of the best known examples of problems that falls into this category is *inheritance anomaly* [7]. This term was first used

```
aspect CodeOnDemand{
    advise Louise(){
        static after{
                ing = new Ingredients();
                ov = new Oven();
        }

    }

    advise void Louise.act(){
        static before{
            try{
                book = p.getBook();
                Cake c = getCake();
            }
            catch(Exception e) {}
        }
    }

    advise Christine(){
        static after{
            class Artusi implements CookBook{
                public Food prepare(String foodName,
                                    Ingredients ii,
                                    Oven o){
                    /*...*/
                }
            }
            book = new Artusi();
        }
    }
}
```

Fig. 5. Code on Demand

in the area of concurrent object-oriented languages [8–10] to indicate the difficulty of inheriting the code used to implement the synchronization constraints of an application written using one of such languages. In the area of AOP languages, the term can be used to indicate the difficulty of inheriting the aspect code in the presence of inheritance. As an example, it could be useful to define the aspect `RemoteEvaluation` as a "sub-aspect" of `ClientServer`. This is reasonable if you consider that remote evaluation involves passing Christine the recipe, then asking her to prepare the cake (this second part being exactly the same operation done when the client-server approach is taken). Unfortunately, this is not possible and it was necessary to rewrite the aspect code entirely.

All these problems show that AOP is still in its infancy. The experience gained in the area of concurrent object-oriented-languages [7] suggests that these problems might result more from the linguistic choices made in developing AOLs, rather than from intrinsic limitations of the approach. The problem of finding adequate linguistic features which do not suffer from inheritance anomaly is thus an open research topic.

4 Conclusion and Open Issues

AOP tries to provide linguistic mechanisms to factor out different aspects of a program, which can be defined, understood, and evolved separately. It pushes the idea of separation of concerns one step forward with respect to existing programming language constructs, which simply provide ways to encapsulate a single functionality in a unit. Aspects in an AOP resemble ViewPoints in design and specification, as advocated by [11].
AOP, however, is still in its infancy. It is more an open research area than an existing technology that one can use. The problems and pitfalls we outlined in the previous section indicate that it is still unclear which constructs an AOL should provide and how they should interact with the functional language and the mechanisms provided to support functional evolution. As we observed, a fully general-purpose AOL, like AspectJ, with full visibility of the internal details of its associated functional module, violates the principles of protection and encapsulation. On the other end, one might predefine a set of possible aspects an AOL should deal with, and then provide ad-hoc AOLs with constructs supporting limited visibility of certain features of the functional module to which the different aspects apply. The tradeoff is between flexibility and power, on one side, and understandability and ease of change on the other. (For a preliminary discussion of these points, see [12]).
In addition, we feel that aspects should be definable in a formal way. The formal definition will allow the AOL to define an algebra of aspect composition, clearly specifying when certain combinations of aspects are applicable (and what the effect is) or, conversely, when their combination is not possible or not defined, because it generates inconsistencies. Again, the problems arising here are strictly related to the ones being investigated in the case of viewpoints and viewpoint composition.
Research work at the programming language level should go hand-in-hand with experimental work, which should try to assess the usefulness and usability of the language. This is especially important since our claim is that AOP can be a vehicle to support tractability of intrinsic dispersed topics (as distribution of application entities) and this eventually will require some sort of experimental validation. [13] did an interesting initial experiment using AspectJ 0.1. Experiments of similar kind will be needed, as further progress will be made in AOP technology.

References

1. E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.
2. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, (Finland), Springer-Verlag, June 1997.
3. XEROX Palo Alto Research Center, *AspectJ: User's Guide and Primer*, 1998.
4. XEROX Palo Alto Research Center, *AspectJ: User's Guide and Primer*, 1999.

5. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," in *Mobile Object Systems*, vol. 1222 of *Lecture Notes in Computer Science*, pp. 49–64, Springer-Verlag, Berlin, 1997.
6. A. Fuggetta, G. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, 1998.
7. S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages," in *Research Directions in Concurrent Object-Oriented Programming* (G. Agha, P. Wegner, and A. Yonezawa, eds.), pp. 107–150, Cambridge, MA: MIT Press, 1993.
8. A. Yonezawa and M. Tokoro, eds., *Concurrent Object-Oriented Programming*. Cambridge, Mass.: The MIT Press, 1987.
9. G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, pp. 125–141, Sept. 1990.
10. O. Nierstrasz, "Composing active objects," in *Research Directions in Concurrent Object-Oriented Programming* (P. W. G. Agha and A. Yonezawa, eds.), pp. 151–171, MIT Press, 1993.
11. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in systems development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 1, no. 2, pp. 31–58, 1992.
12. G. Kickzales, J. Lamping, C. V. Lopes, A. Mendhekar, and G. Murphy, "Open implementation design guidelines," in *Proceedings of the 19^{th} International Conference on Software Engineering*, (Boston, MA), may 1997.
13. G. Kickzales, E. L. Baniassad, and G. C. Murphy, "An initial assessment of aspect-oriented programming," in *Proceedings of the 21^{st} International Conference on Software Engineering*, (Los Angeles, CA), may 1999.