Ad-hoc Constructs for Non Functional Aspects

Mattia Monga

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

June 2000

1 Malaj: A Multi Aspect LAnguage for Java

Aspect-oriented programming [1] (AOP) tries to provide linguistic mechanisms to factor out design concerns, which can be defined, understood, and evolved separately from the functional core of the application. A first approach to AOP can be the definition of a general-purpose Aspect-Oriented Language (AOL) with full visibility of the internal details of its associated functional module. In our previous work [2, 3] we analysed AspectJ [4] – an AOP following this approach – and came to the conclusion that it violates object-oriented principles of protection and encapsulation, thus hindering reuse. Another approach, followed by Hyper/J [5], is based on the idea of partitioning a software system in small atomic units, that can be aggregated according to different dimensions of concern. We found that the number of connections among units increases the overall intricacy of the system, again hindering comprehension and evolution.

We claim that such limitations are intrinsic to all general solutions to the problem and we suggest to predefine the set of aspects an AOL should deal with, providing ad-hoc AOLs (one for each aspect) with constructs supporting limited visibility of certain features of the functional module to which the different aspects apply. In our Malaj (Multi Aspect LAnguage for Java) we define an AOL specific for synchronisation and one specific for distribution and mobility.

1.1 The Synchronisation Aspect

Functional units should be programmed without focusing on their synchronisation: a second step could clearly state what happens when a functional unit is invoked. Three cases may arise:

1. the call violates some precondition and an exception is returned to the

caller (these conditions are named *deny guards*);

2. the call violates some precondition and the caller is suspended until the condition becomes true (*suspend guards*);

3. the call does not violate any precondition and execution of the functional unit may proceed.

Malaj provides the **guardian** construct for expressing synchronisation constraints of methods of a class. Each guardian **guards** a class and each class has at most one guardian.

For each class `C`, the guardian `G` of `C` basically represents the set of **synchronized** methods of `C`.

A guardian may include also a set of local attributes and method definitions to code guards that depend on state conditions. Finally, for each method `m` of the guarded class, the guardian may introduce a fragment of code to be executed **before** or **after** `m`. Observe that, to avoid breaking object encapsulation and to increase separation between the functional and synchronisation aspects, guardian code (i.e., **deny** and **suspend** guards, and **before** and **after** clauses) cannot access private elements of the guarded class and has read-only access to the public and protected attributes of the guarded class.

1.2 The Relocation Aspect

Network aware programs have often the need to relocate functional units among sites. Relocation can be expressed independently specifying objects deployment and relationships to be maintained during objects motion:

Ownership: if an object A owns an object B, then A is the only object entitled to move B. By default, B follows A in its movements.

Interest: if an object A is interested in B, A has to be always able to reach B, but A and B move completely independently.

If an object A does not own B and is not interested in it, it simply does not care of B's location, and even of its existence. Evidently, ownership implies interest.

These relationships are inherently dynamic: they are subject to change during program execution, as objects change their interest in other objects according to the programmers' needs.

Malaj provides the **relocator** construct: it **relocates** the objects of a class. Relocation actions can be executed before or after the execution of any method. To specify this, the relocator provides **before** and **after** clauses that allow programmers to introduce the piece of code that will be executed before or after the execution of the method.

In **before** and **after** clauses one is not allowed to change attributes (i.e., the internal state of an object can be changed only by using the methods it provides). However, it is possible to:

- Take or release the ownership of an object, by using the methods:

  ```
  takeOwnership(Object owned)
    throws ObjectOwnedException

  releaseOwnership(Object owned)
    throws NotOwnerException
  ```

  Only the owner is allowed to release ownership and only objects that have no owner can be arguments of `takeOwnership`. Observe that, by default, each newly created object is owned by the object that created it.

- Express or retract the interest in an object, by using the methods:
  ```
  expressInterest(Object o)

  retractInterest(Object o)
  ```
- Fix the location of an owned object, by using the methods:
  ```
  pin(Site s, Object owned)
    throws NotOwnerException

  unpin(Object owned)
    throws NotOwnerException
  ```
  Unpinned objects reside in the same site of their owner.
- Refer to variable and method definitions that are local to the relocator.

2 Conclusions

Encapsulation is the kernel of object-orientation and each hole we make in its boundaries has to be carefully designed, because can destroy the whole framework. Design criteria behind Malaj [6] were inspired by earlier experience with general-purpose aspect oriented languages. We think our approach offers a good compromise between flexibility and power, on the one side, and understandability and ease of change on the other. It does not allow programmers to code any possible concern, but it enables the comprehension of concern specific relations with functional code. This would be impossible in general. We envision Malaj as a collection of concern-specific aspect languages, built on top of a subset of the Java language. For now we discussed how the synchronisation and relocation aspects can be defined in Malaj. But one ultimate goal is to cover a spectrum of concerns far beyond these two, and to complement the programming support with a formal model that can be used to reason about program construction and concerns interaction.

References

[1] 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.

[2] G. Cugola, C. Ghezzi, and M. Monga, "Language support for evolvable software: An initial assessment of aspect-oriented programming," in *Proceedings of International Workshop on the Principles of Software Evolution*, (Fukuoka, Japan), July 1999.

[3] M. Monga, "Concern specific aspect-oriented programming with malaj," in *Proceedings of Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, (Limerick, Ireland), June 2000.

[4] XEROX Palo Alto Research Center, *AspectJ: User's Guide and Primer*, 1998.

[5] P. Tarr and H. Ossher, *Hyper/J^{TM} User and Installation Manual.* IBM Research, 2000.

[6] G. Cugola, C. Ghezzi, M. Monga, and G. P. Picco, "Malaj: A proposal to eliminate clashes between aspect-oriented and object-oriented programming." Accepted for publication in Proceedings of the 16th IFIP World Computer Congress International Conference on Software: Theory and Practice(ICS2000), Aug. 2000.