Towards a Model-Driven Join Point Model

Walter Cazzola
Dipartimento di Informatica e Comunicazione,
Università degli Studi di Milano
cazzola@dico.unimi.it

Antonio Cicchetti, Alfonso Pierantonio
Dipartimento di Informatica,
Università degli Studi di L'Aquila
{cicchetti,alfonso}@di.univaq.it

ABSTRACT

Aspect–Oriented Programming (AOP) is increasingly being adopted by developers to better modularize object–oriented design by introducing crosscutting concerns. However, due to tight coupling of existing approaches with the implementing code and to the poor expressiveness of the pointcut languages a number of problems became evident. We believe that such problems could be solved shifting the focus of software development from a programming language specific implementation to application design. This work presents a possible solution based on modeling aspects at a higher level of abstraction which are, in turn, transformed to specific targets.

1. INTRODUCTION

AOP [1], is a relatively new paradigm introduced to better deal with the problem of crosscutting concerns, i.e. concepts which are orthogonal to the object definition dimension. In the object–oriented paradigm, the implementation of these concerns affects several classes jeopardizing the functional modularization. Unfortunately, by using it massively makes some issues come to light, as re–usability, scalability and maintainability [2]. These problems are mainly due to the tight coupling of such approaches and the implementation language of the application. Therefore, we believe that a solution could be found by adopting techniques based on Model–Driven Architecture (MDA) world, to leverage the abstraction for a language independent solution.

This paper proposes an approach to coordinate pointcuts, advices and join points by means of appropriate models. In particular, models are used to specify the weaving among pointcuts and advices from both structural and behavioral perspectives, pursuing the following AOP enhancements: *i*) a better pointcut definition by using a "semantical" description; *ii*) an advice weaving improvement enabling more complex merge operations; *iii*) a higher possibility of concerns specification re–use.

SAC'06 April 23-27, 2006, Dijon, France

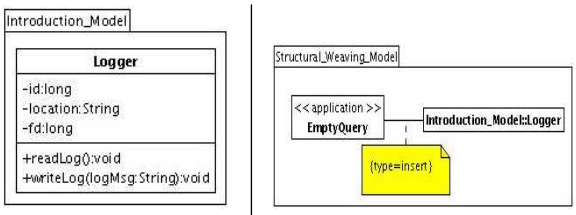
2. RAISING ABSTRACTION

Shifting the focus of software development from a programming language specific implementation to application design, using appropriate representations by means of models is likely the major characteristic of MDA [3]. Accordingly to such vision, modeling techniques can be used to select in a more semantics–based fashion join points and weave code into those points (i.e., a Join Point Model). Pointcuts can be seen as queries to the code, used to weave advices into the original code at the selected join points. A pointcut deals with patterns that could be found in the computational flow of the program, these patterns identify a set of join points. Every time a certain pattern is recognized, the related join point is picked up to apply the correspondent advice. In the proposed approach, models are defined for pointcuts, join points, and advices, respectively; how those different features interact with the initial application, both from the structural (i.e. packages, classes) and the behavioral point of view (i.e. the computational flow), must be provided.

Concerning the structural view, we have the application and the advice models, that we call *introduction model*. Moreover, we need to specify how the previous models will be merged in a single model; in fact, it is often necessary to refactor the application model (base) with the new introduced elements (extension), and not just compose them straight away. Hence, we describe this operation in a detailed way by using the *structural weaving model*; this model relates base and extension by introducing more abstract mappings among the elements. In particular, we need to express three distinct cases: *i*) an extension element is inserted as stand alone in the base model; *ii*) an extension element is linked to some base elements; *iii*) an extension element is used to modify some base elements. In the first case, we use as base element a class with a specific stereotype, namely ■`application`■, which is used to identify the whole base model, and as extension what we would like to insert from the *introduction model*; these elements will be linked with a bidirectional association. In the second case, base and extension will be linked by the desired kind of association (generalization, composition and so on). Finally, the elements will be linked by a bidirectional association, decorated with two tagged values, namely `type` and `scope` which are used as a refinement. The former can be `insert` or `merge`, meaning that the extension will overwrite the base or it will update that respectively. The latter tag is used to narrow the manipulation to the set of base: `attributes`, `methods`, or both. If the just mentioned tagged values are not specified, we assume by default `type=insert` and as `scope` the class as a whole.

Once described the structural cooperation, we specify the behavioral one considering pointcuts as patterns describing computational flows and using activity diagrams for modeling them. Every action name describes the semantic of a certain operation, hence a computational flow is represented by a sequence of actions. Between those actions we identify the join point we want to capture by using the stereotype «joinpoint». We can either add new operations or substitute a set of them. Finally, when describing computational flows, it could be useful (or necessary) to specify a sort of wildcard, «anyflow», meaning that we are not interested in matching a particular computational flow, rather any flow of the base can match that part of the pointcut. In this case the advice model is described by computational flows and it can contain context information passed during the weaving phase by means of an object. As in the structural case, we need to describe the weaving in a third model, namely *behavioral weaving model*; in this model we specify the links between the join points selected by the pointcuts and the advices by means of directed associations from a pointcut to an advice. This link can be refined by using a tagged value, `parameter`, which contains a list of arguments to be passed to the advice.

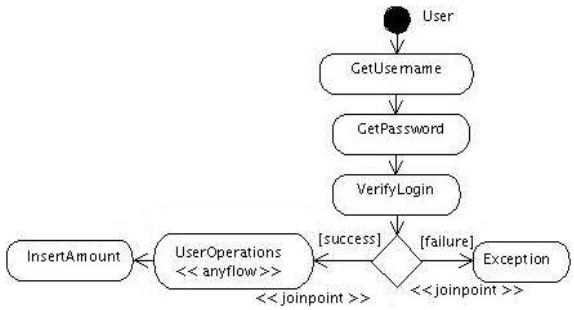
To better explain the approach, we show the classical *logging* example realized by using our model–oriented approach. Hence, we consider an ATM application to extend with the logging non–functional concern. In Fig. 1.a there is the *introduction model* that contains the logger class, i.e. the class that will be added to the application model while the *structural weaving model* (Fig. 1.b) contains the links application elements to *introduction model* ones. In the figure we



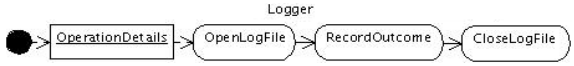
a) Introduction Model b) Structural Weaving Model

Figure 1: The Logger structural models

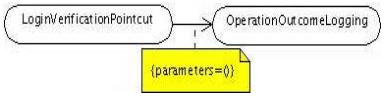
can see that the `Logger` class has to be inserted as a stand–alone class (we have used the stereotype «application»). To weave new behaviors to the application, we specify the operation flows to be modified (pointcuts) and the new flows to be added (advices). In Fig. 2.a, we can see the flow of a general login operation; so we read a userid, a password and then we check the correctness of the introduced data. The log behavior is added to the two join points identified by the «joinpoint» stereotype on the links exiting from the operation outcome diamond. The use of the «anyflow» stereotype permits to exclusively select, in case of success, and then log those computations that include withdrawal operations (`InsertAmount`). As depicted in Fig. 2.b the advice receives context information (represented in this case by the `OperationDetails` object), and describes the new operations. Finally, in Fig. 2.c there is the description of the necessary links; thus, we relate the `LoginVerificationOutcome` pointcut to the `OperationOutcomeLogging` advice. Any join point can be captured by several different pointcuts but there is



a) Pointcut Model (LoginVerificationOutcome)



b) Advice Model (OperationOutcomeLogging)



c) Behavioral Weaving Model

Figure 2: The Logger behavioral models.

no direct association with the advices and we need just a weaving rule to connect them.

3. CONCLUSIONS AND FUTURE WORK

We believe that the key points of this proposal are: *i*) a more semantics–related pointcuts definition. In fact, when we describe operational flows, we are identifying the precise meaning of what we would like to weave with advices, and not the name of the method that performs it; *ii*) complex weaving definitions, such as making a join point dependent on the operations that follows it, included other join points; *iii*) independence from name conventions.

However, transforming the model toward a specific target could be not so simple and obvious; in fact, if on one hand the level of abstraction enables complex definitions, on the other hand it makes the transformations difficult to specify.

4. REFERENCES

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.

[2] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, September 2004.

[3] Bram Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September 2003.