On the Problems of the JPMs

Walter Cazzola[1], Antonio Cicchetti[2], and Alfonso Pierantonio[2]

[1] Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano,
`cazzola@dico.unimi.it`
[2] Dipartimento di Informatica
Università degli Studi di L'Aquila,
{`cicchetti|alfonso`}`@di.univaq.it`

1 Introduction

Increasingly, aspect-oriented programming (AOP) [1] has been considered to pursue a better modularization of object-oriented programs, especially when representing crosscutting concerns. However, current aspect-oriented frameworks such as AspectJ [2], AspectWerkz [3], JMangler [4] and others, operate in a way directly related to the code. They define some points in the computational flow, named *join points*, loci of the crosscutting concerns and provide a pointcut definition language that allows the definition of some predicates on those join points to detect where and when a decoupled crosscutting concern has to be woven again.

Unfortunately, these mechanisms are too tightly connected to the program structure and syntax, in fact join points consider only simple cases, such as method calls, and the pointcut languages are based on a linguistic pattern matching, respectively. Therefore, when using it massively, a number of problems raise due to the strict coupling between the pointcut definitions and the implementation:

– it could be hard to write pointcuts which pick out all and only desired join points;
– it's difficult to express modifications spread all around the code [5];
– when changes occur on the code, some pointcut definitions could be invalidated.

Consequently, instead of improving modularization, actual AOP approaches tend to make systems difficult to scale, maintain and reuse [6] and prevent developers from fully taking advantage of the AOP benefits which suggest to look for a more expressive join point model (JPM).

2 AOP meets Model-Driven Development

Model-driven development (MDD) [7] techniques were devised to shift the focus of software development from coding to modeling in order to abstract system design from platform-specific issues. Thus, the models are the primary artifacts and considered as first-class objects which allow the designer to better understand complex problems and their solutions. The ultimate goal of such a vision is to protect investments in business logic by generating platform-specific models and code for different implementation environments in a (semi) automatic way.

A model is an abstract description of some reality, which aims to point out interesting features for a certain domain; to make such a description they are used well defined syntax and semantic, which usually are specified in terms of the model itself, i.e. the meta-model. Leveraging the abstraction in JPMs is challenging [5, 6, 8], since current proposals suffer from being too syntax-oriented which appears as a limitation to separate non functional concerns tangled with the functional ones within aspects. Investigating on problems related to JPMs, you can rapidly agree that the level of abstraction has to be improved to separate in aspects non functional concerns tangled with the functional ones; thus, following MDD paradigm, we think that it is necessary a JPM defined at meta-model and model level. Pointcuts will become queries to select a particular sub-model of the model itself; the specification level (meta-model or model) will depend on the specificness of the definition: more it needs to be accurated, and more the level of abstraction has to decrease. An advice will be the sub-model used to substitute the one selected by a pointcut; this way join points will completely disappear or, to say better, they will become implicit (i.e. a join point is every possible starting point from which a sub-model can be changed with another).

A JPM at such a high level of abstraction has several advantages:

– definitions semantic-related, not syntax-related;
– scalability;
– reusability;
– maintainability.

First of all the programmer can represent pointcuts in a semantic way, i.e. she/he is completely independent of the program syntax; moreover, the developer can specify complex pointcuts which could require changes spread on the code. This main property enables all the following; scalability, reusability and maintainability directly derive from the breach of relations between base code and JPM. In fact, when you write a query at a model level, you are really partitioning the model in sets of semantically related elements. Thus, when you are going to add elements for example, they automatically make part of a certain set; if you change names of elements which aren't involved in pointcuts definition, you won't need any adjustment.

However, there are some drawbacks:

– application to code;
– re-engineering of actual aspect-oriented code.

Defining aspects at such a high level requires new solutions to efficiently map modeled pointcuts to target code: we could choose to transform directly the model to a “plain” code, i.e. a base code where aspects totally disappear; we could transform the portions that could be implemented in a AOP language and refactor the base code for the remaining; we could build an engine being able to weave aspects and code at runtime. They are all solutions we are actually investigating on. Furthermore, we have to consider that all the code already written with current AOP languages should be re-engineered, thus allowing on one hand the maintenance and reuse, and on the other hand a “soft” introduction of this new kind of JPMs.

3 Conclusions

Using AOP in non trivial projects, raises several problems related to the poorness of JPMs, that in turn is due to the fact that they are defined at a too low level of abstraction. Our opinion is that designing JPMs at a higher level could solve several issues; as MDD paradigm suggest, this level should be the model. We are actually investigating on a good way to query the model (i.e. the definition of JPM) and, as said before, on how to apply modeled weavings at code levels.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. Springer **1241** (1997) 220–242
2. Laddad, R.: AspectJ in Action. Manning Publications Co., Grennwich, Conn. (2003)
3. Vasseur, A.: Dynamic AOP and runtime weaving for Java—How does AspectWerkz address it? In Filman, R., Haupt, M., Mehner, K., Mezini, M., eds.: DAW: Dynamic Aspects Workshop. (2004) 135–145
4. Kniesel, G., Costanza, P., Austermann, M.: JMangler—A powerful back-end for aspect-oriented programming. In Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005) 311–342
5. Cazzola, W., Pini, S., Ancona, M.: AOP for Software Evolution: A Design Oriented Approach. In: Proceedings of the ACM Symposium on Applied Computing 2005, ACM press (2005) 1346–1350
6. Tourwé, T., Brichau, J., Gybels, K.: On the existence of the AOSD-evolution paradox. In Bergmans, L., Brichau, J., Tarr, P., Ernst, E., eds.: SPLAT: Software engineering Properties of Languages for Aspect Technologies. (2003)
7. Selic, B.: The pragmatics of model-driven development. IEEE Software **20** (2003) 19–25
8. Tourwé, T., Kellens, A., Vanderperren, W., Vannieuwenhuyse, F.: Inductively generated pointcuts to support refactoring to aspects. In: Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT’04), Lancaster, UK (2004)