Jumping Aspects

Johan Brichau*, Wolfgang De Meuter, Kris De Volder
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
Email: {jbrichau,wdmeuter,kdvolder}@vub.ac.be
www: http://prog.vub.ac.be

April 4, 2000

Abstract

In this paper we identify a need for AOP-approaches dealing with "jumping aspect code". This is a cross-cutting phenomenon which occurs because code needs to be added to components depending on their usage-context. As a result it is very hard to identify the specific join points at which aspect code needs to be inserted. The join points seem to be jumping around the code depending on the context in which a component is used.

1 Introduction

The fundamental idea behind AOP [4] is the realization that there is a limit to black box reuse. The more components are built to maximize their ease of reuse, the more their highly generic nature seems to require specific tweaks to use them correctly or efficiently in a particular application context. The important point here is that the kind of specialization is dependent on where and how a component is being used rather than with the components themselves. It seems therefore counterintuitive to attack these by weaving extra aspect code into the component program.

This paper will illustrate this point by means of three examples that show that a single application context may have different uses of a single component that conflict with each other and therefore cannot easily be solved by weaving code into the component program at statically determined join points. The join points themselves can only be determined case by case for each individual usage of the component.

*Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

2 Problems

In this section we describe the three concrete examples where reusability of components is hampered because the component code needs to be augmented with aspect code dependent on the dynamic context in which the components are used. The examples illustrate how the calling context of a method can have an influence on the instantiation of an aspect. In both cases it is hard to insert aspect code into the components because it depends on the concrete situation of how the component is used.

2.1 A First Problem

In this section, the 'Components' are methods and the context dependency that influences the aspect is the call-site from which the method was invoked.

Our example comes from the Smalltalk environment. In Smalltalk applications it is standard practice to use model-view-controller to separate the concern of how things are represented and manipulated internally from how they are visually presented to the user. A mechanism for managing this kind of dependencies, between a model and a view, or potentially a model and another 'sub-model' is built-in to the Smalltalk image. The tricky part however, is that in order for the mechanism to work, the model implementor must insert `self changed` into the model implementation at every location where the state of the model is changed. Clearly this is an aspect-like addition to the code, which cross-cuts the entire implementation of the model.

This looks like a relatively easy aspect, rather straightforwardly solved with a relatively simple weaver. However, the problem becomes more complicated when we realize that not all changes need to trigger an update event immediately. Often we want to defer updates until a number of changes have accumulated, and then allow the update to handle them all at the same time.

Consider the following example of a ListModel, which can handle the addition of elements, either one by one, or a number of them all at once. The component code for these two methods might look like this:

```
ListModel>>add: anElement
        myElements at: currentIndex set: anElement.
        currentIndex := currentIndex + 1

ListModel>>addAll: aCollection
        aCollection do: [ :each| self add:each ]
```

A good AOP-approach should allow us to fully separate out the change-update implementation from the basic functionality. Thus the calls of the `changed` messages themselves should be specified as part of the aspect-code. Weaving should result in the following code:

```
ListModel>>add: anElement
        myElements at: currentIndex set: anElement.
        currentIndex := currentIndex + 1.
        self changed

ListModel>>addAll: aCollection
        aCollection do: [ :each| self add:each ].
```

The problem with this code is that it will give rise to a rather "jitery" user interface and that it is hopelessly inefficient. The reason for this is that the changed method is invoked too frequently. Indeed, each time an `addAll:` method is invoked, this will iteratively invoke an `add:` method, which needlessly invokes the changed mechanism every time. However, removing the `self changed` aspect from the `add:` method to the `addAll:` method does not solve the problem as `add:` can be invoked by any other external client in which case the `self changed` aspect code *does* have to be executed. Hence, it seems as if the very presence of the aspect code has become dependent on the properties of the code from which the component code is used: calling the `add:` method "by itself" requires us to insert the aspect code. Calling it "from within `addAll:`" on the other hand, requires us to omit the aspect code.

Of course, many solutions can be imagined, all of which do not change the original component code. The most obvious one is to pass around a status variable indicating "whether or not the callee is in charge of taking care of handling changes" and to make the execution of the `self changed` code dependent on the status variable.

The problem with this solution is the following: we are manually coding the mechanism that controls the applicability of the aspect code into the aspect code itself. However, we feel that the implementation of this aspect-applicability control mechanism should be left to the weaver. This means that the aspect programmer should only express the need for the presence of such a mechanism in the resulting code. Indeed, perhaps the simplest AOP-approach in the world is a transformational language that can transform any code to any other code. The other end of the spectrum consists of very dedicated aspect languages in which the weaver only generates code which is specific to the problem being addressed. It is of course the entire quest of the AOP-research community to find the golden ratio in between. We feel that all the manual solutions to the above stated problem, i.e. coding the aspect-applicability-conditions in the aspect code, are in the first end of the spectrum, and therefore should be rejected.

The fundamental problem we face in our example is that the aspect code "seems to jump around in the component code depending on the context from which that component code is called" and that current aspect definition languages do not allow us to declaratively specify this knowledge. In other words, the join points seem to dynamically jump around.

2.2 A Second Problem

Consider a spreadsheet-like component that is meant to be incorporated in larger applications, i.e. a suite of cells that is somehow logically interconnected. Such a group of cells is to be thought of as an independent object that is capable of containing text, numbers, formulas and the like. So, it should be thought of as a "bean" that can run both as a stand-alone program and as a component to be integrated into a larger whole. For the development of this bean, we have two simple demands that relate to AOP:

1) One of the issues of the component is the handling of exceptions such as division by zero exceptions. For exception handling, we envision some AOP-approach (say, a dedicated aspect language for the sake of the discussion) that declaratively allows us to express that "whenever X goes wrong in class Y, then Z should be executed in class T". This fictional AOP solution allows us to decouple the functionality of the code from the things that might go wrong and how the resulting conflicts should be resolved. The aspect language should allow us to indicate what can go wrong and where it can go wrong. The weaver is responsible for generating the necessary `throws`-clauses at these points. The aspect language should also allow us to indicate where we want to deal with erroneous situations and what to do with them. The weaver is responsible for generating the necessary `try-catch` statements. In a language like Java, the weaver also bears the responsibility for generating all the necessary `throws`-clauses in the signatures of all intermediate methods. Although, we know of no concrete aspect language implementing this concept, we figure this is not a major research issue, but merely a matter of hard work.

2) Having stated this fictional aspectual definition of exception handling, we can elaborate upon our second "aspect jumping problem", which is the following. The problem relates to the demand that we want our instantiated bean to be usable both as a stand alone application and as a component to be absorbed in other applications. In the first case, errors like "division by zero" have to be caught in the "main class of the component" yielding in the display of a dialog box reporting the error. In the second case we want all errors to propagate further towards the larger application. Hence, in the first case, the `catch`-clause has to be associated to the component. In the second case, it has to be associated to some part of the larger application code. Again, it seems as if the aspect code "jumps around" depending on the context in which the component is used.

Once again, many possible solutions to this problem can be expressed using an aspect language. The most obvious one would be through the use of two different API's, one of which does throw the exceptions and one which does not. This solution can be programmed in the most general aspect language, i.e. a transformational language. But for the same reasons as explained in the previous section, we reject such an approach. The point

is that once again, we want a component (in this case the spreadsheet) to be woven with the aspect at different join points, depending on a dynamic property, being the origin of the call to the component.

2.3 A Third Problem

Another similar problem occurs in the aspect of synchronization. Suppose we have a `Stream` object on which the methods `printChar:` and `printLine:` are defined similarly to the `add:` and `addAll:` method in our first problem, i.e. the `printLine:` is defined in terms of the `printChar:`.

The `Stream` object is used to output information in a distributed application, hence we need to synchronize both the `printChar:` and the `printLine:` methods. This has been illustrated many times as a typical AOP-example. Nevertheless, once again, we observe the property of "jumping aspects". If the `printChar:` method is called, it should invoke the synchronization aspect-code but not if the call originates from the `printLine:` method. The reason for this is that, in the latter case, the synchronization has been taken care of in the `printLine:` method and it shouldn't be re-executed in the `printChar:` method. The problem can be extended when introducing a `Paragraph` class implementing a `printParagraph` method which prints the paragraph's contents onto the `Stream` using the `printLine:` method. Clearly, the printing of the paragraph should not be interleaved with another process' printing and hence it should also be synchronized. In this case, only the `printParagraph` method of the `Paragraph` class should take care of the synchronization aspect and both the `printLine:` and the `printChar:` methods on the `Stream` object should not.

One could argue that this is not really a problem since the required aspect-applicability control mechanism to solve this problem is inherent to synchronization, i.e. the support that a single process can enter multiple synchronized methods. However, the typical AOP-solution to this problem is the weaving of extra synchronization code into the mutually exclusive methods. Trivially, this code is needlessly executed over and over again in case of a typical method calling-chain from `printStack:` on through `printChar:`, in which the latter is typically called much more often. This results in unnecessary extra run-time overhead, which should be avoided. Indeed, it is highly desirable to omit the synchronization aspect code if it has already been executed and once again, this leads to a "jumping aspect".

3 Test Criterion for AOP technologies

Based on our three problems, we experience that the join points of our "jumping aspects" cannot be statically specified. Nevertheless, we *can* think about them in a sufficiently conceptual manner: In the first case it is easy to say that `add:` should not perform a `self changed` when called from within

`addAll:`. In the second case, the components should not catch error when running inside a larger application and in the third case, synchronization does not have to be handled in methods whose callers were already synchronized.

Using current AOP-approaches [5, 3, 1, 2], one can incorporate an aspect-applicability control-mechanism *into the aspect code itself*, hence programming the jumping nature of the join points "by hand". However, manually coding such a mechanism into the aspect code (e.g. the use of flags combined with if-tests) only patches the weak expressiveness of the aspect language. We argue that it is desirable to describe the applicability of the aspect code in the aspect language itself; delegating the actual control-mechanism implementation to the weaver. Perhaps the weaver can generate code that uses flags to control the applicability, but the important point is that this is a problem for the weaver and not for the programmer thinking about jumping aspects. But of course, the quest for a weaver that does not generate abundant code is an important issue as far as we are concerned.

We figure it would be useful to have a more expressive aspect language to guide the weaver in his weaving process, i.e. more information about the structure of the component program should be reified in the aspect language. We do not argue in favor of explicit meta programming, the aspect language should not become a full fledged general purpose programming language but should remain a simple transformational system. From the gedankenexperiments done in the previous section we can easily deduce that the applicability of aspect code not only depends on a location in the component code but also from the locations at which that component code is invoked. So we conclude by stating the following criterion for new AOP technologies, which is also the position we defend in this paper.

New AOP technologies should support jumping aspect code. In order to support jumping aspect code we have to reconsider the idea of a crosscut (or joint point) as an interaction between "locations in the component code" and " locations from which this component code is called".

References

[1] The aspectj language specification. Xerox Corporation, Palo Alto, http://www.aspectj.org.

[2] Siobhan Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of OOPSLA '99*, 1999.

[3] Pascal Fradet and Mario Sudholt. Aop: towards a generic framework using program transformation and analysis. In *ECOOP 98 Workshop Reader*. Springer Verlag, 1998.

[4] G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object Oriented Programming*. Springer Verlag, June 1997.

[5] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. In *Proceedings of the Aspect Oriented Workshop at ECOOP '99*.