Concern Specific Aspect-Oriented Programming with Malaj

Mattia Monga

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

1 Motivations

Software engineers know how *separation of concerns,* i.e., thinking about a problem at a time, can greatly improve designers' productivity. The success of object-oriented techniques is tightly coupled to their ability to support the decomposition of complex systems into simpler sub-systems, designed and implemented as much as possible independently. Objects isolate concerns by encapsulating design concepts behind a well-defined interface and by grouping together related functions.

However, some concerns refuse to package themselves in a component, because their influence *cross-cuts* the entire system. Example of these are concurrency, distribution, persistence, security, etc. Research work is on-going to find ways to overcome these limitations of object-orientation without giving up its proven advantages.

A first approach (see Section 2 and Figure 1.a), followed by AspectJ [1], is based on the idea of weakening information hiding rules, so that unencapsulable *aspects* could be woven into encapsulated functional code. The main problem with this approach is that aspects may interfere with functional code or one with another.

Another approach (see Section 3 and Figure 1.b), followed by Hyper/J [2], is based on the idea of partitioning a software system in atomic *units*, that can be aggregated according to different *dimensions*. The problem here is the number of connections among units, which increases the overall intricacy of the system, thus hindering comprehension and evolution.

We believe that a third approach (adopted in Malaj [3], see Section 4 and Figure 1.c) can be more convenient: a number of unencapsulable concerns are to be identified and carefully studied to understand their very relationship with functional code, so that ad-hoc linguistic constructs could be properly designed. This way, we aim at reducing clashes with traditional linguistic features and best practices of object-orientation.

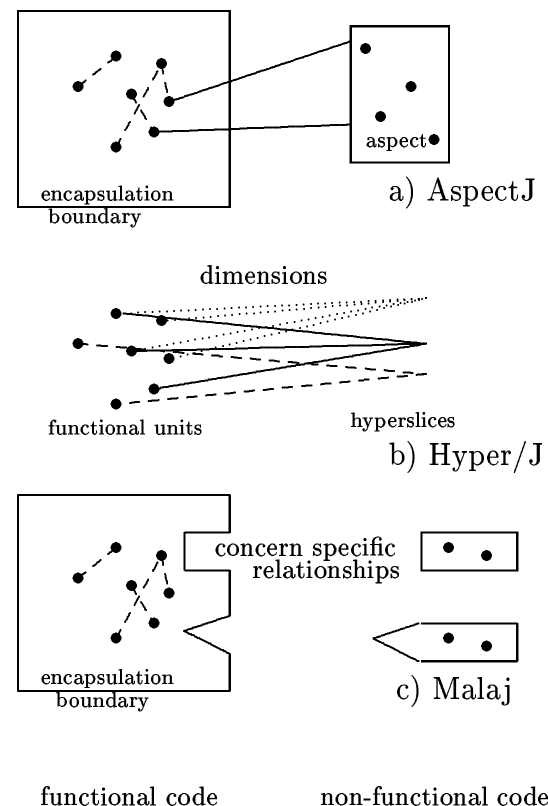

Figure 1: Different approaches to separation of unencapsulable concerns.

2 Aspect-Oriented Programming with AspectJ

AspectJ [1] is an aspect-oriented language (AOL) proposed to make aspects clearly identifiable from functional code, which is written by using Java, called component language (CL). AspectJ pro-

vides:

1. some syntactic sugar to isolate the code for each aspect;
2. a way to identify the *join points* in functional code, i.e., the points where the aspect code is introduced;
3. a *weaver*: an engine that is able to mix aspects and functional code.

In AspectJ an aspect is defined via a construct that is reminiscent of the Java **class**. An **aspect**, has a name and its own data members and methods. With an **aspect** it is possible to **introduce** an attribute or a method in an existing **class** and **advise** that some actions are to be taken **before** or **after** the execution of an existing method.

Classes are unaware of aspects, i.e., it is not possible to name an aspect inside a class. The association between aspect instances and class instances is one-to-one. However, by using the keyword **static**, it is possible to define an association between an aspect instance and all the objects of a class. An aspect can access all the internal details of its associated object (see Figure 1.a).

In a previous analysis [4] we identified a number of problems and pitfalls that currently affect AspectJ[1]. In particular we found three main clashes between the aspect oriented and the object oriented features of the language. Possible clashes occur 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 particular aspect. In AspectJ, the aspect code can access the private attributes of a class. This may be useful in some situations, but results in a potentially dangerous breaking of class encapsulation.
- *Different aspects.* Two aspects could work perfectly when applied individually, but fail when applied together (e.g., because they introduce the same method with different definitions).
- *Aspect code and specific language mechanisms.* One of the best known examples of problems that falls into this category is inheritance anomaly [5]. This term was first used in the area of concurrent object-oriented languages to indicate the difficulty of inheriting the code used to implement the synchronisation constraints of an application written using one of such languages. In the area of aspect-oriented languages, the term can be used to indicate the difficulty of inheriting the aspect code by a subclass.

[1] We used version 0.3.0.

3 Multi-Dimensional Separation of Concern with Hyper/J

Hyper/J [2] is a language to define multiple dimensions of concern within a software system written in Java. Hyper/J considers the system as a set of Java declarations (methods, attributes and classes) and provides:

1. a notation to map declaration units to arbitrary *concerns* and concerns to concern dimensions. Each can pertain to many concerns (see Figure 1.b);
2. a notation to describe various correspondences between declarations and definitions;
3. an engine to bind together declarations and definitions according to such correspondences.

The **hyperspace** composed by all declaration units is partitioned in **hyperslices** containing all concerns pertaining to a dimension plus any other (abstract) declarations needed to achieve *declarative completeness* (an hyperslice is declaratively complete if contains all definitions needed for static checking). Hyperslices are eventually integrated in executable **hypermodule**s specifying which definitions must be linked to abstract declarations. For example, two method declarations can have the same definition because they have the same *name* (i.e., signature) and the actual definition could be made by *merging* (i.e. concatenating) their original definitions. Possible alternatives to merging are overriding, bracketing (some code is executed **before** and **after**), summarising (results of each original definition are aggregating in arbitrary ways). According to the Hyper/J approach,

classes are simply concerns pertaining to an object-orientation dimension, and inheritance becomes a relationship between a base and a derived concern in which the definition of derived method overrides the base one.

In this elegant approach we identify some problems:

- *Software creation.* Dimensions are partitions of hyperspace composed by Java units: therefore, the declaration of these units conceptually precedes the separation of concerns. Designers of the system receive no help from hyperspaces: they have to generate a bunch of classes to cope with all concerns. For example, it is not possible to assign the developing of a dimension to a separated team.
- *Software evolution.* Hyper/J can be very helpful to re-engineer an existing application, by identifying the different dimensions involved in the software. Evolution seems easier, but what about evolution of the evolved software? If designers do not want to go back to square one, they need a "hyper-Hyper/J" to evolve hypermodules.
- *Software complexity.* The overall intricacy of the system does not diminish introducing different dimensions. There is no conceptual economy in the definition of dimensions of concern. Relations among software units become explicit, but their number do not decrease.

4 Malaj: a Multi Aspects LAnguage for Java

A general-purpose approach provides the maximum expressive power but forces programmers to violate the principles of protection and encapsulation. For this reason, Malaj [3], focuses on a well-defined set of aspects (currently, synchronisation and relocation), and provides a different linguistic construct for each aspect. This limits visibility of the features of the functional module to which the different aspects apply. This is made possible by carefully studying the relationship between functional code and a *given* aspect (see Figure 1.c).

As its name says, Malaj is an aspect oriented extension to Java. The Malaj core language is a reduced version of Java, that does not include the features that are provided through the separately specified aspects. More specifically, the Java keyword **synchronized** cannot be used in Malaj, and the same is true for the methods `wait`, `notify`, and `notifyAll` of the standard Java class `Object`. We want to show how Malaj can provide support to synchronisation and relocation of objects, without tangling these concerns in functional code.

The Synchronization Aspect

Synchronisation between the different units that compose an application is a central aspect for any, non-trivial, software. To express this aspect it is necessary to 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.

To support this aspect, Malaj provides the **guardian** construct. Each guardian is a distinct source unit with its own name, possibly coded in a different source file. Each guardian is associated with a particular class (i.e., it **guards** that class) and expresses the synchronisation constraints of a set of related methods of that class. 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.

As for the relationship between the synchronisation aspect and inheritance, the following rules exist:

1. The guardian of a class `C` is inherited by all the subclasses of `C` that do not have a different guardian.

2. A guardian `G1` always extends a parent guardian `G`. If not explicitly mentioned, the parent guardian of `G1` is the guardian `malaj.Guardian`, which is part of the Malaj package. `G1` inherits all the synchronisation constraints specified by `G` and it may add new guards, redefine existing ones, or remove them. To distinguish between added and redefined guards, each guard of a given method `m` has its own label. A guard in `G1` that has the same label of a guard in `G` redefines it, otherwise it is considered as a new guard.

3. The guardian of a class `C` must extend the guardian of the parent class of `C`.

4. The guards redefined in `G1` cannot be stricter than the original ones. In fact, as the next point explains, a sub-guardian `G1` guards a class that extends the class guarded by the parent guardian of `G1` and, as observed by Meyer [6], the precondition of a subclass cannot be stronger than the precondition of the parent class.

5. To reduce the impact of inheritance anomaly, the **before** and **after** clauses of a guardian `G` may refer to the corresponding clauses of the parent guardian through the statement **super**(). Similarly, in redefining a guard it is possible to refer to the original guard through the construct **super**().

The Relocation Aspect

Today software has to be aware of networks and code implementing network awareness is typically dispersed among functional units, thus representing a good candidate to be *aspectified*. In particular, programmers should be able to move objects among sites. We identify two relationships to be maintained as objects move:

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. Each relocator is a distinct source unit with its own name, possibly coded in a different source file. A relocator is associated with a particular class (i.e., it **relocates** the objects of that 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.

As for the relationship between the distribution aspect and inheritance, the following rules exist:

1. The relocator of a class `C` is inherited by all the subclasses of `C` that do not have a different relocator.
2. A sub-relocator `L1` may add **before** and **after** clauses for methods not considered in the parent relocator `L` and may redefine `L` clauses.
3. To reduce the impact of inheritance anomaly, the **before** and **after** clauses of a relocator `L` may refer to the corresponding clauses of the parent relocator through the statement **super**().

5 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 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] XEROX Palo Alto Research Center, *AspectJ: User's Guide and Primer*, 1999.

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

[3] 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.

[4] 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.

[5] 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.

[6] B. Meyer, *Object-oriented Software Construction.* New York, NY: Prentice Hall, second ed., 1997.