Aspect-Oriented Programming Using Reflection

Noury Bouraqadi
bouraqadi@ensm-douai.fr
http://csl.ensm-douai.fr/noury
Dépt. G.I.P
Ecole des Mines de Douai
941 rue Charles Bourseul - B.P. 838
59508 Douai Cedex - France

Thomas Ledoux
ledoux@emn.fr
http://www.emn.fr/ledoux
Dépt. Informatique
Ecole des Mines de Nantes
4, rue Alfred Kastler - BP 20722
44307 Nantes Cedex 3-France

Technical Report 2002-10-3
Ecole des Mines de Douai

Wednesday October 30th 2002

Contents

1 Introduction

Aspect-Oriented Programming (AOP) allows a given application to be developed by implementing crosscutting concerns (i.e. aspects) in a loosely coupled fashion [KLM+97]. The application results from a weaving process, which stands for knitting aspects together in some points named join-points. Thus, how to separate aspects and how to weave them are two critical issues of the AOP paradigm. In order to deal with these issues, one possible approach is to use reflection.

Reflection is the ability of a system to observe and change its own execution. In an object-oriented reflective programming language, this ability is made possible by representing program entities (e.g. classes, methods) and execution mechanisms (e.g. interpreter, garbage-collector) as full fledged objects named meta-objects. Meta-Objects Protocols (MOP) offer the possibility to extend the programming language and adapt the execution mechanisms. Thus, using a reflective language one can implement both applications functionalities using objects and non-functional concerns (i.e. how functionalities will be performed) using meta-objects. While implemented in isolation, functionalities and non-functional concerns are still linked since objects and meta-objects are linked. Therefore, reflection introduces a natural way to separate and to compose functionalities with non-functional concerns.

This chapter is dedicated to the exploration of relationships between reflection and AOP. We show how to support AOP by taking benefit from the natural separation and composition of concerns provided by reflection. In this context, meta-objects are used for building aspects while the MOP supports joint-points.

The chapter is organized as following. First, we present reflection and the underlying concepts. Then, we show the capability of reflection to support AOP and illustrate it with several examples. Next, we discuss the advantages and drawbacks of using reflection for AOP. The chapter ends by a conclusion listing open issues.

2 What is Reflection?

Reflection is the ability of a system to observe and change its own execution [Smi84, Mae87]. A programming language is said to be reflective if it provides an explicit representation (i.e. *reification*) of entities representing either program building blocks (e.g. classes, methods) or involved into program execution (e.g. stack, garbage collector). Thus, using a reflective language, developers can not only define system's (i.e. software) functionalities, but they can also define new program building blocks or execution mechanisms, i.e. how functionalities will be performed. Put another way, developers can not only define the program, but they can also extend the interpreter.

2.1 Base-level vs. Meta-level

Separation between functionalities and execution mechanisms lead to a system with different levels (i.e. layers[1]). On the one hand, we have the *base-level* where are defined system functionalities (e.g. deposits and withdrawals in a banking system). And, on the other hand we have the *meta-level* where are located reified entities such as system's building blocks (e.g. fields, methods) and execution mechanisms (e.g. message handling, process scheduling). So, the meta-level can be viewed as an interpreter that evaluates the base-level.

As shown in figure 1, a reflective system can have more that two levels. The reason is that the meta-level is part of the system. And, since a reflective system can reason and act up on it self, then the system can reason and act up on the meta-level. So, the meta-level can also be viewed as program which is interpreted by some meta-meta-level. The meta-meta-level is also part of the system so we need an extra level, and so on.

Although the tower of levels can be arbitrary high, the number of levels should always be finite. Indeed, the system should be able to perform its tasks within a finite amount of time and memory. This is why reflective systems rely on some default interpreter which acts as the top most level

[1] In the context of reflective systems people prefer the term "level" instead of "layer".

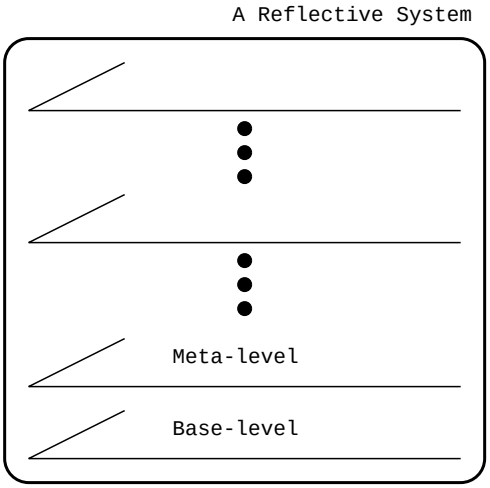


Figure 1: Levels of a Reflective System

[Mae87]. The evaluation of this root level is hardwired so as to stop the infinite regression of towers.

Note that the relationship between the base-level and the meta-level can be translated along the reflective tower. So, any level is described and controlled by the level above. This is why, we can simplify the remaining of this section by focus only on the two first levels (base-level and meta-level).

2.2 Meta-objects and Their Protocols

When developing using an object-oriented reflective language, built systems and hence levels are compound of objects. Classically, objects that define program functionalities are called *base-level objects* (base-objects for short) since objects defining program building blocks or execution mechanisms are called *meta-level objects* (meta-objects for short).

Protocols to manipulate meta-objects are called *meta-object protocols (MOPs)* [KdRB91]. MOPs allow to access to the program structure (e.g. class and inheritance relationships, methods and field defined within some class). They also give access to the execution mechanisms. Examples of execution mechanisms are object creation, message sends and receptions, method lookup and evaluation, and fields reads and writes. Since meta-objects are objects, they are instances of some classes that define fields and methods to properly handle these execution mechanisms. So, using inheritance, one can define new kinds of meta-objects that extend the execution mechanisms. One can even define meta-objects that with completely different semantics (e.g. single vs. multiple inheritance).

Abstraction levels of a reflective system introduce a new kind of relationship between objects: base-objects and meta-objects are connected through a link named *meta link*. Functionalities provided by each base-object are then executed using the execution mechanisms defined in meta-objects the base-object is linked to. We say that such meta-objects *control* base-objects. As an example, a meta-object can extend the default message reception mechanism in order to produce a log. Therefore, we can log messages received by any object by simply linking it to our "log meta-object".

In the above example, we linked a single base-object to one meta-object. However, other alternatives exist [Mae87, WY88, Yok92, McA95, GK99]. Shortly, according to the system's architecture and application requirements, a single base-object can be controlled by one or more meta-objects. Also, meta-objects can be shared or not among different base-objects. When many meta-objects control a same base-object, they have to *cooperate* somehow in order to handle the

execution. Usually, the cooperation policy is not hardwired. So, it can be extended and adapted likewise most of system's features.

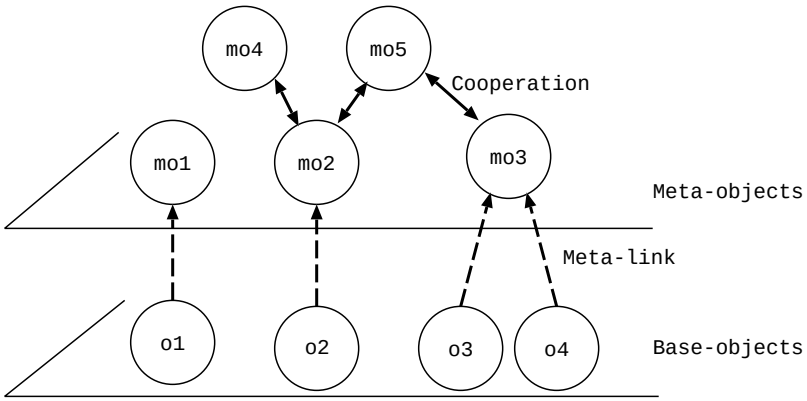


Figure 2: Base vs. Meta-objects and Their Relationships

Figure 2 shows four objects (o1 to o4) which are linked to different meta-objects (mo1 to mo5) using a meta-link of cardinality N-1. The mo1 meta-object controls the o1 object. Three meta-objects (mo2, mo4 and mo5) cooperate in order to control the o2 object. And, meta-objects mo3 and mo5 cooperate in order to control objects o3 and o4. So, these latter objects (o3 and o4) share meta-objects mo3 and mo5. The latter meta-object (mo5) is also shared with the o2 object.

2.3 Example of a Reflective Programming Language and its MOP

Our example is a running reflective programming language named MetaclassTalk [BS99, Bou02]. MetaclassTalk is a reflective extension of Smalltalk that aims easing experiments with new programming paradigms. Smalltalk has been chosen to implement MetaclassTalk because it provides many reflective facilities [Riv96]. As an example, classes and methods are reified and can be handled as plain objects. As objects they are instances of some classes that are available to developers (for extension, adaptation or simply browsing).

Although Smalltalk reflective facilities are numerous, they provide little help for changing the execution mechanisms. MetaclassTalk addresses this weakness by opening-up the execution process. As an example, MetaclassTalk allows changing the object creation process, the inheritance policy, and method evaluation, etc.

2.3.1 The MetaclassTalk MOP

MetaclassTalk allows controlling most execution mechanisms of a reflective OO language. Indeed, among methods defined by the root class for meta-objects we find:

- **send:** *methodName* **from:** *sender* **to:** *receiver* **arguments:** *argArray* **superSend:** *superFlag* **originClass:** *originClass*
  Controls message sends (i.e. outgoing messages). Its arguments are the following:

  methodName the name of the method to invoke[2]

  sender the object who emits the message.

  receiver the object who should receive the message.

  argArray an array with message parameters.

  superFlag is true if the message is sent to super.

[2]Remember Smalltlak is dynamically typed. So, the name of a method is also its signature.

originClass the class where the message is emitted. This is useful for super sends since it make it possible to find out the starting class for look up.

- **receive:** *methodName* **from:** *sender* **to:** *receiver* **arguments:** *argArray* **superSend:** *superFlag* **originClass:** *originClass*
  Controls message receptions (i.e. incoming messages). Takes the same arguments as the previous method. The only difference is that it is performed by the meta-object of the receiver of the message.

- **atIV:** *fieldIndex* **of:** *anObject*
  Controls fields[3] read accesses. The object structure is based on an array where each element represents a field. So, this method takes two arguments :

  fieldIndex the index of the field to access.

  anObject the object which holds the field.

- **atIV:** *fieldIndex* **of:** *anObject* **put:** *value*
  Controls fields write accesses. The last parameter denotes the value to store.

As said above, this list provides only part of the MetaclassTalk actual MOP. Other reflective facilities are also available (e.g. memory allocation for created objects, method lookup and evaluation).

2.3.2 Meta-link and Meta-Object Cooperation

Because of its openness, MetaclassTalk makes it possible to implement a variety of relationships between base-objects and meta-objects. By the time of writing three policies are implemented:

- A single meta-object shared between instances of a same class.
- A single specific meta-object private to each object.
- Many meta-objects shared among many objects.

In the following, we'll use the latter policy where many meta-objects cooperate to control one or many base-objects. Then, we need a cooperation rule. A simple one consists of stacking meta-objects. That is, the meta-object which is the last to be linked to a given base-object first takes control. So, this meta-object can perform some meta-processing. Then, it can (or not!) give the next meta-object on the stack a chance to perform its specific meta-processing. This cooperation, which relies on the "chain of responsibility" design pattern [GHJV95], requires that meta-objects should be aware of it [MMC95]. This is why every meta-object hold a reference to the next one on the chain.

2.3.3 An Example: Logging

Suppose we have a banking application and we want to log the activity of a specific account (e.g. deposits, withdrawals). First we need to build a class of meta-object for logging as shown on figure 3.

The LogMetaObject class defines a field named logStream which references the stream where logs are written (e.g. a stream on some log file). LogMetaObject also defines a method for setting the stream and redefines the method for handling received messages. This method simply prints the message signature into the log stream. Next the default behavior for reception handling is performed.

In order to log the behavior of a given account, we need to add the "log" property to its meta-object. Figure 4 shows a code that :

[3]Instance Variables in the Smalltalk jargon.

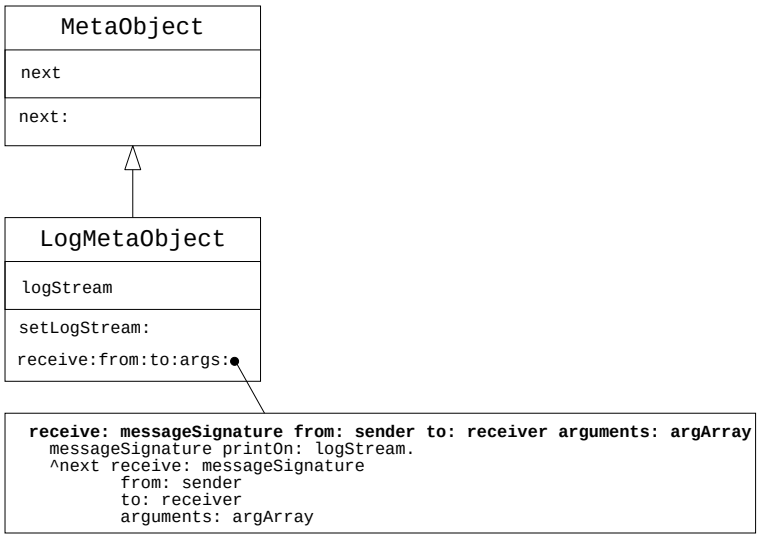


Figure 3: Logging Meta-object class

1. | myAccount myMeta |
2. myAccount := BankAccount new.
3. myMeta := LogMetaObject new.
4. myMeta logStream: (WriteStream...)
5. myAccount addMetaObject: myMeta.

Figure 4: Setting up a specific meta-behavior for a base object

- Declares two temporary variables myAccount and myMeta (line 1).
- Creates a new account and references it using the myAccount variable (line 2).
- Creates a new log meta-object and store it within the myMeta variable (line 3).
- Setup the stream where logs should be stored in (line 4).
- Last, myAccount is linked to myMeta (line 5). As suggested by the protocol used for this linking (addMetaObject:), a base-object can be linked to many meta-objects. Those meta-objects cooperate in order to control the behavior of the base object. Of course, some (or all) these meta-objects can be shared with another base-object.

```
deposit: amount
        balance := balance + amount

withdraw: amount
        balance := balance - amount

transfer: amount to: otherAccount
        otherAccount deposit: amount
        self withdraw: amount
```

Figure 5: Methods Defined by the BankAccount Class

Once myAccount is linked to a log meta-object, its activity gets logged. We suppose that the BankAccount class hold methods for deposit and withdrawals defined in figure 5. So, whenever the myAccount receives the deposit: 100 message, its meta-object (myMeta) takes the control. This is concretized into an implicit message sent to myMeta as shown on figure 6. As a result, first logging is performed and then the deposit: 100 message is evaluated.

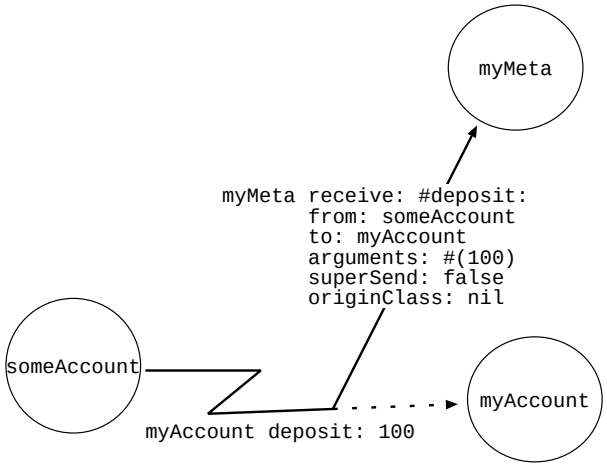


Figure 6: Handling of Reception of the deposit: Message

In the above example, among all instances of the BankAccount, only the one referenced by myAccount is logged. Now, imagine that we want to log all and every instance of BankAccount, including ones that are not created yet. This can be done by sending the addMetaObjectClass: message to the BankAccount class[4] as following:

[4]Remember that likewise Smalltalk, MetaclassTalk treats classes as plain objects that can receive messages

```
BankAccount addMetaObjectClass: LogMetaObject
```

This message makes the BankAccount class link every new instance with a new meta-object instance of LogMetaObject. This link is established on the initialization time of every new bank account. So, logs of every new account go into a specific stream[5].

3 AOP Using Reflection

Building software based on the AOP paradigm requires defining aspects in isolation and then weaving them together. Aspect separation can be achieved thanks to the base-meta separation, while weaving is performed using the meta-link [HL95]. If we draw an analogy with AspectJ [KHH+01b, KHH+01a], methods of meta-object classes play the role of advices while a MOP can be viewed as a set abstract point-cuts. Concreteness is obtained by linking a given base-object to some meta-object, since the meta-object acts on some concrete execution flow (i.e. some concrete join-points such as a particular message send or field access).

The base-meta separation addresses only a part of the AOP issue since not aspects are untangled. Nevertheless, these basic separation and weaving are the core building blocks for supporting AOP using reflection. In the following, we show, through an example, how reflection helps achieving full aspect separation and weaving.

3.1 Example of an Application with Multiple Aspects

Consider the example of a digital bookstore which sells books on the web (see figure 7). Among objects setting up the digital bookstore we have customers, managers, books and orders. The digital bookstore supports various functionalities such as :

- search for books,
- order some books (for customers),
- add/remove books (for bookstore managers),
- update price lists (for bookstore managers).

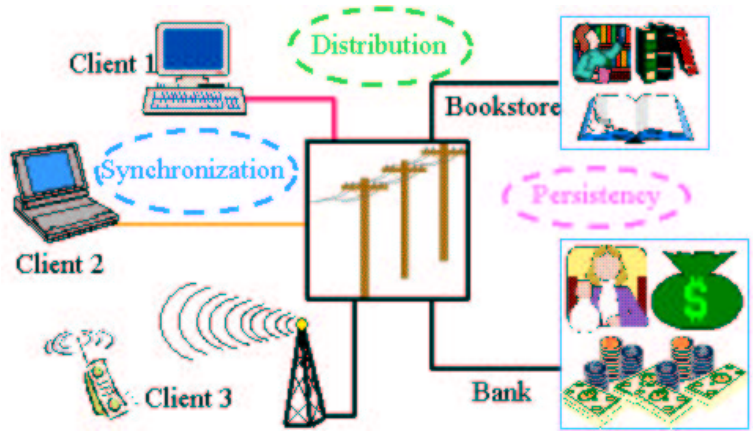


Figure 7: A Digital Bookstore

The digital bookstore system also has several aspects that developers should be concerned with. Obviously, distribution is one of them. The digital library system should interact with

[5]We make the assumption that LogMetaObject initializes the log stream of every new log meta-object.

remote entities (e.g. customers, bank). Another aspect is concurrency. Multiple customers can search for books or make orders while library managers handle orders or update the price list. Yet another aspect is persistency. It is crucial to make objects such as orders or books persist even if the application stops running (for maintenance or due to some crash).

3.2 Separating Aspects

Thanks to the base-meta separation provided by reflection, we can decompose our digital bookstore application as shown on figure 8. On the one hand, aspect are defined at the meta-level using meta-objects. On the other hand, application core functionalities are defined at the base-level using base-objects.

Separating aspects one from another is done by decomposing the meta-level into sets of meta-objects. Each set includes meta-objects specific to a single aspects. So, intersection between sets of meta-objects corresponding to different aspects is empty.

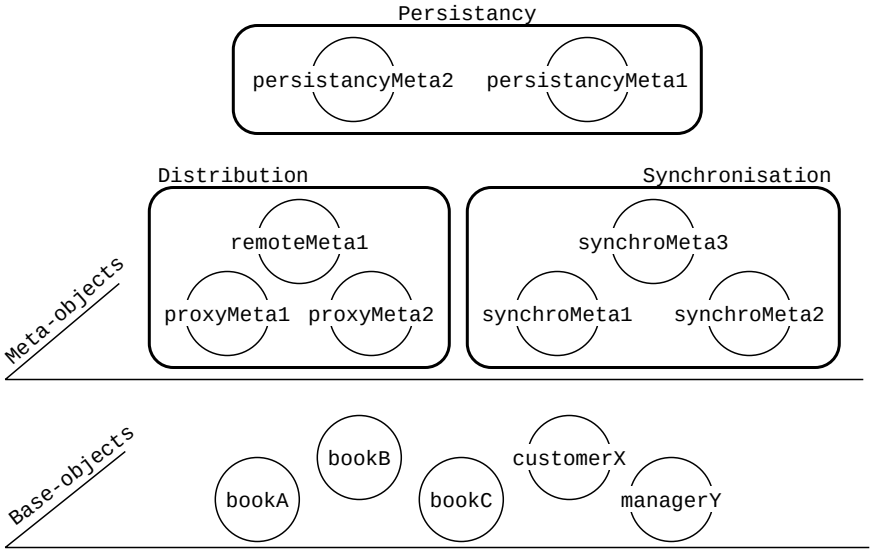


Figure 8: Separating Aspects in Bookstore Application

The set of meta-objects specific to the distribution aspect includes meta-object that make some base-objects be remotely accessible or behave as proxies (possibly with cache). Meta-objects for a proxies forward all received messages to some remote-objects and take care of marshalling. While meta-objects for remote-objects register these latters into some name server and also takes care of marshalling.

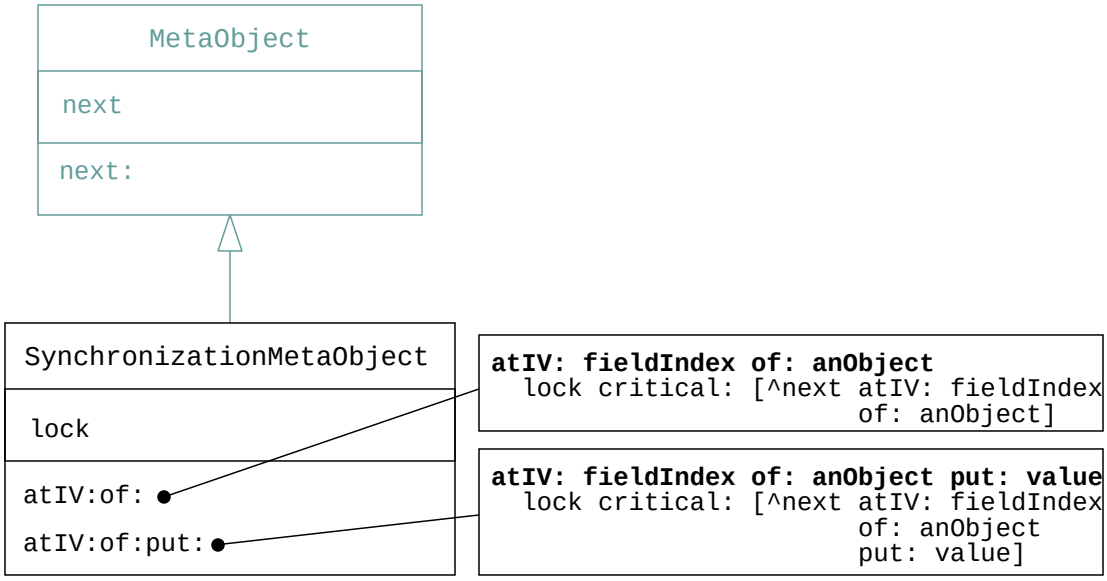
The synchronization aspect is represented by a set of meta-objects that allows defining critical sections of code (e.g. mutually exclusive methods). For example, a such meta-object handles accesses to the price field of a book so that :

- when the value of the price is changed by some thread, other threads attempting either a read or a write access of the same field are locked, and
- when the price is read by some thread, other threads attempting a write access to the price field are locked.

The persistency aspect is represented by a set of meta-objects that define how base-objects should be stored, and when the storage should be updated. As an example, a such meta-object will handle a book field accesses so as to update some relational data-base.

Likewise aspects representation (i.e. sets of meta-objects) are separated, aspects definition are also separated. Infact, the code of each aspect is consists of:

- the definitions of some specific meta-object classes,
- and a *configuration script*, that is the code that :
  - creates meta-objects,
  - performs any required initialization for meta-objects (e.g. linking persistency meta-objects to some data-base),
  - and sets up the meta-link, i.e. links the aspect's specific meta-objects to application base-object.



Configuration script :

1. Book addMetaObjectClass: SynchronizationMeta.
2. Order addMetaObjectClass: SynchronizationMeta.

Figure 9: Definition of the synchronization aspect

As an example, figure 9 provides the code defining the synchronization aspect. This definition is quite simple. There is only one meta-object class named SynchronizationMetaObject, while the configuration script only ensures the creation of a synchronization meta-object for every instance of classes Book and Order. So, an instance of the SynchronizationMetaObject is created and linked to every new instance of one of these two classes.

The SynchronizationMetaObject class defines a new field named lock. This field holds a lock object that forbids critical sections of code be executed by more than one thread simultaneously. Sections of code that should be synchronized using this lock are those where accesses to fields are performed. We ensure this synchronization by making the SynchronizationMetaObject class redefine methods for controlling fields reads and writes. In each of those two methods (atIV:of: and atIV:of:put), the critical section of code is between square brackets.

As stated above an aspect definition distinguishes between :

- aspect specific processing provided by meta-object classes, and

- when to perform this processing (i.e. join-points) provided by configuration script.

Since the specification of the join-point is provided by the configuration script, meta-object classes are generic. So, they can be reused in other aspects which paves the way for aspect reuse.

3.3 Weaving Aspects

In the context sketched in section 3.2, weaving consists in performing configuration scripts of all aspects. As a result, each new base-object involved in multiple aspects, is linked to multiple meta-objects. These meta-object cooperate in order to control the activity of the newly created base-object. The meta-object cooperation is important since it aims solving possible conflicts between "non-orthogonal aspects". Such a conflict materializes when at least two meta-objects that provide two different semantics for a same execution mechanism (e.g. message reception) require be linked to a same base-object.

Since a reflective system gives access to its internal, the shape of meta-objects and their abilities to cooperate can be changed. One possible solution is based on the "chain of responsibility" design pattern [GHJV95]. All meta-objects that are supposed to control the same base-object are arranged in a single chain [MMC95]. The head of the chain is the meta-object that is actually linked to the base-object. Whenever this meta-object takes the control of some base-object activity (e.g. handling of messages received by the base-object), it first performs some meta-processing related to some non-functional concern. Then, the head meta-object forwards information about the activity to control to the next meta-object on the chain. Thus, this latter has the opportunity to perform its specific meta-processing. And so on. Once the last meta-object of the chain performed its meta-processing, the flow of control goes back through the chain.

As an example, consider a book base-object in our digital bookstore. Such an object should be both synchronized and persistent. Each of these two concerns is represented by a specific meta-object. The synchronization meta-object ensures that only one thread can access (read or write) some field of a book. The persistency meta-object ensures that whenever a field value is changed (i.e. write access), the new value is stored on some data base. We organize these two meta-objects in a chain shown in figure 10. Synchronization meta-object is the head of the cooperation chain, and thus it is linked to the book object.

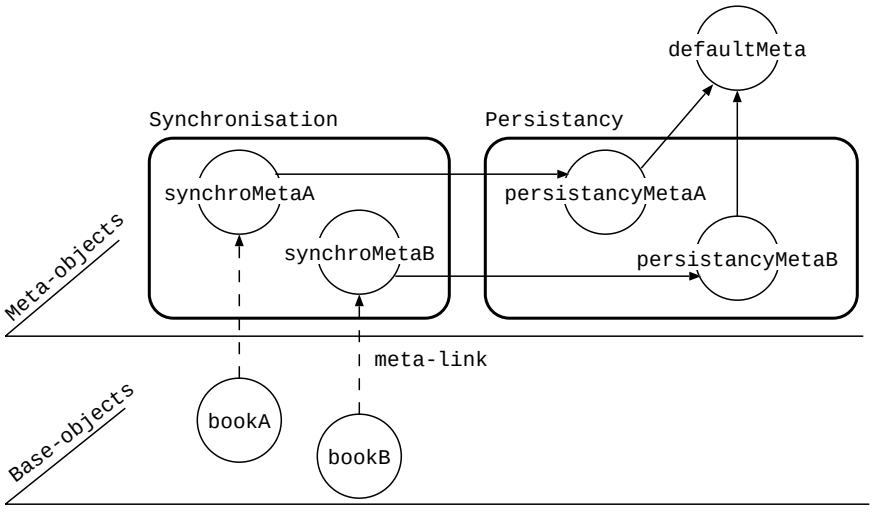


Figure 10: Weaving Aspects Using Meta-Objects Cooperation

In order to show the meta-object cooperation in action let examine what happens when the bookstore manager tries to update the price of some book. The price update means assigning a

new value to the price field of the given book. Figure 11 shows the main steps for the processing of this assignment.

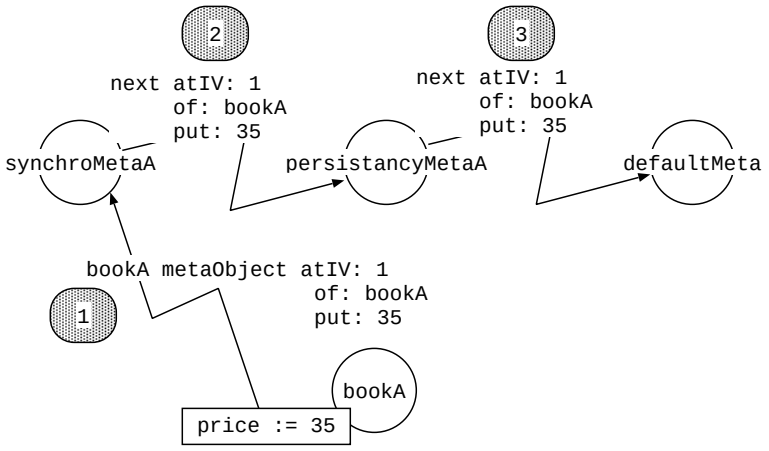


Figure 11: Interaction Between Cooperating Meta-Objects

The evaluation of the assignment of the new price to the price field is performed by sending a message atIV:of:put: to the meta-object linked to the bookA book. Remind that, the first parameter of this message is the index of the accessed field (we suppose that price is the first field of the class Book), the second parameter is the object holding the field and the last parameter is the new value. Note that the message metaObject allows the retrieval of the meta-object linked to some base-object. In the case of the bookA object, its meta-object is synchroMetaA which is in charge of synchronization.

When synchroMetaA receives the atIV:of:put: message (step 1 in figure 11), it first locks field access to other threads. Then, it requests from the next meta-object in the cooperation chain to perform the field access (step 2 in figure 11). The next meta-object in the chain is persistancyMetaA which enforces persistency. When this latter meta-object receives the atIV:of:put: it does update the data-base with new price. Then, it requests from the next meta-object in the cooperation chain to perform the field access (step 3 in figure 11). The next meta-object is defaultMeta which provides the default semantics for the MetaclassTalk MOP. It is defaultMeta that actually writes the new value into the price field. Once the value of the price updated, the flow of control goes back through the inverse path (defaultMeta, persistancyMeta and then synchroMeta).

As stated above, the meta-object cooperation policy presented here is not the only possible one. Many other strategies to perform meta-object cooperation (and hence deal with conflicting aspects) are possible. One can even imagine a completely different approach for weaving. Instead of doing it at the instance (meta-object) level it is possible to do it at the level of meta-object classes. In this context, each base-object is controlled by a single meta-object. This latter is instance of a class that "merges" definitions provided by classes related to aspects relevant to the base-object. The "merge" can for example rely on multiple inheritance or mixin-based inheritance [BC90].

To conclude this section, we should recall that aspect weaving (and particularly weaving of non-orthogonal aspects) is still an open issue in AOP. Although, reflection does not solve or even avoid possible conflicts, it has two advantages: (1) it expresses the aspect weaving issue into a more familiar way which is object or class composition, and (2) it makes possible the exploration of various strategies for weaving and conflicts handling.

4 Discussion

In this section, we discuss advantages and drawbacks of the use of reflection to support AOP.

4.1 Flexibility

Many reflective systems make meta-objects live during run-time (this is the case of the MetaclassTalk MOP that we used in examples of previous sections) [WS99, OB99, PDFS01]. So, on every base-object action (e.g. message reception, access to a field), one or more meta-objects are involved in order to provide the semantics of the action (e.g. how to perform a message). One of the main benefits of this matter of fact is *flexibility*. The set of meta-objects controlling a base-object can be changed dynamically at run-time.

The flexibility provided by reflection make it possible to *weave or unweave aspects dynamically*. Quality of service is among the first benefits of this flexibility one can think of. A system can dynamically adapt its execution according to changes happening in its environment [DL02]. As an example, consider the case of a distributed system where a meta-object that makes some base-object behave as a proxy of a remote object. When the network load increases, the previous meta-object can be replaced by another one that makes our base-object behave as a proxy with cache.

Another possible use of flexibility provided by reflection is maintenance or unplanned evolution of critical systems. Suppose that we found a synchronization bug that introduces a dead-lock in a concurrent application. One can replace meta-objects responsible of dead-locks with new ones corresponding to a new debugged version of the synchronization policy. One can also add a completely new aspect by adding new meta-objects. Changes can be done at run-time without requiring stopping the system.

4.2 Performance

If flexibility is an advantage of reflection, the price to pay it is a non-negligible overhead. Indeed, meta-objects act likewise interpreters[6]. This overhead is even more important since in a reflective system meta-objects can be controlled by some meta-meta-objects.

Various approaches exist to deal with this overhead. One possible solution is to "merge" base and meta levels at compile-time or load-time such as done in OpenC++ 2.0 [Chi95] and OpenJava [CT98]. This approach that had led to work on AspectJ [Tho02], suppresses any "indirection" by flattening the reflective tour. The program is transformed so as to have only one level of objects. However, while considerably speeding up the processing, this solution has the major drawback of sacrifying flexibility.

Another approach consists in restricting the activity of meta-objects to only places where it is required. This approach adopted by Iguana [GC96] and Reflex [TBSN01] avoids hooks introduction and hence jumps from base to meta-level when no meta-processing is required. So, many meta-objects only partially control the activity of base-objects. And even, many objects are not linked at all to any meta-object. This solution makes it possible to limit the overhead to only cases where reflection is required. However, flexibility is restricted to planned scenarios. Also, unplanned evolution is not possible.

An alternative solution consists in using *partial evaluation* [BN00, Sul01]. This approach promise is to keep the flexibility of reflection while drastically reducing performance overhead. Roughly, the idea behind this approach is the following. Assuming that meta-objects will not change, an optimized version of application code is generated. This optimized code is run instead of the original code. In case of a meta-object change, the optimized code is discarded and a new one can be regenerated.

4.3 Other Issues

4.3.1 Complexity

AOP is a goal, for which reflection is one powerful tool [KLM+97]. Indeed, reflection open up the programming language and make it extensible and adaptable. However, reflection have been

[6] except that they don't require parsing.

criticized for its complexity. The underlying concepts tend to be hard to learn and may make programs difficult to understand [VD99].

Nevertheless, we believe that this complexity can be hidden at least partially. Aspect developpers can be provided some high-level (even domain specific) language that compiles into a reflective language. The high-level language eases expressing aspects while restricting the access to available reflective facilities.

4.3.2 Tooling

One of the issues that is faced when building programs using AOP is the lack of appropriate development tools. This is particularly true for debug. The difficulty of building such tools is among the reasons behind this scarcity. Aspects usually can not be tested in isolation. And, it is difficult to retrieve aspect specific code by analyzing weaved code.

Regarding this issue, reflection provides the benefit of using existing development tools. We showed that aspects can be expressed in terms of classes of meta-objects. So, we can make use of class browsers to view and edit their code. Test and debug can also be done using existing tools (e.g. inspectors, debugger, xUnit test framework). Debug is made easier since meta-objects aspects remain decoupled one from another. Indeed, each aspect is represented by a set of meta-objects that can be distinguished from other (meta-)objects even after weaving. One can argue that these tools are rather "low-level" and do not suit aspects. However, (1) they are a first answer to the tooling issue, and (2) they can be extended (especially by means of reflection) to better support aspect development.

4.3.3 Reuse

Software reuse is one of the main goals of software engineering. In order to reuse aspects in different applications, aspects should be generic. Then, the definition of an aspect should not explicitly reference application components. However, weaving requires linking aspects to application components. This is somewhat contradictory with aspect genericity.

By expressing aspects using (meta)objects, reflection allows brings to AOP reuse solutions provided by object-oriented programming (e.g. inheritance). One can easily build new aspects using some meta-object classes that subclass existing meta-object classes used in some other aspects. Furthermore, meta-objects classes can easily be made generic i.e. independent of any application. Indeed, only configuration scripts that link meta-objects to base-objects need to be application specific.

5 Conclusion

In this chapter we presented reflection and how it can be used to support AOP. We described a MOP and showed one possible approach to use it in order to implement isolated aspects and then weave them together.

An aspect can be implemented in isolation since it definition is compound of a set of meta-object classes and a configuration script. Meta-object classes define aspect specific processing (e.g. synchronization) while the configuration script expresses when to perform this processing i.e. it specifies join-points. In fact, the configuration script allows to create meta-objects, initialize them and link them to base-objects. Since the specification of the join-point is provided by the configuration script, meta-object classes are generic. So, they can be reused in other applications and hence paves the way for aspect reuse.

In the above context, the process of weaving consists in evaluating aspects configuration scripts. As a result, each base-object can be linked to many cooperating meta-objects. This relationship can be updated at run-time by adding or removing meta-objects. Thus, the use of reflection provides flexibility. Applications can be adapted or evolved by dynamically weaving and unweaving aspects.

It worth recalling that the approach presented above is only one possible way to support AOP. Many other approaches to represent and weave aspects can be experimented using reflection. Particularly, different strategies for dealing with non-orthogonal aspect weaving can be studied. Reflection is not only a way to support AOP, it is also an ideal testbed to evaluate solutions for AOP open issues.

References

[BC90] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of ECOOP/OOPSLA'90, Ottawa, Canada*, pages 303–311, October 1990.

[BN00] M. Braux and J. Noyé. Towards partial evaluating reflection in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Boston, MA, USA, January 2000. ACM Press. ACM SIGPLAN Notices, 34(11).

[Bou02] N. Bouraqadi. Metaclasstalk: Reflection and meta-programming in smalltalk. MetaclassTalk web page http://csl.ensm-douai.fr/MetaclassTalk, October 2002.

[BS99] M. N. Bouraqadi-Saâdani. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects.* Thèse de doctorat, Université de Nantes, Nantes, France, July 1999. A Smalltalk MOP for Studying Metaclass Commposition and Compatibility: Application to Aspect Oriented Programming (in french).

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceeding of OOPSLA'95*, pages 285–299, 1995.

[CT98] Shigeru Chiba and Michiaki Tatsubori. Yet Another java.lang.Class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[DL02] P.C. David and T. Ledoux. An infrastructure for adaptable middleware. In *DOA 2002, Distributed Objects and Applications*, Lecture Notes in Computer Science. Springer-Verlag, October 2002. to appear.

[GC96] Brendan Gowing and Vinny Cahill. Meta-Object Protocols for C++: The Iguana Aproach. In Gregor Kiczales, editor, *Proceedings of Reflection'96*, pages 137–152, San Francisco, California, April 21-23 1996.

[GHJV95] Erich Gamma, Richard Helem, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GK99] Michael Golm and Jürgen Kleinöder. Jumping to the Meta Level: Behavioral Reflection Can Be Fast and Flexible. In Pierre Cointe, editor, *Proceedings of Reflection'99*, number 1616 in LNCS, pages 22–39, Saint-Malo, France, July 1999. Springer.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KHH+01a] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, October 2001.

[KHH+01b] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceeding of ECOOP 2001*. Springer, June 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of ECOOP'97*, number 1241 in LNCS, pages 220–242. Springer-Verlag, June 1997.

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155, Orlando, Florida, 1987. ACM.

[McA95] Jeff McAffer. Meta-level Programming with CodA. In *Proceedings of ECOOP'95*, volume LNCS 952, pages 190–214. Springer-Verlag, 1995.

[MMC95] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a Methodology for Explicit Composition of MetaObjects. In *Proceedings of OOPSLA'95*, pages 316–330, Austin, Texas, October 1995.

[OB99] Alexandre Oliva and Luiz Eduardo Buzato. Composition of Meta-Objects in Guarana. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems (COOTS'99)*, San Diego, California, USA, May 1999.

[PDFS01] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Jac: a flexible solution for aspect oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of Reflection 2001*, number 2192 in LNCS. Springer Verlag, September 2001.

[Riv96] F. Rivard. Smalltalk: a reflective language. In Gregor Kiczales, editor, *Proceedings of Reflection'96*, San Francisco, California, April 21-23 1996.

[Smi84] Brian C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, POPL'84*, pages 23–35, January 1984.

[Sul01] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, October 2001.

[TBSN01] Eric Tanter, Noury M. N. Bouraqadi-Saâdani, and Jacques Noyé. Reflex - towards an open reflective extension of java. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of Reflection 2001*, number 2192 in LNCS. Springer Verlag, September 2001.

[Tho02] D. Thomas. Reflective software engineering - from mops to aosd. *Journal of Object Technology*, 1(4):17–26, September-October 2002.

[VD99] Kris De Volder and Theo D'Hondt. Aspect-oriented logci meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection (Second Internation Conference, Reflection'99)*, number 1616 in LNCS, Saint-Malo, France, July 1999. Springer.

[WS99] Ian Welch and Robert Stroud. From Dalang to Kava - the Evolution of a Reflective Java Extension. In Pierre Cointe, editor, *Proceedings of Reflection'99*, number 1616 in Lecture Notes in Computer Science, pages 2–21, Saint-Malo, France, July 1999. Springer.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of OOPSLA'88*. ACM, 1988.

[Yok92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of OOPSLA'92*. ACM, 1992.