Suitability of Object and Aspect Oriented Languages for Software Maintenance

Khalid Al-Jasser Peter Schachte Ed Kazmierczak
The University of Melbourne
Australia
{aljasser,schachte,ed}@cs.mu.OZ.AU

Abstract

Program maintenance is an important and frequently a difficult part of the software life cycle. One reason for its difficulty is the presence of crosscutting concerns*: aspects of a program that cannot be confined to a single program component. Crosscutting concerns defy the standard wisdom of program design that individual components should be highly cohesive (they should address only one concern) and loosely coupled (they should not share concerns with other components). We consider several approaches to a simple maintenance task in both object-oriented and aspect-oriented languages, analyzing how well they maintain high cohesion and low coupling. We conclude that none of these approaches is entirely satisfactory, and present a few changes to aspect oriented programming language design that would better support maintenance in the face of crosscutting concerns.*

1 Introduction

Maintenance is a central part of the software lifecycle, commonly identified as accounting for more than half of the cost of software development. It is not surprising, then, that suitability for maintenance has been a key feature in the design of many programming languages. The *robustness* of existing programs in the face of maintenance, that is how strongly they resist introducing faults during maintenance, is one important determinant of program maintainability. Two other important determinants are *sustainability* and *scalability*: how maintainable does maintenance leave a program and how does a program's size impact its maintainability? Also important for maintainability, as well as for initial development, is the *reusability* of existing program components. We consider how well existing programming languages and methodologies support all these characteristics as programs are maintained.

Probably the greatest single factor in determining how maintainable a program will be is how well structured it is, that is, how well it is divided into separate components. In proposing structured design, Stevens *et al.* [14] suggest that two key considerations in choosing a program decomposition is how high is its level of *cohesion* (how well each component focuses on a single concern of the program) and how low is its level of *coupling* (to what extent do components share concerns). Similarly, Dijkstra [4] suggested that *separation of concerns*, or focusing on one aspect of a design at a time, in one component at a time, is essential to the construction of a good design. A program that does not separate concerns is difficult to maintain because a change to a single concern may need to be made in many parts of the program. By *concern* we mean any aspect of a program's desired behaviour; any individual requirement.

Dijkstra also wrote "even if not perfectly possible, [separation of concerns] is yet the only available technique for effective ordering of one's thoughts" [4]. Programming languages, providing a limited set of abstractions, cannot support every way to separate concerns that we may informally conceive; different separations of concerns are possible in different programming languages. We use the term *crosscutting concern* [10] to describe any concern that relates to a program component other than its central focus. That is, crosscutting concerns are those that interfere with the cohesion of individual components. Monitoring, synchronization, and error-handling are all considered crosscutting concerns. Each crosscutting concern is thus *tangled* in the code of other concerns, thus making the other concerns difficult to maintain. Moreover, it may be *scattered* across many program components, making the concern itself difficult to maintain.

Different programming languages and paradigms support different sorts of components, such as procedures, functions, classes, and templates, but virtually every one requires components to be named where they are defined and referred to by name where they are used. This ensures each use of a component is *visible* in the program. The definition of many sorts of components specify any parameters that may be supplied, defining an *interface* between the component and its clients. The interface constitutes the

Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)
0-7695-2778-7/07 $20.00 © 2007 IEEE

agreement between the developer and users of the component, permitting the component to be developed separately from its uses. This convention of explicit interfaces and visible component uses is so common across programming languages that it may seem inevitable, but we shall observe later that it is not, and failing to follow it is undesirable. Furthermore, many programming languages check component uses during compilation to ensure that the interface is adhered to, enhancing program robustness by detecting faults before testing begins.

The present paper investigates how well current programming languages support separation of concerns. We shall focus on object oriented programming languages (using Java as an exemplar) as they are currently the most popular, and aspect oriented languages (using AspectJ), because they are designed specifically to promote separation of concerns. However, most of our conclusions apply equally well to imperative, functional, logic, and other programming languages.

We approach this problem from the standpoint of programming language design. An alternative approach would be to consider how a sophisticated development environment could assist the programmer in maintaining crosscutting concerns. This is a promising research direction, but our view is that if programming languages themselves can better support separation of concerns, that would be preferable to a solution that heavily relies on sophisticated tools.

The remainder of the paper is organized as follows. Section 2 discusses crosscutting concerns in more depth. Section 3 briefly introduces aspect oriented programming and AspectJ. Section 4 discusses several approaches to add a new concern to an existing program, considering whether the resulting program is still maintainable and concludes that neither object oriented nor aspect oriented programming is entirely satisfactory for program maintenance. Section 5 suggests a new approach to improve visibility of crosscutting concerns and hence maintainability of the whole system. It also shows how the proposed approach helps in producing *reusable* crosscutting concerns. Section 6 outlines some related work and, finally, Section 7 concludes this paper.

2 Crosscutting Concerns

To better understand crosscutting concerns in object oriented languages, we classify them according to the breadth of their effect:

1. System-wide crosscutting
2. Class located crosscutting
3. Object located crosscutting
4. Method located crosscutting

This classification follows Hanenberg and Unland [15], except that we have added item 1. Each of these sorts of crosscutting has different characteristics, and may merit different treatment.

System-wide crosscutting refers to those concerns that cannot be restricted to a single class. For example, a concern that requires the logging of every message sent during program execution would be considered system-wide crosscutting. Such concerns typically have the flavor of "policies." Such concerns are generally difficult to implement, as they tend to require wholesale modification to the original program. However, such concerns tend to be fairly uncommon. They are often only needed during program development, and are removed from the delivered system.

Class located crosscutting describes a concern that spans all instances of a single class. This sort of crosscutting is very common. Synchronization, that is ensuring that certain methods of a class are executed to completion before another execution thread begins executing one of those methods on the same object, is an example of class located crosscutting. Modifying a class to implement synchronization is not generally difficult, but does tend to tangle synchronization logic into the core logic of the class, decreasing its cohesion. Furthermore, it fails to reuse the logic of synchronization.

Object located crosscutting is similar to class located crosscutting, but involves only some instances of the affected class. Whether a particular instance is affected is determined at the time the instance is created. This sort of crosscutting is also common. Object monitoring, that is arranging that every change to an instance triggers a message to another object reporting the change, may be an example of object located crosscutting. This is somewhat harder to implement, as we discuss in Section 4.

Finally, method located crosscutting refers to concerns that require access to the internal structure of methods. The visualization of a binary search, showing how the upper and lower bounds converge on the desired value, is an example of method located crosscutting. Separating the visualization logic from the searching logic would improve cohesion. However, this improved cohesion will be a Pyrrhic victory if it is achieved at the cost of a high degree of coupling between the two concerns. Separating such low-level concerns with a well-defined interface is often very difficult. Fortunately, this kind of crosscutting seems to be rare.

In the present paper we focus on class and object located crosscutting, as they appear to be most prevalent.

3 Aspect Oriented Programming

Aspect oriented programming (AOP) [10] is relatively a new programming paradigm originated at Xerox PARC specifically to better support separation of concerns. Aspect

oriented programming languages provide a new abstraction allowing encapsulation of crosscutting code in separate components called *aspects*. This separation greatly improves the cohesion of the main and crosscutting logic.

AspectJ [10, 9] is a general-purpose aspect oriented version of the Java programming language. The main construct in this language is the *aspect* which provides a modular way to encapsulate crosscutting concerns. Compiling an AspectJ program is performed in two steps: weaving and compiling. The first step glues aspects together with classes in a way as if the code in the aspects is scattered across the core classes. The second step does the normal Java compilation using the `javac` command. AspectJ uses the following constructs to create aspects:

Join-point a join-point is simply any point of execution in the program whose behavior can be modified by an aspect. In AspectJ, all of the following are considered join-points: method call, method execution, returning of a method, object construction, field accessing, field updating, and exception handling.

Pointcut pointcuts are used inside aspects to specify a join-point or a set of join-points in main modules (or even in other aspects).

Advice advice is a method-like construct that is used to encapsulate the crosscutting code. In AspectJ, there are three kinds of advice: `before`, `after`, and `around`. `Before` advice, as the name suggests, runs before executing the specified join-point. Similarly, `after` advice runs after the join-point. `Around` advice runs instead of the body of the join-point, but programmers may execute the join-point inside the advice using the `proceed()` statement.

3.1 Class-Aspect Association

In AOP languages, crosscutting concerns are encapsulated inside aspects. Therefore, there must be a way to connect these aspects to the target classes. In AspectJ, for example, the interface or relation between the main classes and the aspects is defined inside the aspect itself. Kersten and Murphy [8] identified four types of class-aspect association: 1) *open*, 2) *closed*, 3) *class-directional*, and 4) *aspect-directional*. Open association happens when both the class and the aspect do not include any information about each other. A wildcard-based pointcut in AspectJ is a clear example of how this can be achieved. Closed relation happens when both the class and the aspect refer to each other. This kind strongly ties each module to the other. The third type, class-directional, is the one that is followed by current aspect oriented languages. It describes the case when the aspect includes information about the class it is targeting, but not the opposite. Finally, aspect directional association allows the class to know about the aspect but not the converse.

In current AO languages, only open and class directional associations are supported, which makes the crosscutting concern *invisible* to the class. We use the term *visibility/invisibility* to indicate whether a certain crosscutting logic is visible in the main module or not. In other words, if we can identify the various crosscutting concerns that affect a certain class by looking at the class or the client application, then the crosscutting concern is *visible*.

4 Implementing Crosscutting Concerns

In this section we discuss some approaches used in OO and AO languages to implement crosscutting concerns, considering how well each maintains the desirable characteristics discussed in the introduction: high cohesion, low coupling, reusability, and robustness. We also consider whether each approach is sustainable and scalable.

Listing 1. Bank account class

```
public class BankAccount{
  private double balance;
  public void deposit(double amount){ ... }
  public void withdraw(double amount){ ... }
  public double getBalance() { ... }
}
```

We will use a simple bank account class as an illustrative example, extending it with new crosscutting concerns. Listing 1 shows a Java implementation of the basic bank account functionality. As a first maintenance task, we add the requirement that certain bank accounts should be monitored, that is, changes to the balance should trigger a message to another object informing it of the new balance. Secondly, we require that every bank account is synchronized, so that multiple simultaneous updates to the balance from different execution threads cannot interfere with one another. Since this concern involves every instance of the class, it is a case of class located crosscutting. Also, to see how sustainable our approaches are, we consider whether the maintained program handles future changes to the requirements, such as introducing a new method to compute and credit interest payments.

4.1 Direct Modification

The most straightforward way to implement newly introduced requirements is to directly modify the code to support them. For the monitoring requirement, we modify the `deposit` and `withdraw` methods so that they update the

client application with changes made to the balance. Standard software design principles would suggest that we create a private `setBalance` method, and use this wherever the balance needs to be modified; this would create a single point of control for the monitoring logic. To handle synchronization, we also modify `deposit` and `withdraw` to act as critical regions.

This approach is certainly the simplest and most obvious. It is easy to debug, since all the logic involved is readily visible in the class itself. Since all code remains in a single class, it maintains the low coupling of the original class. It also maintains the robustness of the original, as all interfaces are maintained and checked by the compiler (although we shall shortly see that robustness is reduced in some ways). However, the cohesion of the original class is reduced, since the monitoring and synchronization logic is tangled with the main logic of the class. Similarly, reusability is reduced, since the original bank account class, without the added requirements, is no longer available. Likewise, the newly introduced monitoring and synchronization concerns are not available as reusable components.

The observed decrease in cohesion also suggests that the direct modification approach will not be sustainable or scalable, a guess confirmed by considering the introduction of a method to credit interest earned. It is essential to the monitoring requirement that all changes to the balance be made through the new `setBalance` method. Yet nothing stops the programmer from directly modifying the balance. Similarly, the new `creditInterest` method must be a critical region to maintain synchronization, and again nothing ensures this. Clearly there is some reduction in robustness from this maintenance approach.

Finally, note that direct modification is not ideal for object located crosscutting, since it adds the new concern to every instance of the class. Here every account is monitored, although only some accounts need to be. This can be worked around by having every monitoring action check if *this* particular instance is meant to be monitored before performing any actual notifications.

4.2 Subclassing

A more systematic approach introduces subclasses to add new concerns. To add monitoring logic to the `BankAccount` class, we would introduce a new `MonitoredBA` class as a subclass of `BankAccount` class which extends its behaviour by overriding methods to implement the new concern. Similarly, we would introduce a new `SynchronizedBA` class for the synchronization concern. This approach maintains the original `BankAccount` class unchanged, leaving its reusability unaffected, and introduces new reusable subclasses as well, although the monitoring and synchronization logic are still not encapsulated in reusable components. Because the original `BankAccount` class remains, whenever a new account object is created, the application can choose whether to create a `BankAccount` class or a `MonitoredBA`, so this approach is suitable for object located crosscutting. Conversely, contrary to our requirement, not all accounts are synchronised, so subclassing is not suitable for class located crosscutting. Certainly a hybrid approach is possible, using subclassing for object located crosscutting and direct modification for class located crosscutting, but this hybrid still suffers all the drawbacks of each of these approaches when multiple concerns of each type are added.

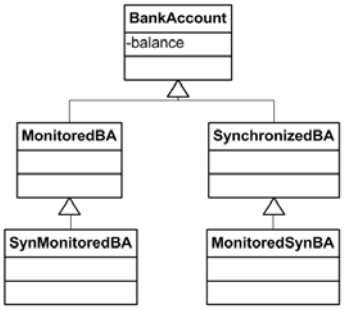

Figure 1. Class diagram for the subclassing method

However we have not satisfied all the requirements. In addition to a synchronized and a monitored bank account, we also need a class that is both synchronized and monitored. This can be implemented as a subclass of either of the new classes, adding synchronization to the `MonitoredBA` or adding monitoring to `SynchronizedBA`. In our simple bank account example, the two crosscutting concerns are independent (it doesn't matter whether the monitoring is handled inside the critical region or after it), so only one of these new classes need be implemented. In some similar cases, multiple crosscutting concerns will interact, and all combinations of subclasses will need to be implemented. Were that the case for this example, our class hierarchy would resemble that shown in Figure 1. Even in this simple case, the code for one of the crosscutting concern needs to be completely duplicated, creating significant maintenance overhead. Although there is not strictly speaking high coupling between, say, the `MonitoredBA` and `SynMonitoredBA` classes, it introduces all the maintenance problems associated with high coupling, since every change in one requires a corresponding change in the other, and because there are no semantic checks performed by the compiler to ensure that the corresponding changes are made. Worse still, as new crosscutting concerns are added, the inheritance tree will grow exponentially, as will the number of duplicates. Clearly this approach is not scalable or sustainable.

4.3 The Decorator Design Pattern

In the subclassing approach, crosscutting logic was separated from `BankAccount` class by putting it into a subclass. The decorator pattern [5] instead puts the crosscutting logic in a sort of "wrapper" class which, in effect, delegates all bank account operations to an actual `BankAccount` object, after performing the operations necessary for monitoring. These wrappers can be chained together, with the final object in the chain being the `BankAccount`. According to Gamma *et al.* [5], the intent of the decorator pattern is to:

> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The decorator pattern has some advantages over previously discussed approaches. Like the subclassing approach, the original `BankAccount` functionality is preserved, so the original class is reusable. Also like subclassing, when a bank account is created, the programmer chooses which sort to create, as follows:

```
ABankAccount ba1 = new BankAccount();
ABankAccount ba2 =
  new MonitoredBA( new BankAccount() );
ABankAccount ba3 =
  new MonitoredBA(
        new SynchronizedBA( new BankAccount() )
  );
```

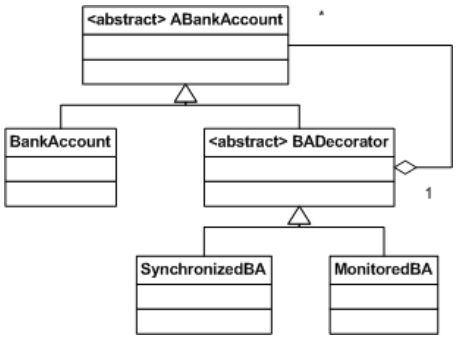

Figure 2. Implementing Decorators for the BankAccount class

Each concern is implemented in a single separate class, including the original bank account, so cohesion is high and coupling is low. Each concern added requires adding only one new class, so it is scalable and sustainable. The original class is reusable, but the crosscutting concerns are not, because they must be specialized for the class they "wrap."

Because each time a bank account is created, the exact combination of wrappers is specified, it is well suited to object located crosscutting. However, not all bank accounts created are syncronized as required; it is not well suited to class located crosscutting.

One drawback of the decorator pattern is its unnatural inheritance hierarchy, shown in Figure 2. Note that, in addition to the `BankAccount`, `SyncronizedBA`, and `MonitoredBA` classes we expect to need, we also need and second, abstract `ABankAccount` class, as well as an abstract `BADecorator` class which delegates every bank account message to another `ABankAccount` object. Where a bank account object is created, the program must create an instance of the `BankAccount` class, however, variables meant to hold a bank account object must be declared as the abstract `ABankAccount` type, to permit decorated bank accounts to be used. Thus although this approach scales well, it requires a bit of extra infrastructure to be built and maintained. A smaller disadvantage is that long chains of wrapper classes can reduce program performance. Every message to the object must go through the entire chain of wrappers, even if every wrapper only forwards the message unchanged.

In summary, the decorator approach exhibits all the desirable characteristics for object located crosscutting except for reusability of crosscutting concerns. However, it is perhaps a bit heavyweight to use for every crosscutting concern.

Listing 2. Aspect for the bank account class

```
public aspect BankAccountMonitoring {
  pointcut UpdatingMethodsPC
       (BankAccount account, int amount):
    (
    execution( void BankAccount.deposit(int) ) ||
    execution( boolean BankAccount.withdraw(int))
    )     &&  target(account) &&  args(amount);

  after(BankAccount account, int amount):
    UpdatingMethodsPC(account,amount){
      // monitoring code
  }//end after
}
```

4.4 Aspect Oriented Programming

Finally we consider using aspect oriented programming [10], in particular AspectJ [9], to solve the problem of crosscutting concerns. For our example, we can implement the monitoring and synchronization logic in separate aspects, as illustrated in Listing 2. The AspectJ compiler then "injects" the monitoring code into the `BankAccount` class.

This implementation does not require any modifications to the `BankAccount` class, maintaining its high cohesion. Moreover, the aspect can simply be removed or disabled,

returning the BankAccount class to its original behavior. Thus this approach maintains the cohesion of the original class and allows creation of cohesive new components (aspects) for the crosscutting concerns. Furthermore, since each new crosscutting concern requires only a single aspect, this approach avoids the scalability and sustainability problems of the subclassing approach.

Unfortunately, coupling presents a significant problem for aspect oriented programming. As shown in Listing 2, the BankAccountMonitoring aspect is tightly coupled to the BankAccount class. Myers [12] distinguishes five levels of coupling; this example exhibits control coupling, Myers' middle level. This level of coupling results from the lack of an agreed interface between the class and the aspect. If an interface could be specified, any change to either the class or the aspect that preserved the interface would not require the other to be modified. Lacking such an interface, any change to BankAccount class may require the BankAccountMonitoring aspect to be revisited. Consider the introduction of a creditInterest method to the BankAccount class. Invocations of this new method would not be captured by our BankAccountMonitoring aspect, leading to incorrect behavior.

This defect could be fixed by writing the pointcut to catch all changes to the private balance instance variable, but this approach changes the coupling between BankAccount and BankAccountMonitoring to content coupling, the worst possible sort. This violates the principle of information hiding [13], which suggests that decisions that may need to be changed should be kept in a private part of the module involved, and no other module should be privy to those decisions. Once content coupling is created, *every* change to the class, whether to public or private members, interface or implementation, requires careful review of every aspect content coupled to that class. Even simply modifying our example aspect to catch all methods that modify the balance would mean that just consistently renaming the (private) balance member throughout the class would break the aspect. This fragility might be managable if very few aspects are involved, but it quickly becomes overwhelming where there are many, so the aspect oriented approach is clearly unsustainable.

One factor that exacerbates this problem is the fact that the programmer editing the class source code cannot see what aspects affect the code they are editing. They may be forced to resort to searching through the texts of all aspects in the system, and even then do not know what to search for. Clearly, appropriate development tools could help in this search, but it would be preferable if the search were not necessary. In any case, it would seem prudent to document in the source code of any class affected by any aspects exactly what assumptions the aspects make about the class.

Listing 3. Using metadata in the bank Account Class

```
public class BankAccount{
  ...
  @UpdatingMethod
  public void deposit(double amount){ ... }

  @UpdatingMethod
  public void withdraw(double amount){ ... }
}
```

```
public aspect BankAccountAspect {
    pointcut
  UpdatingMethodPC(BankAccount account,int amount):
    execution(
      @UpdatingMethod  void BankAccount.*( int )
    )
    &&  target(account) &&  args(amount);
    ...
}
```

That is, the interface between classes and aspects should be documented, even if it is only informal and unchecked.

Regarding suitability for object or class located crosscutting, note that the aspect code is injected into the main class, so in our example every bank account is syncronized, as required. However, every account is also monitored. Thus the aspect oriented programming approach is suitable for class located crosscutting but not for object located crosscutting. It would appear impossible for a single approach to handle both sorts of crosscutting, since object located crosscutting requires each object creation to specify which variety is needed, while class located crosscutting does not permit different varieties. Aspect oriented programming can in fact support object located crosscutting, but not without content coupling.

4.4.1 AspectJ and Metadata.

AspectJ (version 5) makes good use of the Metadata feature introduced in J2SE 5.0 [7]. Metadata allows programmers to add additional information to program elements (e.g. classes and methods) which can later be used to enhance the documentation of the class or even perform compile-time tasks. AspectJ uses metadata to simplify the definition of pointcuts by relying on the metadata tag rather than the name of the join point (e.g. class, method, member, etc.).

Using metadata, we can shift some of the pointcut definition to the main class. For example, the deposit() and withdraw() methods in the BankAccount class can be *annotated* or *tagged* with a common annotation which can be used later in the aspect. Listing 3 shows the BankAccount class with annotations and how can

Table 1. Difference between the four approaches

		Direct Modification	Subclassing	Decorator	AspectJ
Low Coupling [a]		n/a	✓	✓	✗
High Cohesion		✗	✓	✓	✓
CC Visibility	in M	✓	✗	✗	✗
	in M'	n/a	✓	✓	n/a
	in Client	✓	✓	✓	✗
	in Aspect	n/a	n/a	n/a	✓
Reusability	M	✗	✓	✓	✓
	CC	✗	✗	✗	✗
Crosscutting Type [b]		S,C,M	C,O	O	S,C
Scalability/ Sustanability		✗	✗	✓	✗

[a] M: Main logic; CC: Crosscutting Concern; M': derivative of the main class e.g. subclass or decorator
[b] S: system-wide crosscutting; C: class crosscutting; O: object crosscutting; M: method crosscutting

these annotations be used in the aspect. This is an important step toward defining an interface between aspect and class. The first advantage of this approach is that even if we want to change the name of a certain method in the `BankAccount` class, we do not need to change the pointcut definition in the aspect (unless we change the arguments, return type, or modifiers). Furthermore, if we want to add a new `creditInterest` method, then we only need to tag it with the `@UpdatingMethods` annotation to ensure the monitoring logic continues to work correctly.

4.5 Discussion

In this section we have investigated four possible ways of supporting crosscutting concerns using object and aspect oriented languages. Table 1 summarizes the advantages and disadvantages of these approaches from the standpoint of program maintenance, focusing on coupling, cohesion, reusability of the original component as well as of the crosscutting concern, and visibility, as well as the sustainability and scalability of the approach. We also indicate for each which sorts of crosscutting it is suitable for.

None of the approaches we considered supports all of our requirements. Direct modification is simple, but does not produce reusable or cohesive components. Subclassing produces reusable main components, but is not scalable or sustainable. The decorator pattern is sustainable and scalable, and exhibits low coupling and high cohesion, however it is somewhat heavyweight, and is not suitable for class located crosscutting. Aspect Oriented Programming is fairly lightweight, however it introduces a high degree of coupling between main and crosscutting concerns, with the connection not visible to the main concern, and is not suitable for object located crosscutting. Only aspect oriented programming supports producing reusable crosscutting components using *abstract* aspects. None of the approaches supports both object and class located crosscutting very well. These two kinds of crosscutting are inevitably different, but it would be very useful to be able to write reusable crosscutting components that could be used both for object or class located crosscutting. For example, it would be useful to be able to reuse a monitoring component for object located crosscutting with the `BankAccount` class, but for class located crosscutting with some other class that must always be monitored.

5 Enhancing Maintenance of Crosscutting Concerns

In the previous section, we discussed some of approaches for implementing crosscutting concerns using the techniques available in OO and Aspect Oriented languages. We found that what makes maintaining crosscutting concerns difficult is one of the following:

- Scattering of the crosscutting logic (OOP);
- Tangling of the crosscutting logic (OOP); and
- Invisibility of the crosscutting logic (AOP).

The first two problems have been successfully solved by the AOP approach, however, invisibility is still a problem. Visibility is an obvious feature in most programming paradigms like the object oriented or functional paradigms because components usually communicate through message passing, function calling, or similar mechanisms. Hassoun and Constantinides [3] mentioned four types of visibilities in object oriented languages: 1) *attribute* visibility, 2) *parameter* visibility, 3) *locally declared* visibility, and 4) *global* visibility. Attribute visibility indicates that the concern is

visible in the target class as an attribute or field. Parameter visibility means that the concern is passed to the class as an argument to one of its constructors or methods. Locally declared visibility means that the concern is declared inside one of the methods in the target class. Finally, global visibility indicates that the concern is declared as a global entity that can be seen by other objects. Regardless of the visibility type, the main issue here is that we can determine the presence of different crosscutting concerns by looking at the code in the main module or when objects are created.

In aspect oriented languages, concerns can be added to a certain class without affecting its source code. Moreover, we don't need to change the way we declare our object. Thus, the crosscutting concern is totally invisible to the developer of the main class. This can be considered a good feature for system-wide crosscutting, but can also be confusing in many other cases. We believe that the invisibility problem is a direct consequence of the way classes and aspects are currently associated i.e. *class-directional* relationships. In this section we will see how we can improve the AOP approach by providing *aspect-directional* relationships and how this will affect the visibility and reusability of the crosscutting concerns.

McEachen and Alexander [11] identified several fault scenarios that may occur because of the *invisibility* of the aspect in the target class. One of these scenarios is the potential fault of inheriting from an *aspected* class i.e. a class that has an aspect woven into it. An ordinary Java compiler will not weave an aspect that has been woven into the parent class into the new subclass. As a result, there might be a behavior inconsistency in the main class and its subclasses.

The question is: how to separate crosscutting concerns from core concerns (as in AOP) without introducing other problems like high coupling and invisibility of the crosscutting concern?

5.1 Characteristics of a Better Solution

Before discussing any ideas we should clearly identify what we consider to be a good solution for implementing and maintaining crosscutting concerns. By looking at Table 1, we see that the decorator and AspectJ solutions provided most of the desired properties we mentioned in Sec 1: *robustness*, *sustainability*, *scalability*, *reusability* , *low coupling*, and *high cohesion scalability*. The Decorator pattern provides us with a solution that keeps the `BankAccount` class uncoupled with the crosscutting concerns and also make them visible to the client. Also, it allows multiple concerns to be added to the main object easily. However, it is not adaptable to changes in the `BankAccount` class. AspectJ leaves us a clean and cohesive `BankAccount` class, however, it creates tightly coupled aspects that are not visible in the `BankAccount` class. To summarize, what we consider to be a good solution for class crosscutting (e.g.logging and synchronization) is a solution that satisfies the following:

1. **Separation of Concerns:** separates main module from crosscutting modules, maintaining high cohesion and low coupling [4];
2. **Visibility:** makes the crosscutting logic visible in the main concern [3];
3. **Robustness:** is robust and adaptable to changes in both modules (i.e. main class and crosscutting module);
4. **Scalability:** allows easy addition of many concerns to the main module; and
5. **Reusability:** provides reusable main and crosscutting modules.

In the remainder of this section, we sketch our proposed modifications to aspect oriented programming that we believe will provide all these characteristics. These ideas are very rough, suggesting future research we propose to pursue.

5.2 Improving Visibility: Aspect-Directional Relations

As mentioned in section 3.1, current AOP systems uses either *open* or *class-directional* relations to associate classes with aspects. In both cases, classes do not have any information about the aspects that are connected to them. This is one of the main factors making AOP maintenance hard. So, what if we use other association types like aspect-directional? Does this contradict the separation of concerns principle?

In current AOP languages, the aspect includes pointcut declarations (which do the linking between the aspect and the target class) and advice bodies (which represent the crosscutting concern code). The idea in the aspect-directional association is to shift the pointcut from the aspect to the class side. Doing so, we can see the various aspects affecting our class by looking at its source code (hence, improving visibility) and we still have the crosscutting concern code encapsulated in the aspect (avoiding the tangling and scattering problems).

5.3 Improving Reusability: Parametric Aspects

Using aspect-directional associations ensures the visibility of aspects in the classes they modify. It also avoids the problem of content (extreme) coupling between class and aspect, since even if pointcuts refer to private parts of the

class, they appear in the class itself. However, to reduce the coupling to the lowest level, a proper interface between class and aspect must exist.

We propose allowing aspects to accept parameters to specify any sort of information needed to distinguish different uses of an aspect. Such information might include pointcuts, types, data member names, method names and signatures. We believe this would be more flexible, or at least more convenient, than using aspect inheritance to create a separate specialized aspect for each use of a reusable aspect.

The structure of the new solution is as follows:

Aspect:

- contains the crosscutting logic only, i.e. no information about the target class;
- a constructor providing a proper interface for the class; the constructor can have parameters such as the name of the target class and the pointcut.

Class/ Client Application:

- pointcuts are defined inside the class or the client application rather than the aspect;
- aspects can be initialized using their constructors by passing the name of the target class and the appropriate pointcuts.

This solution does not contradict the separation of concerns principle in the sense that the code of the main logic is not mixed with the crosscutting logic. Moreover, this approach gives the class control over what aspects to include. By eliminating the pointcuts from the aspect, we improve the chances of aspects reusability. Most importantly, we ensure the visibility of the aspects affecting a class in the class itself.

Applying this solution to the Bank Account problem would result in reusable aspects containing the various crosscutting concerns. Now it is the responsibility of the `BankAccount` classdeveloper to or not to use the available aspects.

5.4 Supporting Object Located Crosscutting

Although the decorator pattern supports object located crosscutting rather well, it would be preferable to have a solution that was compatible with the solution for class located crosscutting. That is, it would be useful to be able to reuse the same aspect for either object or class located crosscutting, depending on requirements, or to switch the same use from one to the other without needing to modify the crosscutting component. This would seem perfectly possible, since once an object is created, whether a concern is class or object located is irrelevant.

Therefore we propose that the syntax for object creation to be extended to allow the specification of a number of aspects that should be woven into this particular object. We envision two possible approaches. The decorator-like approach would assign the created object the same type as the main class. This is simple, but would not permit the aspect to introduce new public members to the class. The multiple inheritance-like approach would consider the type of the new object to be the combination of the main type and the specified aspects. This is more complicated, as it requires extension of the type system, but it does permit aspects to introduce new methods. Either of these approaches would permit aspects to be reused compatibly for both object and class located crosscutting. Furthermore, both avoid the strange inheritance hierarchy needed for the decorator pattern.

5.5 Limitations

Our approach does not support System-wide or method located crosscutting. We would argue that the maintainability problems they create due to invisible tight coupling make supporting these sorts of crosscutting undesirable. Where system-wide crosscutting is only needed during program development, a powerful interactive development environment would seem a better solution. In other cases, standard software engineering techniques such as refactoring, or using current aspect oriented programming languages such as AspectJ, may be the best solution.

6 Related Work

Creating modular, maintainable, and reusable aspects has been a concern for many researchers. Parametric introductions have been proposed by [6] to provide a modular way to inject new members to target classes. This work provides a better solution to develop reusable aspects for cases where modular techniques in AspectJ (abstract aspects) are inadequate. However, the main focus was on parameterizing introductions rather than the whole aspect. This might reduce the chance of reusing aspects although their introductions are parameterized. Moreover, this solution does not help in resolving the visibility issue discussed before. In other word, target classes are still unaware of aspects that affect them.

In another study, Jordi Alvarez [2] has proposed parametric aspects as an extension to the AOP languages to increase the chance of aspect reuse. Focusing on providing a solution for some of the popular design patterns (e.g. factory design pattern), the proposed language extension added a set of new constructs to define roles of the different design

patterns. Again, this solution did not pay attension to the visibility issue of aspects.

Another line of research involves minimizing or eliminating the bad effect of aspects on main classes. Open Modules [1] and *Pure aspects* [16] have been proposed to restrict aspects from modifying base code. The idea in these proposals is to define an interface containing the set of joinpoints that can be altered by the aspects. Although this idea resolves some of the problems produced by the invisibility problem, it does not resolve the invisibility problem itself.

7 Conclusion

In this paper, we have investigated the difficulty of maintaining programs in the face of crosscutting concerns. In object oriented languages, the scattering and tangling problems resulted in these difficulties at both the implementation and maintenance levels. Aspect oriented languages, on the other hand, avoid these problems, but create high levels of invisible coupling that also impede maintenance.

To overcome this problem, we suggested using aspect-directional connection between classes and aspects to improve the visibility of crosscutting concern. Also, in order to enhance reusability of aspects, we suggested allowing them to be initialized with parameters (e.g. name of target class). This allows aspects to abstract out more than can be abstracted in abstract aspects in AspectJ. Finally, we proposed allowing aspects to be specified where objects are created, adding support for object located crosscutting. Taken together, these proposals avoid the maintenance problems we have identified in both object oriented and aspect oriented programming.

References

[1] Jonathan Aldrich. Open modules: Modular reasoning in aspect-oriented programming.

[2] Jordi Alvarez Canal. Parametric aspects: A proposal. In Walter Cazzola, Shigeru Chiba, and Gunter Saake, editors, *ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAMSE)*, pages 91–100, June 2004.

[3] Constantinos A. Constantinides and Youssef Hassoun. Considerations on component visibility and code reusability in aspectJ. February 14 2003.

[4] E.W. Dijkstra. On the role of scientific thought. Unpublished essay, August 1974.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns — Elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, Wokingham-Reading-Menlo Park-New York-Don Mills-Amsterdam-Bonn-Sydney-Singapore-Tokyo-Madrid-San Juan-Milan-Paris-Mexico City-Seoul-Taipei, 1995.

[6] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 80–89, New York, NY, USA, 2003. ACM Press.

[7] New features and enhancements in j2se 5.0. Available from http://java.sun.com/j2se/1.5.0/ docs/relnotes/features.html, 2005.

[8] Mik Kersten and Gail C. Murphy. Atlas: a case study in building a Web-based learning environment using aspect-oriented programming. *ACM SIGPLAN Notices*, 34(10):340–352, 1999.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[10] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[11] Nathan McEachen and Roger T. Alexander. Distributing classes with woven concerns: an exploration of potential fault scenarios. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 192–200, New York, NY, USA, 2005. ACM Press.

[12] G.J. Myers. *Composite/Structured Design*. Van Nostrand Reinhold, New York, 1978.

[13] D.L. Parnas. Information distribution aspects of design methodology. In *Procedings of the IFIP Congress*, pages 339–344, Ljubljana, Yugoslavia, 1971.

[14] W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[15] Rainer Unland and Stefan Hanenberg. A proposal for classifying tangled code, January 30 2002.

[16] Elcin Recebli Wolfson. Pure aspects, August 2005.