The Case for Aspect Oriented Programming

André Restivo <arestivo@fe.up.pt>

Faculdade de Engenharia da Universidade do Porto

Abstract. Aspect Oriented Programming (AOP) deals with what are called cross-cutting concerns. AOP practitioners believe that single abstraction frameworks (like OOP) are not sufficiently powerful to separate cross-cutting concerns. They also state that the tangling of concerns is one of the major contributors to the complexity of large software applications. This paper will show some typical examples of concerns that are difficult to separate from the main core of the application. It will also exemplify how AOP can be used to describe each one of those concerns in a separate and natural form. Finally an overview of where AOP research is heading towards will be described.

1 Introduction

One of the major goals of Software Engineering has always been achieving a clear Separation of Concerns (SoC). By separating each one of the software components into well defined units, a number of advantages, like higher re-usability of coding, unit testing and many others, arise. The emergence of Object Oriented Programming (OOP), as opposed to Procedural Oriented Programming (POP), was clearly a step forward in the attempt to achieve an easier and more comprehensible programming environment.

However, practice shows that even following the most rigid OOP prerogatives, total SoC is not always possible to achieve, because some concerns seem to get inevitably tangled throughout the code [1]. This happens because OOP is a single abstraction paradigm, meaning that we are forced into organizing the code following a single perspective. The core concerns of the applications force other, less important, concerns (cross-cutting concerns) to get scattered throughout the application. It feels like trying to solve the *Rubik's Cube* by solving one face at a time[1].

Aspect Oriented Programming (AOP) advocates that it is possible to build upon the OOP paradigm creating a richer framework[2]. AOP is a new programming technique where *aspects* (cross-cutting concerns) are captured in their own units of modularity and then *weaved* together, and in this way creating the application through a process of composition [2]. Weaving can be done at many

[1] When solving the *Rubik's Cube* one normally solves all faces in parallel. If we try to solve one face at a time, solving the second face will destroy the first one.

[2] AOP is not an evolution of the OOP paradigm and can also be easily applied over the POP paradigm.

levels but at the moment it is mainly seen as an integrating part of the compiling process. At compilation time, aspects and units are weaved together creating a tangled version of the source code, then this code is compiled as usual generating a binary for the application (see Figure 1).

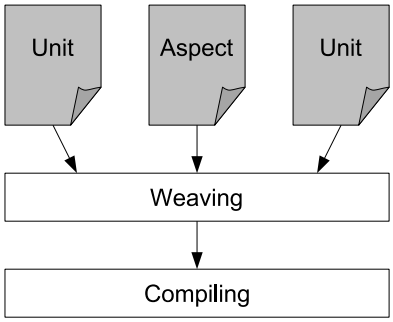

Fig. 1. The weaving process

1.1 Implementation

In order to be possible to specify where units and *aspects* should be weaved together a set of points in the code where weaving can occur must be specified. These are called *joinpoints*. Each AOP language can define its own set of *joinpoints*. For instance, *AspectJ*, an AOP language based in Java and the current de facto standard for AOP, defines the following *joinpoints: Method call* and *execution*, *Constructor call* and *execution*, *Read/write access to a field*, *Exception handler execution,* and *Object* and *class initialization execution*. In the remaining of this section we will see what other characteristics are implemented by AspectJ in order to transform the Java language into a AOP language.

As cross-cutting concerns normally are scattered throughout the application, most of the times we want to weave an *aspect* to several different *joinpoints*. *AspectJ* introduced the notion of *pointcut designators* which allow the filtering of *joinpoints*. A *pointcut designator* refers to several *joinpoints* and can be defined with the help of wildcards.

In AspectJ the element that defines how an application behaviour is altered in order to implement a certain *aspect* is the *advice*. Advices are code blocks that execute implicitly whenever a certain *joinpoint*, belonging to the pointcut associated to it, is reached (see Figure 2). AspectJ defines three types of advices: *after*, *before* and *around*. *After* advices run just after the joinpoint, *before* advices run before the actual joinpoint code is executed and *around* advices have control over the joinpoint execution.

Finally, the last element introduced by *AspectJ* is the *aspect*. An *aspect* combines *pointcuts* and *advices* composing a single cross-cutting unit.

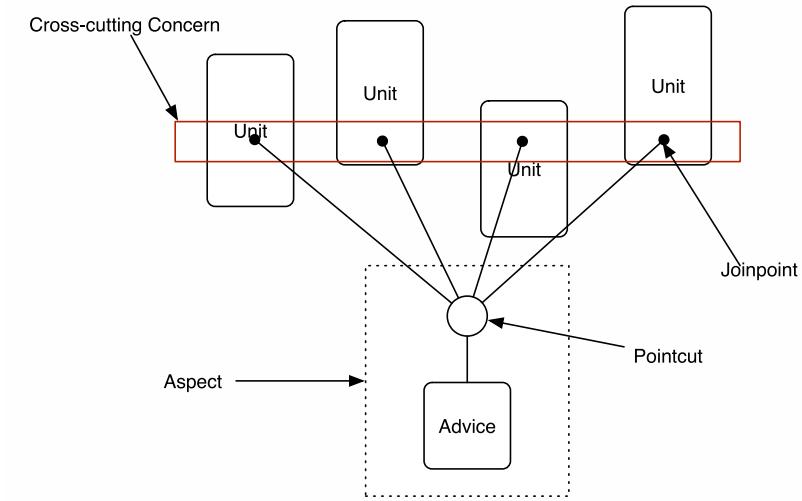

Fig. 2. Joinpoints and Pointcuts

1.2 Advantages

Literature identifies the following as the main advantages of using AOP [3]:

Explicitness *Cross-cutting concerns* are explicitly captured by aspects;
Reusability. It is possible through a single *aspect* to describe crosscutting concerns common to several components;
Modularity. Since *aspect*s are modular crosscutting units, AOP improves the overall modularity of an application;
Evolution. Evolution becomes easier since implementation changes of cross-cutting concerns occur locally within an *aspect* and save the need to adapt existing classes;
Stability. Special AOP language support makes it possible to express generic *aspects*, which will remain applicable throughout future class evolution;
Pluggability. Since *aspects* are modular, they can be easily plugged in and out of an application.

1.3 Structure

In the next Section some typical examples of cross-cutting concerns will be described as well as how they can be implemented using AOP. Some of the current issues being researched at the moment will be depicted in Section 3. And finally conclusions will be drawn in Section 4.

2 Examples of Cross-Cutting Concerns

The most typical example that can be found in AOP introduction papers and tutorials is the *Logging* concern example. This fact is not that peculiar as this example has all the major characteristics for a good application of AOP techniques. To follow the trend this section will start by describing how AOP can be used in that particular scenario and then other possible uses of AOP will be discussed.

2.1 Logging

Most software systems write information about its actions to log files. Several reasons make *logging* the perfect candidate for AOP.

Due to its own nature, logging code gets scattered throughout the application even if a good architecture is used. Besides that, logging is something that tends to change often.

Most logging is done using a specific logging package. The need of changing the logging package should be something that developers should anticipate, if the logging code is spread all over the application these changes could prove lengthy and complicated.

The most important factor off all is that logging, normally, is not a core concern of the application. This makes it even more important to separate this code from the code that really matters. For example, imagine a *class* named *HTTPConnection*, responsible for handling an HTTP Connection from a single client. If it is necessary to log each connection made by a client, and probably it will, possible places to put that code would be the *HTTPConnection* constructor or the class *connect* method. However that would subvert the objective of those functions. Class constructors only should know how to create a new instance of the class and should not be *concerned* with the logging *aspect* of the application, the same can be applied to the *connect* method.

In order to separate the *connection handling* code from the *logging* code, one only needs to *weave* the logging code into the *joinpoint* defined by the *HTTPConnection* connect method. Listing 1.1 shows how this can be done by defining an *aspect* named *HTTPConnectionLogging*. This aspect uses an *advice* that executes *before* any *call* to the connect method of the *HTTPConnection* class. This *advice* then simply uses the local variable *logger,* defined in this same *aspect,* to write a new line to the log file.

This will allow the HTTPConnection class to remain free from any code concerning the logging *aspect* of the application.

2.2 Security and Authentication

Security plays a major role in many software applications. It is also a good candidate for AOP as its scope crosses the entire application.

AOP could be used, for example, to control objects owners and permissions in a non pervasive way[4]. This could be accomplished by associating an *aspect*

Listing 1.1. Logging Aspect implemented with AspectJ

```
1 aspect HTTPConnectionLogging {
2     Logger logger = Logger.getDefaultLogger();
3
4     before() : call(void HTTPConnection.connect(IP clientIP,
          Date date))
5     {
6         logger.notice(date.toString()+": "+clientIP);
7     }
8 }
```

to each constructor of the objects whose access has to be restricted, this *aspect* would automatically assign ownership to the current system user.Another *aspect* would be associated to each restricted access method of those same objects, and would verify the ownership of the object (See Listing 1.2).

Another security concern that could be tackled with AOP would be encryption. One could develop an application without thinking about how data would be transmitted in a safe way and then add an *aspect*, around each message sending method, that would encrypt data before it is sent. The same could be done for each message receiving method. This would allow developers to focus in the core concerns of the application and would even allow changing the encryption algorithm very easily.

2.3 Caching

In many cases database access is the bottleneck of an application. One way of dealing with this issue is to cache data that is retrieved very often. Caching can also be seen as a cross-cutting concern as it spreads throughout the code and can be removed without any loss in functionality.

The implementation of data caching with AOP can be done at several levels. One could, for example, weave an *aspect* around each query to the database, weave an *aspect* around a generic query method, or even weave an *aspect* around each object retrieval method (See Listing1.3). These *aspects* would then be responsible for verifying if the object, or query, is cached and if not proceed with the operation.

2.4 Contract Enforcement

Design by Contract (DbC) was first introduced by B. Meyer in 1992[5]. DbC specifies that modules should have formalized obligations, described as sets of rules, regarding their interaction with other modules.

At first glance DbC seems another perfect candidate for AOP implementation. For example, one could create an *aspect* that would verify pre and post-conditions in a certain method (See Listing 1.4).

Listing 1.2. Authentication Aspect implemented with AspectJ

```
aspect OwnerManagement perthis(this(Article)){
        String owner;

        after(): execution(Article.new(..)){
                owner = Authentication.getUser() ;
        }
}

aspect Authorization(){
        pointcut restrictedAccess():
                execution(* Article.update(..)) ||
                execution(* Article.delete(..));

        void around() : restrictedAccess(){
                if(! OwnerManagement.aspectOf(thisJoinPoint.
                    getThis()).owner.equals(Authentication.
                    getUser()))
                System.out.println("Access Denied !") ;
                else proceed() ;
        }
}
```

Listing 1.3. Caching Aspect implemented with AspectJ

```
aspect Caching {
    CacheManager manager = CacheManager.getDefaultManager();

    around() : call(void Account.retrieveAccount(int accountId
        ))
    {
        if (manager.isCached("Account", accountId))
                return manager.getCachedObject("Account",
                    accountId);
        else
                proceed();
    }
}
```

Listing 1.4. Contract implemented with AspectJ

```
aspect MathContract {
        before : (double x): sqrt(x) {
                if ( x < 0 )
                        throw new IllegalArgumentException("
                            negative not allowed");
        }
}
```

However, the same B. Meyer argues *aspects*, although having some advantages like allowing easy contract reutilization, are not as powerful and safe to use as a proper contract oriented implementation [3]. This happens because contracts developed using AOP do not support inheritance based refinements. Meyer also claims that contracts are not cross-cutting concerns.

This debate about AOP applicability is an important one. As with all development techniques, applying AOP in the wrong scenarios could lead to an unnecessary increase of the system complexity. When thinking about using AOP one should analyze:

- if the concern being implemented is really a cross-cutting concern;
- if it is possible to implement the concern in a modular form using OOP without adding too much complexity;
- if the concern is a core concern of the application (normally secondary requirements are better candidates for AOP);
- if reusability and pluggability are important issues for this particular concern;
- if changes in the units this concern connects with will also imply changes in the concern itself or if the concern depends semantically on the units;

Only after this kind of reasoning one could expect to extract the benefits described in Section 1.2 from AOP.

2.5 Flavours

Several software developing companies have products that come in different *flavours*. Sometimes these different *flavours* have more or less functionalities (and different prices) and sometimes they are adapted to different types of customers (or even tailored to a specific customer). Either way having different versions of the same application has a tremendous impact in the software developing process as changes to the base version must be tested against each one of the different especial versions.

If the differences between these versions can be encapsulated as *aspects*, then by simply removing and adding *aspects* we could have different versions with different capabilities making the development process much simpler.

3 Future Developments

AOP is still a very immature paradigm and a lot of work still needs to be done to bring it into mainstream programming. It took about about 20 years for OOP to become the dominant programming methodology (from Simula 67 and Smalltalk to C++). AOP stands now like OOP did in the mid sixties. In this section some of the current research topics in AOP will be described.

3.1 Tools

The non-linearity of AOP makes debugging harder than in classical OOP programming. This happens because code being run is not the same that as been written by the programmer as the weaving process already has cross-cutting concerns tangled throughout the code. For example, stack traces will be more difficult to understand. AOP programmers need new tools and methods for debugging their applications. Besides that, unit-testing can not be applied to cross-cutting concerns in the same way it is applied to units developed with the OOP paradigm. So, some research into how unit-testing methods and applications can be adapted to the AOP world has to be done.

The integration of AspectJ into the Eclipse IDE[3] (AJDT) has been a big step forward in the AOP evangelizing process [6]. For example, with AJDT Visualizer, one can see how a cross-cutting concern is scattered throughout the code (See Figure 3). However further research must be done into what new tools AOP developers will need and how they can be integrated into existing IDEs.

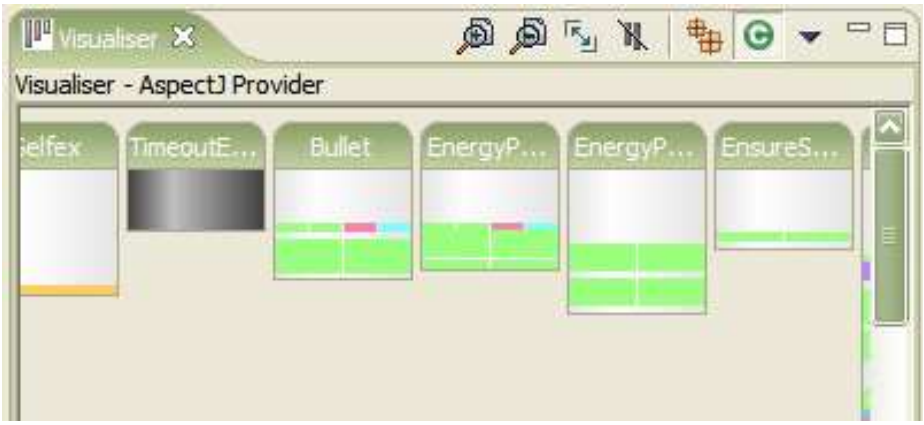

Fig. 3. AJDT Visualizer

[3] Integrated Development Environment

3.2 Software Design

The Design Patterns [7] revolution created an universal language that made thinking and talking about OOP implementation solutions a lot easier. AOP poses two new challenges in this area: are there any AOP specific design patterns, and are there some OOP design patterns that are easier implemented using AOP [8,9].

Refactoring is another major trend in the software development field. First introduced by M. Fowler[10], software refactoring is a set of methods that help developers correct software design mistakes in existing code, normally by transforming the existing code into Design Patterns. Refactoring of existing OOP code into the AOP paradigm is a new field where research is already under way [2].

3.3 Documentation and Specification

A lot of effort has been done in the area of software documentation including, for example, automatic code documentation. With the introduction of AOP and *aspects*, code documentation will have to evolve and accommodate the concepts introduced by the new paradigm.

Finally, *aspects* should not be represented only at code level but also at higher levels of abstraction. The currently most used software modeling language, UML[4], isn't prepared to represent crosscutting concerns in its various diagrams. Some work has already been done in extending UML to incorporate these kind of concerns [11,12] but some more research must be done in this field. Tools supporting these new extensions allowing modeling, code generation and reverse engineering must be created.

4 Conclusions

This paper has shown that AOP can be used to encapsulate cross-cutting concerns into modular units in a way that OOP is not capable of. Examples have been used to explain how cross-cutting concerns can be modeled using AOP.

It has also been shown that the applicability of AOP is not always unanimous. When using AOP the developer must rationalize if the aspect being modeled is really a cross-cutting concern and if using AOP will help in the developing process or increase the complexity of the system without any real gain.

By describing several future developments in the field, AOP has been described as an interesting research field with many possible research topics.

Acknowledgements

The author would like to thank Ademar Aguiar for the inspiring introduction to the world of *Aspect Oriented Programming* and Sérgio Carvalho for the endless discussions on AOP scenario applicability.

[4] Unified Modeling Language

References

1. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
2. Monteiro, M.: Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts. PhD thesis, Universidade do Minho (2005)
3. Balzer, S., Eugster, P.T., Meyer, B.: Can Aspects Implement Contracts? In: Proceedings of RISE 2006 (Rapid Implementation of Engineering Techniques). (2006)
4. Win, B.D., Joosen, W., Piessens, F.: (Developing Secure Applications through Aspect-Oriented Programming)
5. Meyer, B.: Applying "Design by Contract". Computer **25**(10) (1992) 40–51
6. : (AspectJ Development Tools (AJDT)) http://www.eclipse.org/ajdt/.
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
8. Hachani, O., Bardou, D.: (Using Aspect-Oriented Programming for Design Patterns Implementation)
9. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java AspectJ. In: Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2002)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
11. Kandé, M.M., Kienzle, J., Strohmeier, A.: (From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach)
12. Suzuki, J., Yamamoto, Y.: Extending UML with Aspects: Aspect Support in the Design Phase. In: ECOOP Workshops. (1999) 299–300