Systems Evolution and Software Reuse in Object-Oriented Programming and Aspect-Oriented Programming

Adam Przybyłek

University of Gdansk
Piaskowa 9, 81-824 Sopot, Poland
adam@univ.gda.pl

Abstract. Every new programming technique makes claims that software engineers want to hear. Such is the case with aspect-oriented programming (AOP). This paper describes a quasi-controlled experiment which compares the evolution of two functionally equivalent programs, developed in two different paradigms. The aim of the study is to explore the claims that software developed with aspect-oriented languages is easier to maintain and reuse than this developed with object-oriented languages. We have found no evidence to support these claims.

Keywords: AOP, maintainability, reusability, separation of concerns.

1 Introduction

Object-oriented programming (OOP) aims to support software maintenance and reuse by introducing concepts like abstraction, encapsulation, aggregation, inheritance and polymorphism. However, years of experience have revealed that this support is not enough. Whenever a crosscutting concern needs to be changed, a developer has to make a lot of effort to localize the code that implements it. This may possibly require him to inspect many different modules, since the code may be scattered across several of them.

An essential problem with traditional programming paradigms is the tyranny of the dominant decomposition [36]. No matter how well a software system is decomposed into modules, there will always be concerns (typically non-functional ones) whose code cuts across the chosen decomposition [25]. The implementation of these crosscutting concerns will spread across different modules, which has a negative impact on maintainability and reusability.

The need to achieve better separation of concerns (SoC) gave rise to aspect-oriented programming (AOP) [19]. The idea behind AOP was to implement secondary concerns as separate modules, called aspects. AOP has been proven to be effective in lexically separating different concerns of the system [33]. However, the influence of AOP on other quality attributes is still unclear.

J. Bishop and A. Vallecillo (Eds.): TOOLS 2011, LNCS 6705, pp. 163–178, 2011.

On the one hand, replacing code that is scattered across many modules by a single aspect can potentially reduce the number of changes during maintenance [27]. In addition, modules may be easier to reuse, since they implement single concerns and do not contain tangled code.

On the other hand, constructs such as pointcuts and advices can make the ripple effects in aspect-oriented (AO) systems far more difficult to control than in OO systems. Current AO languages rely on referencing structural properties of the program such as naming conventions and package structure. These structural properties are used by pointcuts to define intended conceptual properties about the program. The obliviousness property of AspectJ implies that the underlying system does not have to prepare any hooks, or in any way depend on the intention to apply an aspect over it [18]. Thus, maintenance changes that conflict with the assumptions made by pointcuts introduce defects [27]. This phenomenon is called the pointcut fragility problem [20]. It occurs when a pointcut unintentionally captures or misses a given join point as a consequence of seemingly safe modifications to the base code [20], [27]. Kästner et al. [17] reported such silent changes during AO refactoring.

Obliviousness also leads to programs that are unnecessarily hard to understand [14]. Since not all the dependencies between the modules in AO systems are explicit, an AO maintainer has to perform more effort to get a mental model of the source code [35]. Creating a good mental model is crucial to understand the structure of a system before attempting to modify it [24]. Studies of software maintainers have shown that 30% to 50% of their time is spent in the process of understanding the code that they have to maintain [11], [34], [13].

Moreover, incremental modifications and code reuse are not directly supported for the new language features of AspectJ [15]. In particular, concrete aspects cannot be extended, advice cannot be overridden, and concrete pointcuts cannot be overridden. Hanenberg & Unland proposed four rules of thumb [15], which allow one to build reusable and incrementally modifiable aspects. However, enormous complexity is the price that has to be paid for it.

2 Motivations

Many unsupported claims have been made about AOP. Here are a few examples:

- AOP “can be seen as a way to overcome many of the problems related to software evolution” [25].
- AOP “produces code that is simpler and more maintainable, as well as increasing the flexibility, extensibility and re-usability of the separated concerns” [3].
- AO software “is supposed to be easy to maintain, reuse, and evolution” [41].
- AOP leads to “the production of software systems that are easier to maintain and reuse” [33].
- AOP “increases understandability and eases the maintenance burden, because modules tend to be more cohesive and less coupled” [22].

It is commonly acknowledged that designs with low coupling and high cohesion lead to software that is both, more reusable and more maintainable. Table 1 enumerates work that documented these relationships. Since in our previous study [30] we did not

find empirical evidence that AOP increases cohesion, but we found that AOP increases coupling, we doubt the claims about the positive impact of AOP on reusability and evolvability. However, we do not intend to reject these claims as invalid with indirect evidences. Therefore, we conduct a quasi-experiment. We assume that the reader has a basic knowledge of AspectJ programming.

Table 1. Impact of coupling and cohesion on reusability and maintainability

	reusability	**maintainability**
coupling	[5], [16]	[5], [16], [6], [8], [23]
cohesion	[5], [4]	[5], [29]

3 Measurement System

In order to identify the metrics to be collected during the study, we used the G-Q-M (Goal-Question-Metric) approach [2]. G-Q-M defines a measurement system on three levels (Fig. 1) starting with a goal. The goal is refined in questions that break down the issue into quantifiable components. Each question is associated with metrics that, when measured, will provide information to answer the question.

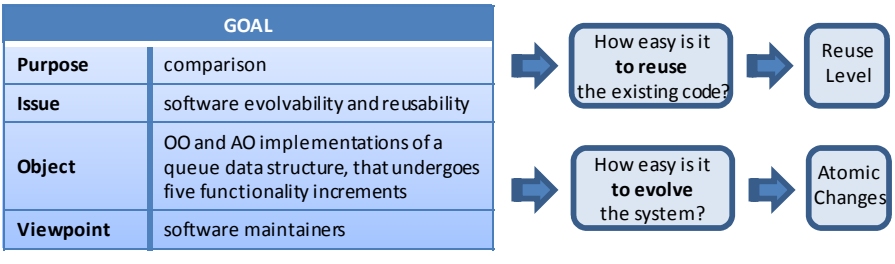


Fig. 1. GQM diagram of the study

Our goal is to compare AO and OO systems with respect to software evolvability and reusability from the viewpoint of the developer. Evolvability and reusability are quality characteristics that we cannot measure directly. Instead, we can perform an experiment that involves maintenance tasks and then we can measure how much effort is required to evolve the system and how much of the existing code can be reused in the consecutive release.

The amount of reuse is usually measured by comparing the number of reused "items" with the total number of "items" [12], where items depend on the granularity chosen, e.g. lines of code (LOC), function, or class. Since we are going to measure code reuse, we have chosen the granularity of LOC, yet we count only these reused lines that are part of the modules reused by applying the composition mechanisms of the underlying programming language. Thus, our reuse level metric is defined as: LOC_of_reused_modules / total_LOC_in_system.

The evolution metric we use is based on previous studies performed by Zhang et al. [40] and Ryder & Tip [32]. In their work, the difficulty of evolvability is defined in terms of atomic changes to the modules in a program. At the core of this approach is the ability to transform source code edits into a set of atomic changes, which captures the semantic differences between two releases of a program. Zhang et al. [40] presented a catalog of atomic changes for AspectJ programs. For the purpose of our study, we have slightly modified their catalog. Firstly, we consider deleting a non-empty element as an atomic change. Secondly, we use the term "module" as a generalization of class, interface, and aspect. Our list of atomic changes is follows: add an empty module, delete a module, add a field, delete a field, add an empty method, delete a method, change body of method, add an empty advice, delete an advice, change an advice body, add a new pointcut, change a pointcut body, delete a pointcut, introduce a new field, delete an introduced field, change an introduced field initializer, introduce a new method, delete an introduced method, change an introduced method body, add a hierarchy declaration, delete a hierarchy declaration, add an aspect precedence, delete an aspect precedence, add a soften exception declaration, delete a soften exception declaration.

4 Empirical Evaluation

The difficulty of performing evolvability and reusability evaluation in AOP is that there are not yet industrial maintenance reports for AO software projects available for analyses. Thus, we have to simulate maintenance tasks in a laboratory experiment. We compare OOP with AOP on a classical producer-consumer problem. In a producer-consumer dilemma two processes (or threads), one known as the "producer" and the other called the "consumer", run concurrently and share a fixed-size buffer. The producer generates items and places them in the buffer. The consumer removes items from the buffer and consumes them. However, the producer must not place an item into the buffer if the buffer is full, and the consumer cannot retrieve an item from the buffer if the buffer is empty. Nor may the two processes access the buffer at the same time to avoid race conditions. If the consumer needs to consume an item that the producer has not yet produced, then the consumer must wait until it is notified that the item has been produced. If the buffer is full, the producer will need to wait until the consumer consumes any item.

We assume to have an implementation of a cyclic queue as shown in Fig. 2a. The put(..) method stores one object in the queue and get() removes the oldest one. The nextToRemove attribute indicates the location of the oldest object. The location of a new object can be computed using nextToRemove, numItems (number of items) and buf.length (queue capacity). We also have an implementation of a producer and a consumer.

The experiment encompasses five maintenance scenarios which deal with the implementation of a new requirement. We have selected them because they naturally involve the modification of modules implementing several concerns.

4.1 Adding a Synchronization Concern

To use Queue in a consumer-producer system an adaptation to a concurrent environment is required. A thread has to be blocked when it tries to put an element into a full buffer or when it tries to get an element from an empty queue. In addition, both put(..) and get() methods have to be executed in mutual exclusion. Thus, they have to be wrapped within synchronization code when using Java (Fig. 2b). Since the code supporting the secondary concern may throw an exception, there is also a technical concern of error handling. The core concern here is associated with adding and removing item from the buffer. The presented implementation tangles the code responsible for the core functionality with the code responsible for handling errors and for cooperating synchronization. Moreover, the implementation of both secondary concerns are scattered through the accessors methods. As a result, the put(Object) and get() methods contain similar fragments of code.

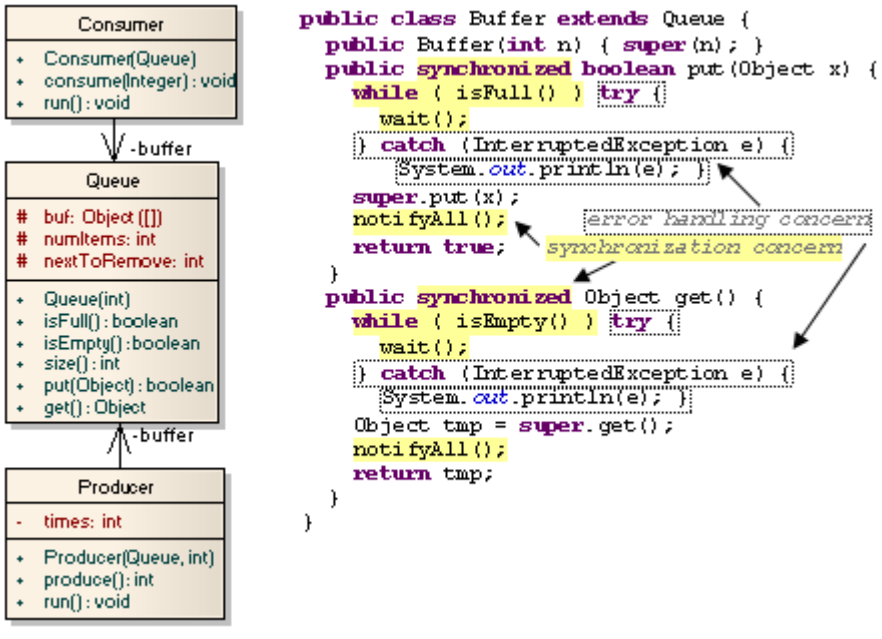


Fig. 2. a) An initial implementation; b) A new class for Stage I

Lexical separation of concerns can be achieved by using AO constructs (Fig. 3). The secondary concerns are implemented in ErrorHandler and SynchronizedQueue. SynchronizedQueue::waiting() is a hook method to introduce an explicit extension point. This joinpoint is used by ErrorHandler to wrap wait() invocation. Despite of lexical separation, SynchronizedQueue is explicitly tied to the Queue class, and so cannot be reused in other contexts. Moreover, Queue is oblivious of

```
public aspect ErrorsHandler {
  protected pointcut waiting():execution(void SynchronizedQueue.waiting());
  void around(): waiting() {
   try{proceed();} catch (InterruptedException e){System.out.println(e);}
  }
  declare soft:InterruptedException:waiting();
}
public aspect SynchronizedQueue pertarget( instantiation() ) {
  protected pointcut instantiation(): target(Queue);
  protected pointcut call_get(): execution( Object Queue.get() );
  protected pointcut call_put(Object x):
    execution( boolean Queue.put(Object) ) && args(x);
  protected void waiting() { wait(); }
  Object around(Queue q): call_get() && target(q){
    synchronized(this) {
      while( q.isEmpty() ) waiting();
      Object tmp = proceed(q);
      notifyAll(); return tmp;
    }
  }
  boolean around(Queue q, Object x): call_put(x) && target(q) {
    synchronized(this) {
      while ( q.isFull() ) waiting();
      proceed(q,x);
      notifyAll(); return true;
    }
  }
}
```

Fig. 3. New aspects for Stage I

SynchronizedQueue. This makes it difficult to know what changes to Queue will lead to undesired behavior.

4.2 Adding a Timestamp Concern

After implementing the buffer a new requirement has occurred – the buffer has to save current time associated with each stored item. Whenever an item is removed, the time how long it was stored should be printed to standard output. A Java programmer may use inheritance and composition as reuse techniques (Fig. 4a). The problem is that three different concerns are tangled within put/get and so these concerns cannot be composed separately. It means that e.g. if a programmer wants a queue with timing he cannot reuse the timing concern from TimeBuffer; he has to reimplement the timing concern in a new class that extends Queue. A slightly better solution seems to be using AOP and implementing the timing as an aspect (Fig. 4b).

Unless explicitly prevented, an aspect can apply to itself and can therefore change its own behavior. To avoid such situations, the instantiation pointcut is guarded by !cflow(within(Timing)). Moreover, the instantiation pointcut in SynchronizedQueue has to be updated. It must be the same as in Timing. This can be done only destructively, because AspectJ does not allow for extending concrete aspects.

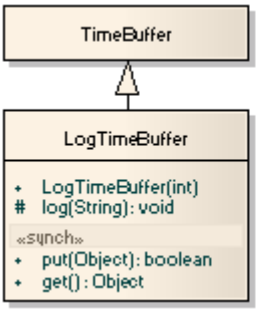
```
public class TimeBuffer extends Buffer {
  protected Queue delegateDates;
  public TimeBuffer(int capacity) {
    super(capacity);
    delegateDates = new Queue(capacity);
  }
  public synchronized boolean put(Object x) {
    super.put(x);
    delegateDates.put(new Long(System.currentTimeMillis()));
    return true;
  }
  public synchronized Object get() {
    Object tmp = super.get();
    Long date = (Long) delegateDates.get();
    long curr = System.currentTimeMillis();
    System.out.println(curr - date.longValue());
    return tmp;
  }
}
public privileged aspect Timing pertarget( instant() ) {
  protected Queue delegateDates;
  protected pointcut instant():target(Queue) &&! cflow(within(Timing));
  protected pointcut init(Queue q):execution(Queue.new(..)) && target(q);
  protected pointcut execution_get():execution( Object Queue.get() );
  protected pointcut execution_put():execution(boolean Queue.put(Object));
  after(Queue q): init(q) { delegateDates = new Queue(q.buf.length); }
  after(): execution_get() {
    Long date = (Long) delegateDates.get();
    System.out.println(System.currentTimeMillis() - date.longValue());
  }
  after(): execution_put() {
    delegateDates.put(new Long(System.currentTimeMillis())); }
}
```

Fig. 4. a) The TimeBuffer class; b) The Timing aspect

4.3 Adding a Logging Concern

The buffer has to log its size after each transaction. The OO mechanisms like inheritance and overridden allow a programmer for reusing TimeBuffer (Fig. 5a). The only problem is that four concerns are tangled within the LogTimeBuffer class. A module that addresses one concern can generally be used in more contexts than one that combines multiple concerns.

The AO solution is also noninvasive and it reuses the modules from the earlier stages. It just requires defining a new aspect (Fig. 5b). When advice declarations made in different aspects apply to the same join point, then by default the order of their execution is undefined. Thus, the declare precedence statement is used to force timing to happen before logging. The bufferChange pointcut enumerates, by their exact signature, all the methods that need to captured. Such pointcut definition is particularly fragile to accidental join point misses. An evolution of the buffer will require revising the pointcut definition to explicitly add all new accessor methods to it.



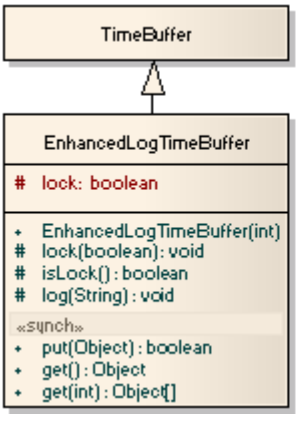

```
public aspect Logging {
  declare precedence : Logging, Timing;
  pointcut bufferChange(): (
    execution( * Queue.get() ) ||
    execution( * Queue.put(..)
    ) && !cflow(within(Timing));

  after(Queue q): bufferChange() && target(q) {
    System.out.println(
      "buffer size: "+q.size() );
  }
}
```

Fig. 5. a) A new class for Stage III; b) The Logging aspect

4.4 Adding a New Getter

The buffer has to provide a method to get "N" next items. There is no efficient solution of this problem neither using Java nor AspectJ. In both cases, the condition for waiting on an item has to be reinforced by a lock flag. A lock flag is set when some thread initiates the "get N" transaction by getting the first item. The flag is unset after getting the last item. In Java (Fig. 6a), not only does the synchronization concern has to be reimplemented but also logging. The reason is that in LogTimeBuffer logging is tangled together with synchronization, so it cannot be reused separately. The duplicate implementation might be a nightmare for maintenance.




```
public aspect Logging {
  pointcut bufferChange():
     !cflow(within(Timing)) &&
     !cflow(withincode(* Queue.get(int) )) &&
     ( execution( * Queue.get(..) ) ||
       execution( * Queue.put(..) ) );
  //...
}
public aspect ErrorsHandler {
  protected pointcut waiting():
     execution (
       void EnhancedSynchronizedQueue.waiting() )
     || execution( void Queue.waiting() );
  //...
}
```

Fig. 6. a) A new class for Stage IV; b) Modifications in the pointcuts

In AspectJ, although synchronization is implemented in a separate module, it also cannot be reused in any way because an aspect cannot extend another concrete aspect. Thus, all code corresponding to the synchronization concerns has to be reimplemented (Fig. 7). A new method to get N items and locking mechanism are introduced to Queue by means of inter-type declaration.

```
public aspect EnhancedSynchronizedQueue pertarget( instant() ) {
  private boolean Queue.lock = false;
  public void Queue.lock(boolean b) { lock = b; }
  public boolean Queue.isLock() { return lock; }
  public synchronized Object[] Queue.get(int n) {
    while ( isEmpty()||isLock() ) waiting();
    lock(true);
    Object[] tmp = new Object[n];
    for(int i=0; i<n; i++) {
      while ( isEmpty() ) waiting();
      tmp[i] = get();
    }
    lock(false);
    return tmp;
  }
  private void Queue.waiting() { wait(); }
  protected void waiting() { wait(); }
  protected pointcut instant():target(Queue)&&!cflow(within(Timing));
  protected pointcut call_get():
    !cflow(withincode(* Queue.get(int) ))&& call(Object Queue.get());
  Object around(Queue q):call_get() && target(q) {
    synchronized(this) {
      while(q.isEmpty()||q.isLock()) waiting();
      Object tmp=proceed(q);
      notifyAll(); return tmp;
    }
  }
  declare precedence :
    EnhancedSynchronizedQueue, Logging, Timing;
  //...
}
```

Fig. 7. A new aspect for Stage IV

In addition, destructive changes in the Logging::bufferChange() pointcut are required (Fig. 6b). Otherwise logs would be reported n times in response to the get(int n) method, instead of just once after completing the transaction. This is due to that get(int n) uses get() for retrieving every single item from the buffer. Furthermore, the ErrorsHandler::waiting() pointcut also needs adjusting to the new decomposition.

4.5 Removing Logging and Timestamp

A programmer needs the enhanced buffer from Stage IV, but without the logging and timing concerns. In Java, he once again has to reimplement the get(int) method and much of the synchronization concerns. All to do in the AO version is to remove Logging and Timing from the compilation list.

5 Lessons Learned

In an AO system, one cannot tell whether an extension to the base code is safe simply by examining the base program in isolation. All aspects referring to the base program

need to be examined as well. In addition, when writing a pointcut definition a programmer needs global knowledge about the structure of the application. E.g. when implementing the Timing aspect, a programmer has to know that the current implementation of the synchronization concern affects each Queue structure, while the timing concern requires a non-blocking Queue.

Moreover, when a system includes multiple aspects, they can begin to affect each other. At Stage C, we have had to explicitly exclude logging the state of the queue that is used by the Timing aspect. Furthermore, we have observed the problem of managing interactions between aspects that are being composed. When advice declarations made in different aspects affect the same join point, it is important to consider the order in which they execute. Indeed, a wrong execution order can break the program. In our experiment, we have used precedence declarations to force timing to happen before logging and to force both of them to happen within the synchronization block.

In most cases, aspects cannot be made generic, because pointcuts as well as advices encompass information specific to a particular use, such as the classes involved, in the concrete aspect. As a result, aspects are highly dependent on other modules and their reusability is decreased. E.g. at Stage I, the need to explicitly specify the Queue class and the two synchronization conditions means that no part of the SynchronizedQueue aspect can be made generic. In addition, we have confirmed that the reusability of aspects is also hampered in cases where "join points seem to dynamically jump around", depending on the context certain code is called from [3]. Moreover, the variety of pointcut designators makes pointcut expressions cumbersome (see EnhancedSynchronizedQueue::call_get()).

Some advocates of AOP believe that appropriate tools can deal with the problems of AOP we encountered. We think that they should reject AOP at all, since some research [31] "shows" that OOP with a tool support solves the problem of crosscutting concerns:)

6 Empirical Results

Table 2 presents the number of Atomic Changes and Reuse Level for both releases for every stage. The measures were collected manually. Lower values are better for Atomic Changes but worse for Reuse Level. AOP manifests superiority at Stage III and V, while OOP in the rest of the cases. At Stage III we have implemented a logging concern which is one of the flagship examples of AOP usage. At this Stage, the OO version requires significantly more atomic changes and new lines of code than its AO counterpart. At Stage V, the maintenance tasks are focused on detaching some concerns instead of implementing new ones. The AO solution has turned out to be more pluggable.

Table 2. Number of Atomic Changes and Reuse Level per stage

Stage	Atomic Changes		Reuse Level	
	OOP	AOP	OOP	AOP
I. Adding a synchronization concern	7	19	0,71	0,66
II. Adding a timestamp concern	8	19	0,85	0,67
III. Adding a logging concern	9	6	0,88	0,95
IV. Adding a new getter	9	16	0,73	0,58
V. Removing logging and timestamp	5	3	0,74	1,00

7 Threats to Validity

7.1 Construct Validity

Construction threats lie in the way we define our metrics. Evolvability and reusability like other quality factors are difficult to measure. Our dependent variables are based on previous studies performed by Zhang et al. [40], Ryder & Tip [32] and Frakes [12]. It is possible that other metrics will be better fitted for the purpose of our study.

7.2 Internal Validity

Internal validity of our experiment concerns the question whether the effects were caused only by the programming paradigm involved, or by other factors. The experiment has been carried out by the author during his research for the achievement of a Doctor of Philosophy Degree. As the author does not have any interest in favour of one approach or the other, we do not expect it to be a large threat. Nevertheless, other programmers could have chosen the different strategies for implementing secondary concerns.

7.3 External Validity

Synchronization, logging, and timing present the typical characteristics of crosscutting concerns and as such they are likely to be generalizable to other concerns. Unfortunately, the limited number of maintenance tasks and size of the program make impossible the generalization of our results. However, the academic setting allows us to present the whole programs in detail and to put forward some advantages and limitations of AOP.

8 Related Work

Coady & Kiczales [9] compared the evolution of two versions (C and AspectC) of four crosscutting concerns in FreeBSD. They refactored the implementations of the following concerns in v2 code: page daemon activation, prefetching for mapped files, quotas for disk usage, and tracing blocked processes in device drivers. These implementations were then rolled forward into their subsequent incarnations in v3 and

v4 respectively. In each case they found that, with tool support, the AO implementation better facilitated independent development and localized change. In three cases, configuration changes mapped directly to modifications to pointcuts and makefile options. In one case, redundancy was significantly reduced. Finally, in one case, the implementation of a system-extension aligned with an aspect was itself better modularized.

Bartsch & Harrison conducted an experiment [1] in which 11 students were asked to carry out maintenance tasks on one of two versions (Java and AspectJ) of an online shopping system. The results did seem to suggest a slight advantage for the subjects using the OO version since in general it took the subjects less time to perform maintenance tasks and it averagely required less line of code to implement a new requirement. However, the results did not show a statistically significant influence of AOP at the 5% level.

Sant'Anna et al. [33] conducted a quasi-controlled experiment to compare the use of OOP and AOP to implement Portalware (about 60 modules and over 1 KLOC). Portalware is a multi-agent system (MAS) that supports the development and management of Internet portals. The experiment team (3 PhD candidates and 1 M.Sc. student) developed two versions of the Portalware system: an AO version and an OO version. Next, the same team simulated seven maintenance/reuse scenarios that are recurrent in large-scale MAS. For each scenario, the difficulty of maintainability and reusability was defined in terms of structural changes to the artifacts in the AO and OO systems. The total lines of code, that were added, changed, or copied to perform the maintenance tasks, equaled 540 for the OO approach and 482 for the AO approach.

Kulesza et al. [21] present a quantitative study that assesses the positive and negative effects of AOP on typical maintenance activities of a Web information system.They compared the AO and OO implementations of a same web-based information system, called HealthWatcher (HW). The main purpose of the HW system is to improve the quality of services provided by the healthcare institution, allowing citizens to register complaints regarding health issues, and the healthcare institution to investigate and take the required actions. In the maintenance phase of their study, they changed both OO and AO architectures of the HW system to address a set of 8 new use cases. The functionalities introduced by these new use cases represent typical operations encountered in the maintenance of information systems. Although they claim that the AO design has exhibited superior reusability through the changes, there is no empirical evidence to support this claim. The collected metrics show only that aspects contributed to: (1) the decrease in the lines of code, number of attributes, and cohesion; (2) the increase in the vocabulary size and lexical separation of crosscutting concerns. They also tried to evaluate coupling, but in our earlier study [30] we argued why their coupling metric is invalid. An additional interesting observation from Kulesza's study [21] is that more modules were needed to be modified in the AO version, because it requires changing both the classes along the layers to implement the use case functionality and the aspects implementing the crosscutting issues.

Munoz et al. [28] showed that aspects offer efficient mechanisms to implement crosscutting concerns, but that aspects can also introduce complex errors in case of evolution. To illustrate these issues, they implemented and then evolved a chat

application. They found that it is very hard to reason about the aspects impact on the final application.

Mortensen et al. [26] examined the benefits of refactoring three legacy applications developed by Hewlett-Packard. They followed the evolution of the applications across several revisions. The modifications needed to evolve these systems required changes to fewer software items in the refactored systems when compared to the original. The reduction of the average number of modules and files changed between revisions was 4% and 3% respectively.

Taveira et al. conducted two studies to check if AOP promotes greater reuse of exception handling code than a traditional, OO approach. In the first study [38], they assessed the suitability of AOP to reuse exception handling code within applications. They refactored three medium-size applications implemented originally in Java. Aspects were used to implement the exception handlers. Though AOP promoted a large amount of reuse of error handling code, the overall size of the refactored systems did not decrease due to the code overhead imposed by AspectJ. The number of handlers was sensibly lower in the refactored versions but the amount of error handling code was much higher. In the second study [37], they refactored seven medium-size systems to assess the extent to which AOP promotes inter-application reuse of exception handling code. They found out that reusing error handling across applications is not possible in most of the cases and requires some a priori planning. Only extremely simple handlers could be reused across applications.

The experiment closest to ours is the one conducted by Figueiredo et al. [10] in which they quantitatively and qualitatively assess the positive and negative impacts of AOP on a number of changes applied to MobileMedia. MobileMedia is a software product line for applications with about 3 KLOC that manipulate photo, music, and video on mobile devices. The original release was available in both AspectJ and Java (the Java versions use conditional compilation as the variability mechanism). Then, a group of five post-graduate students was responsible for implementing the successive evolution scenarios of MobileMedia. Each new release was created by modifying the previous release of the respective version. A total of seven change scenarios were incorporated. The scenarios comprised different types of changes involving mandatory, optional, and alternative features, as well as non-functional concerns. Figueiredo et al. found that AOP usually does not cope with the introduction of mandatory features. The AO solution generally introduced more modules and operations. A direct result of more modules and operations is the increase in LOC. Moreover, depending on the evolution scenario, AspectJ pointcuts were more fragile than conditional compilation. In order to compare their and our results, we have derived the simplest form of Reuse Level and Atomic Changes (Table 3) from their measures. Atomic Changes has been limited to counting operations only, while Reuse Level has been calculated as: number_of_reused_LOC / LOC. In general, the measures demonstrate that there is no winner with respect to Reuse Level. The AO solution is significantly better only at Stage VII. With regard to Atomic Changes, the OO implementations are superior for every release.

Table 3. Atomic changes and Reuse Level in MobileMedia

release ▶		II	III	IV	V	VI	VII	VIII
Reuse Level	OO	0,73	0,86	0,96	0,48	0,75	0,01	0,76
	AO	0,62	0,82	0,93	0,55	0,76	0,29	0,74
Atomic Change	OO	120	68	20	111	88	335	149
	AO	150	90	22	134	102	437	175

7 Summary

In 2001 the editors of January/February "MIT Technology Review" announced AOP as a standard in the commercial production of software in the next 15 years. Nowadays, ten years later, AOP is still not widely adopted. We believe that, the transfer of AOP to the mainstream of the software development depends on our ability to find its true benefits and to be aware of its potential pitfalls. In this paper, we have evolved a simple program in order to assess the potential of AOP to improve evolvability and reusability in the presence of crosscutting concerns. Although a definitely conclusion cannot be drawn from only the one discussed experiment, an important outcome has been achieved in that the advocates of AOP have to take a position on our results. By reviewing other research, we have shown that the claims presented in Section 2 are not backed up by any convincing evidence. In our study, the superiority of AOP has been observed only when detaching secondary concerns and when implementing logging, which is a flagship example of AOP usage. OOP has fared better in implementing secondary concerns in three out of four scenarios.

The experience gathered during the maintenance tasks points out that (1) understanding the intricate dependencies existing between the modules of an AO system is an arduous task; (2) aspects are holding too much information (the crosscutting logic and target module information) to fully take advantage of lexical SoC. Thus, it seems that the abstractions that AOP has provided to solve some of the evolution problems with traditional software, actually introduce a series of new evolution problems. This phenomenon has been called the evolution paradox of AOP [39], [28].

References

1. Bartsch, M., Harrison, R.: An exploratory study of the effect of aspect-oriented programming on maintainability. Software Quality Journal 16(1), 23–44 (2008)
2. Basili, V.R., Caldiera, G., Rombach, H.D.: Goal Question Metric Approach. In: Encyclopedia of Software Engineering, pp. 528–532. John Wiley & Sons, Inc., Chichester (1994)
3. Beltagui, F.: Features and Aspects: Exploring feature-oriented and aspect-oriented programming interactions. Technical Report No: COMP-003-2003, Computing Department, Lancaster University (2003)
4. Bieman, J.M., Kang, B.: Cohesion and reuse in an object-oriented system. SIGSOFT Softw. Eng. Notes 20(SI), 259–262 (1995)
5. Bowen, T.P., Post, J.V., Tai, J., Presson, P.E., Schmidt, R.L.: Software Quality Measurement for Distributed Systems. Guidebook for Software Quality Measurement. Technical Report RADC-TR-83-175 vol. 2 (July 1983)

6. Breivold, H.P., Crnkovic, I., Land, R., Larsson, S.: Using Dependency Model to Support Software Architecture Evolution. In: 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy (2008)
7. Brichau, J., De Meuter, W., De Volder, K.: Jumping Aspects. In: Workshop on Aspects and Dimensions of Concerns at ECOOP 2000, Sophia Antipolis and Cannes, France (2000)
8. Chaumun, M.A., Kabaili, H., Keller, R.K., Lustman, F., Saint-Denis, G.: Design Properties and Object-Oriented Software Changeability. In: 13th Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany (2000)
9. Coady, Y., Kiczales, G.: Back to the future: a retroactive study of aspect evolution in operating system code. In: 2nd Inter. Conf. on Aspect-oriented software development (AOSD 2003), Boston, Massachusetts (2003)
10. Figueiredo et al.: Evolving software product lines with aspects: An empirical study on design stability. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany (2008)
11. Fjeldstad, R., Hamlen, W.: Application program maintenance-report to to our respondents. In: Parikh, G., Zvegintzov, N. (eds.) Tutorial on Software Maintenance, pp. 13–27. IEEE Computer Soc. Press, Los Alamitos (1983)
12. Frakes, W.: Software Reuse as Industrial Experiment. American Programmer 6(9), 27–33 (1993)
13. Glass, R.L.: Facts and Fallacies of Software Engineering. Addison-Wesley, Reading (2002)
14. Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H.: Modular Software Design with Crosscutting Interfaces. IEEE Software 23(1), 51–60 (2006)
15. Hanenberg, S., Unland, R.: Using and Reusing Aspects in AspectJ. In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA 2001, Tampa Bay, Florida (2001)
16. Hitz, M., Montazeri, B.: Measuring Coupling and Cohesion in Object-Oriented Systems. In: 3rd International Symposium on Applied Corporate Computing, Monterrey, Mexico (1995)
17. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features using AspectJ. In: 11th International Conference of Software Product Line Conference (SPLC 2007), Kyoto, Japan (2007)
18. Katz, S.: Diagnosis of harmful aspects using regression verification. In: Workshop on Foundations of Aspect-Oriented Languages at AOSD 2004, Lancaster, UK (2004)
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Cristina Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
20. Koppen, C., Störzer, M.: PCDiff: Attacking the fragile pointcut problem. In: European Interactive Workshop on Aspects in Software, Berlin, Germany (2004)
21. Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., von Staa, A., Lucena, C.: Quantifying the effects of aspect-oriented programming: A maintenance study. In: 22nd IEEE International Conference on Software Maintenance (ICSM 2006), Dublin, Ireland (2006)
22. Lemos, O.A., Junqueira, D.C., Silva, M.A., Fortes, R.P., Stamey, J.: Using aspect-oriented PHP to implement crosscutting concerns in a collaborative web system. In: 24th Annual ACM International Conference on Design of Communication, Myrtle Beach, South Carolina (2006)
23. MacCormack, A., Rusnak, J., Baldwin, C.: The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. Harvard Business School Technology & Operations Mgt. Unit Research Paper, vol. 08-038 (2007)

24. Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., Gansner, E.R.: Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In: 6th International Workshop on Program Comprehension (IWPC 1998), Ischia, Italy (1998)
25. Mens, T., Mens, K., Tourwé, T.: Software Evolution and Aspect-Oriented Software Development, a cross-fertilisation. In: ERCIM special issue on Automated Software Engineering, Vienna, Austria (2004)
26. Mortensen, M., Ghosh, S., Bieman, J.: Aspect-Oriented Refactoring of Legacy Applications: An Evaluation. IEEE Trans. Software Engineering 99 (2010)
27. Mortensen, M.: Improving Software Maintainability through Aspectualization. PhD thesis, Department of Computer Science, Colorado State University, CO (2009)
28. Munoz, F., Baudry, B., Barais, O.: Improving maintenance in AOP through an interaction specification framework. In: IEEE Intl. Conf. on Software Maintenance, Beijing, China (2008)
29. Perepletchikov, M., Ryan, C., Frampton, K.: Cohesion Metrics for Predicting Maintainability of Service-Oriented Software. In: 7th International Conference on Quality Software (QSIC 2007), Portland, Oregon (2007)
30. Przybyłek, A.: Where the truth lies: AOP and its impact on software modularity. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 447–461. Springer, Heidelberg (2011)
31. Robillard, M.P., Weigand-Warr, F.: ConcernMapper: simple view-based separation of scattered concerns. In: Workshop on Eclipse technology eXchange at OOPSLA 2005, San Diego, CA (2005)
32. Ryder, B.G., Tip, F.: Change impact analysis for object-oriented programs. In: 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, Utah (2001)
33. Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: 17th Brazilian Symposium on Software Engineering (SEES 2003), Manaus, Brazil (2003)
34. Standish, T.: An essay on software reuse. IEEE Transactions on Software Engineering 10(5), 494–497 (1984)
35. Storey, M.D., Fracchia, F.D., Müller, H.A.: Cognitive design elements to support the construction of a mental model during software exploration. J. Syst. Softw. 44(3), 171–185 (1999)
36. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: multi-dimensional separation of concerns. In: 21st International Conference on Software Engineering (ICSE 2009), Los Angeles, California (1999)
37. Taveira, J., Oliveira, H., Castor, F., Soares, S.: On Inter-Application Reuse of Exception Handling Aspects. In: Workshop on Empirical Evaluation of Software Composition Techniques at AOSD 2010, Rennes, France (2010)
38. Taveira, J.C., et al.: Assessing Intra-Application Exception Handling Reuse with Aspects. In: 23rd Brazilian Symposium on Software Engineering (SBES 2009), Fortaleza, Brazil (2009)
39. Tourwé, T., Brichau, J., Gybels, K.: On the Existence of the AOSD-Evolution Paradox. In: AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, Massachusetts (2003)
40. Zhang, S., Gu, Z., Lin, Y., Zhao, J.: Change impact analysis for AspectJ programs. In: 24th IEEE International Conference on Software Maintenance, Beijing, China (2008)
41. Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. In: 10th International Software Metrics Symposium, Chicago, Illinois (2004)