Malaj: A Proposal to Eliminate Clashes Between Aspect-Oriented and Object-Oriented Programming

Gianpaolo Cugola Carlo Ghezzi Mattia Monga
Gian Pietro Picco

Politecnico di Milano, Dip. di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 I 20133 Milano — Italy
{cugola, ghezzi, monga, picco}@elet.polimi.it

Abstract

Aspect-oriented programming (AOP) aims at providing linguistic mechanisms that enable better separation of concerns in program development. Mainstream techniques, like object-oriented programming, do not provide a satisfactory solution to this problem, in that they do not capture the frequent case of aspects that cross-cut system components, and hence cannot be fully encapsulated. On the other hand, AOP is in its early stages, and it is still unclear what are the fundamental design criteria for aspect-oriented languages (AOLs). A previous evaluation of a general-purpose AOL suggested that the flexibility gained by the ability to freely intertwine aspect and functional code opens several potential conflicts with the principles of object-orientation. Based on this experience, in this paper we present an alternative approach where the vision of a general-purpose AOL is replaced by the vision of a system where several, independent aspects are gathered, each providing specialized support for a single concern. The hypothesis underlying this work is that such a design approach provides a good compromise between flexibility and preservation of good object-oriented and programming principles.

1 Introduction

Engineering work is largely dealing with separation of concerns [1]: designers want to think about one problem at a time. Software engineers learned how decomposition of a complex system into simpler sub-systems can make the problem tractable because the complete solution can be built out of sub-solutions, found relatively independently.

Existing programming languages support the partitioning of software in modular units of functionality. Such parts can be viewed as *generalised procedures* [2] that are assembled to get the desired functionality of the whole system. Several linguistic constructs are aimed at achieving isolation of a concern in a generalised procedure. For example, encapsulation limits the effects of change to localised portion of code; inheritance allows one to incrementally evolve a component by adding new features or redefining existing features; exception handling mechanisms separate the normal flow of execution from the emergency one.

Even optimal functional decompositions omit to encapsulate some concerns because those *cross-cut* the entire system, or parts of it.

As an example, suppose that a Java class is used to describe the pure functionality of certain objects. Additional separate *aspects* [2] may include the definition of:

- constraints on sequences of applicable operations (e.g., to get information from an object one must first apply a setup operation, and then one of a set of assignment operations);
- synchronisation operations to constrain concurrent access to the object (e.g., a consumer trying to read a datum from a queue must be suspended if the queue is empty);
- how objects are distributed on the nodes of a network, either statically or through dynamic migration.
- security or accounting policies (e.g., to get information from an object one must first ask some permission).

In principle, the various aspects should not interfere with functional code, they should not interfere with one another, and they should not interfere with the features used to define and evolve functionality, such as inheritance.

A first approach to aspect-oriented programming (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. We analysed AspectJ [3, 4], probably the best-known, general purpose AOL. We found (see sec. 3) that the "general-purpose" approach adopted results in violating the object-oriented principles of protection and encapsulation, thus increasing the chance that the different aspects could interfere with each other or with the functional code.

In this paper we describe Malaj (see sec. 4), a system supporting aspect-oriented programming in which we take a different approach: we define separate linguistic construct for *specific* aspect domains, giving aspect code limited visibility of the functional code. This way we can reduce the clashes with traditional linguistic features.

2 Aspect-Oriented Programming

Recently, some AOLs were proposed to make aspects clearly identifiable from functional code, which is written by using a so-called component language (CL). AOLs claim that the writing of self-contained and easily changeable aspects is made possible by providing:

1. some syntactic sugar to isolate code for an aspect;
2. a way to identify *join points* in functional code. These are the points where aspect code is introduced;
3. a *weaver*: an engine that is able to mix–not necessarily at compile time–aspect and functional code.

The best known system implementing this approach is probably AspectJ [3, 4].

In AspectJ an aspect is defined via a construct which is reminiscent of the Java **class**: an **aspect**, with 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 **class** 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 objects 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.

The creators of AspectJ, in an early version of their system [1], provided two concern-specific languages: COOL, to control thread synchronisation; and RIDL, to program interactions among remote components. With COOL and RIDL it was possible to define *coordinators* and *portals*, which had total visibility of internal details of objects, but did not have the permission to change their state. Coordinators could specify self and mutual exclusion between class methods and pre/post-conditions on methods execution. Portals could specify if data transfers across site boundaries are made possible by copying objects or transferring a reference to them. Currently, COOL has become a coordination library, whose features are woven into functional code by using the AspectJ engine. In the following section we show the problems we found with this generalised approach.

3 AOP and OOP

In a previous analysis [6] we identified a number of problems and pitfalls that affect currently available AOLs. In particular, focusing on AspectJ[2], 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. As an example, in AspectJ, the aspect code may access the private attributes of a class. This can be useful in some situations, but results in a potentially dangerous breaking of class encapsulation. Imagine a situation in which a class `Foo` has a private variable `i` that needs to be accessed by aspect `Bar`. Imagine also that subsequently class `Foo` is changed by changing type of variable `i` from **int** to **float**. This results in breaking the aspect code. In general, it is not possible to change the internals of a functional unit without changing the aspects that access the private part of that unit.
- *Different aspects.* Suppose (see Figure 1) that a class `Point` exists with two variables `x` and `y` and two methods, `setX` and `setY`. Suppose we have developed an aspect `TraceBefore` to trace the start of execution of methods of class `Point` and an

[1] For a detailed description see [5]

[2] We used version 0.3.0.

aspect `TraceAfter` to trace the end of execution of the same methods. The two aspects work perfectly when applied individually (for example, to trace the start of execution or to trace the end of it). Unfortunately, since they introduce the same method (i.e., the method `print`) with different definitions, they fail when applied together.

- *Aspect code and specific language mechanisms.* One of the best known examples of problems that falls into this category is inheritance anomaly [7]. This term was first used in the area of concurrent object-oriented languages [8, 9, 10] 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 AOP languages, the term can be used to indicate the difficulty of inheriting the aspect code by a subclass. As an example, consider class `Window` in Figure 2. Methods `show` and `paint` cannot be called before method `init` is called. This behaviour is controlled by the aspect `WindowSync`. Now consider class `SpecialWindow` in Figure 3. It redefines method `show` in such a way that it does not require a previous invocation of method `init`. (Note that this way of subclassing `Window` is consistent with the OO type theory, which requires subclasses not to strengthen the precondition for redefined methods.) In principle, it should be possible to "inherit" the `WindowSync` aspect just by modifying the code associated to method `show` (e.g., replacing it with the empty sequence). Unfortunately, this is not possible and it is necessary to rewrite entirely the aspect code (see aspect `SpecialWindowSync` in Figure 3).

Our claim is that these conflicts result more from the linguistic choices made in developing AOLs, rather than from intrinsic limitations of the approach. In particular, a general-purpose AOL, with full visibility of the internal details of the functional modules, provides the maximum expressive power at the expense of violating the principles of protection and encapsulation. An alternative approach, which we followed in designing Malaj, is to predefine the set of possible aspects an AOL should deal with, and then provide ad-hoc constructs to implement these aspects, providing limited visibility of the features of the functional module to which the different aspects apply. As the next sections will show, this 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 use the AOL to code any possible aspect, but it eliminates the problems encountered with a general purpose AOL.

4 Malaj: a Multi Aspects LAnguage for Java

As mentioned in the previous section, Malaj is not a general purpose AOL. It focuses on a well-defined set of aspects (namely, synchronisation and relocation), and provides a different linguistic construct for each aspect. We embrace the same philosophy behind COOL and RIDL, emphasising the need for restricted visibility and for clear rules of composition with traditional constructs.

As its name says, Malaj is an aspect-oriented extension to Java. The Malaj core language is a reduced version of Java, which 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.

To describe the Malaj constructs for aspect programming, the following sections refer to a common example: a very simplified electronic commerce system. The system (see Figure 4) is composed of two main classes: `Shop` and `Customer`. The `Shop` class provides methods to add and remove items from the list of available articles. It also exports two methods to ask for the price of a specific article and to deliver the article to a specific address after it has been bought. Class `Customer` models the behaviour of an e-commerce agent: it goes through a list of shops in search of a given article looking for the best price. At the end of the process, it buys the article. Finally, class `ECom` creates two customers and starts them.

4.1 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;

```
class MakePoint {
  public static void main(String args[]){
    Point p=new Point();
    p.setX(1);
    p.setY(1);
  }
}

class Point {
  int x,y;
  public Point(){
    x=y=0;
  }
  public void setX(int x) {
    this.x=x;
  }
  public void setY(int y) {
    this.y=y;
  }
}
```

```
aspect TraceBefore {
  introduce private void
    Point.print(String methodName) {
    System.out.println("Tracing method "
        +MethodName
        +" before");
    System.out.println("x="+x+"  y="+y);
  }
  advise void Point.setX(int i),
        void Point.setY(int i) {
    static before {
      print(thisJoinPoint.methodName);
    }
  }
}
aspect TraceAfter {
  introduce private void
    Point.print(String methodName) {
    System.out.println("Tracing method "
        +methodName
                  +" after");
    System.out.println("x="+x+"  y="+y);
  }
  advise void Point.setX(int i),
        void Point.setY(int i) {
    static after  {
      print(thisJoinPoint.methodName);
    }
  }
}
```

Figure 1: An example of clash between two aspects

```
class MakeWindow {
  public static void main(String args[]){
    Window w=new Window();
    w.init();
    w.show();
  }
}

class Window {
  public void init() {
    // ...
  }
  // Requires initialization
  public void show() {
    // ...
  }
  // Requires initialization
  public void paint() {
    // ...
  }
}
```

```
aspect WindowSync {
  introduce boolean Window.initDone=false;
  advise void Window.init() {
    static after {
      initDone=true;
    }
  }
  advise void Window.show(),
        void Window.paint() {
    static before {
      if(!initDone)
       System.out.println(
      "Error: init never called");
    }
  }
}
```

Figure 2: An aspect to control the sequence of invocation of different methods

```
class SpecialWindow extends Window {
  // This version of show does not
  // need any initialization
  public void show() {
    // ...
  }
}
```

```
aspect SpecialWindowSync {
  introduce boolean Window.initDone=false;
  advise void Window.init() {
    static after {
      initDone=true;
    }
  }
  advise void Window.paint() {
    static before {
      if(!initDone)
   System.out.println(
      "Error: init never called");
    }
  }
}
```

Figure 3: An example of inheritance anomaly

```
public class ECom {
    Shop [] shops;
    Customer paul, john;

    public ECom(){
        Shop [] shops = new Shop[5];
        for(int i=0; i<5; i++) {
            shops[i] = new Shop();
            // add new articles to shops[i]
            shops[i].addArticle("book", 40−i);
            shops[i].addArticle("CD", 10+i);
        }
        paul = new Customer(shops,
                            "paul home","book",
                            60);
        john = new Customer(shops,
                            "john home","CD",
                            30);
    }

    public void startShopping(){
        paul.start();
        john.start();
    }

  public static void main(String[] args){
      ECom e = new ECom;
      e.startShopping();
  }
}
public class Shop {
  private Hashtable goodies;
  public Shop() {
    goodies=new Hashtable();
  }
  public void addArticle(String article,
    int price) {
    goodies.put(article,
  new Integer(price));
  }
  public void rmArticle(String article) {
    goodies.remove(article);
  }
  public int query(String article) {
    return ((Integer)goodies.
     get(article)).intValue();
  }
  public void deliver(String article,
        String address) {
      // deliver the article to the
      // address specified
  }
}
```

```
public class Customer extends Thread {
  private Shop bestShop;
  private int bestPrice;
  private Shop[] shops;
  private String myAddress;
  private String desire;
  private int wallet;

  public Customer(Shop[] shops,
    String address,
    String article,
    int wallet) {
    this.shops = shops;
    myAddress = address;
    desire = article;
    this.wallet=wallet;
    bestShop = null;
    bestPrice = wallet;
  }

  public void shopping(Shop s) {
    int price = s.query(desire);
    if(price <= bestPrice) {
      bestShop = s;
      bestPrice = price;
    }
  }
  public void buy() {
    if(bestShop != null) {
      bestShop.deliver(desire, myAddress);
      System.out.println("I bought a "+
    desire+
    " at "+
    bestShop);
    }
  }
  public void run() {
    for(int i=0; i<shops.length; i++) {
      shopping(shops[i]);
    }
    buy();
  }
}
```

Figure 4: An example of a simplistic e-commerce application

2. the call violates some precondition and the caller is suspended until the condition becomes true;

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. Figure 5 shows the guardians for classes `Shop` and `Customer` mentioned above.

For each class `C`, the guardian `G` of `C` basically represents the set of **synchronized** methods of `C`. As an example of the **synchronized** statement see the guardian `ShopGuardian` in Figure 5. `G` expresses also the conditions that, if not satisfied, result in an exception when `m` is called (i.e., the **deny** guards), and the conditions that, if not satisfied, result in suspending the caller of `m` (i.e., the **suspend** guards). As an example of the **deny** and **suspend** statements, see methods `addArticle` and `deliver` of guardian `ShopGuardian` in Figure 5, respectively. Observe that deny guards are always considered before suspend guards and they are considered in the order in which they appear in the code. This

means that if different deny and suspend guards are true, only deny guards are considered and among them the first is taken and the exception it defines is returned to the caller.

A guardian may include also a set of local attributes and method definitions to code guards that depend on state conditions (e.g., attribute `elements` of guardian `ShopGuardian` in Figure 5). Finally, for each method `m` of the guarded class, the guardian may introduce a fragment of code to be executed before or after `m` (e.g., method `addArticle` of guardian `ShopGuardian` in Figure 5). 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 library. `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 (see Figure 5). 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 [11], the precondition of a sub-class 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**().

4.2 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 (see example in Figure 6).

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:

```
guardian ShopGuardian guards Shop {
  HashSet elements=new HashSet();
  synchronized {
    addArticle, removeArticle,
    query, deliver
  }
  void addArticle(String article,
                  int price):
    deny A: (price<0) with new PriceTooLow();
    before {
elements.add(article);
    }
  void removeArticle(String article):
    deny A: (!elements.contains(article))
            with new ArticleNotFound();
    before {elements.remove(article);}
  int query(String article):
    deny A: (!elements.contains(article))
            with new ArticleNotFound();
  void deliver(String article,
               String address):
    suspend A: (!elements.contains(article));
}
```

```
guardian CustomerGuardian guards Customer{
  void Customer(Shop[] shops,
                String address,
                String article,
                int wallet):
    deny A: (shops==null || shops.length<1)
            with new NotEnoughShops();
         B: (wallet<1) with new NotEnoughMoney();
}
```

Figure 5: An example of guardians for an e-commerce application

```
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.

Figure 6 shows the relocators for classes `ECom` and `Customer` introduced above. After creation of an `ECom` instance, some shops are distributed in a worldwide market. Customers pin their wallet in a secure site and query around for best prices.

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 and Future Work

In this paper we presented the main concepts of Malaj, a new system supporting aspect-oriented programming. The design criteria where inspired by earlier experience with general-purpose aspect-oriented languages. We envision Malaj as a collection of concern-specific aspect languages, built on top of a subset of the Java language. In this paper, we discussed how the synchronization and relocation aspects can be defined in Malaj. The ultimate goal is to cover a spectrum of concerns far beyond the two presented here, and to complement the programming support with a formal model that can be used to reason about program construction and aspect interaction.

Whether or not this approach is effective in practice is still an open question at the present, and one that can be answered only empirically, along the lines of [12, 13].

A prototype implementation of Malaj exists for the aspects described in this paper. The next steps of our research will extend Malaj to other aspects, will start an experimental assessment of the approach, and will provide development tools for building Malaj applications.

```
relocator EComRelocator relocates ECom{
  Site [] market;
  EComRelocator(){
    market = new Site[shops.length()];
    for(int i=0; i<shops.length(); i++){
      // just an example...
      Site[i] = new Site("host[i]");
  }
  after ECom(){
    for(int i=0; i<shops.length(); i++){
      try{
        this.takeOwnership(shops[i]);
        this.pin(market[i], shops[i]);
      }
      catch(Exception e){
        e.printStackTrace();
    }
  }
}
```

```
relocator CustomerRelocator relocates Customer{
  Site secureHome = new Site("home");
    after Customer(){
      try{
        // Customer creates wallet
        // next instruction is redundant
        this.takeOwnership(wallet);
        // pinned wallet doesn't follow me
        this.pin(secureHome, wallet);
      }
      catch(Exception e){
        e.printStackTrace();
      }
    }
    before void shopping(Shop s){
      this.expressInterest(s);
    }
}
```

Figure 6: An example of relocators for an e-commerce application

References

[1] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[2] 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.

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

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

[5] C. I. V. Lopes, *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, nov 1997.

[6] 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), jul 1999.

[7] 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.

[8] A. Yonezawa and M. Tokoro, eds., *Concurrent Object-Oriented Programming*. Cambridge, Mass.: The MIT Press, 1987.

[9] G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, pp. 125–141, Sept. 1990.

[10] O. Nierstrasz, "Composing active objects," in *Research Directions in Concurrent Object-Oriented Programming* (P. W. G. Agha and A. Yonezawa, eds.), pp. 151–171, MIT Press, 1993.

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

[12] G. Kiczales, E. L. Baniassad, and G. C. Murphy, "An initial assessment of aspect-oriented programming," in *Proceedings of the 21st International Conference on Software Engineering*, (Los Angeles, CA), may 1999.

[13] M. A. Kersten and G. C. Murphy, "Atlas: A case study in building a web-based learning environment using aspect-oriented programming," Tech. Rep. TR-99-04, Department of Computer Science – University of British Columbia, 201-2366 Main Mall – Vancouver BC Canada V6T 1Z4, apr 1999.