

AOSD & Security: a practical assessment

Bart De Win, Wouter Joosen, Frank Piessens

February 28, 2003

Abstract

This paper reports on our practical experience in using AOSD for security. In particular we describe one of our past case studies, which focused on the security measures of an existing FTP server implementation. Based on our experience within this scope, we elaborate on several software engineering properties of AOSD.

1 Introduction

Security is regularly claimed to be a typical example of a crosscutting concern that can be handled with AOP/AOSD techniques. But these claims are often only illustrated at a conceptual level, or with toy examples. The security requirements of larger applications are typically much more complex. In particular, for most applications, it is clear that there is no single “security concern”. Instead many related but distinct concerns contribute to the overall security of an application. Examples include access control, data protection, defensive input validation, accounting/auditing, and so on. Also secure coding, i.e. avoiding security-related bugs such as buffer overflows or race conditions, can be considered a security-related concern.

Over the past few years, we have investigated how some of these concerns can be handled using AOSD techniques, and we have grown confident that such techniques can indeed be of significant help in several security-related concerns [4, 5]. Our current research aims to generalize some of these specific security-related concerns into a framework of security aspects in order to make the process of securing an application easier and more systematic [10].

Our experience with using AOSD techniques in security-related real-life case studies has also shed some light on strong and weak points of the current AOP tools and languages. In this paper, we describe one specific case study that we have done. The case consisted of modularizing the access control and auditing concerns of an existing FTP server implementation. After briefly describing the case, we elaborate on our experiences by addressing some specific software engineering properties.



Figure 2: Migration of FTPConnection

colored class contains nothing but security related code. We have used three different colors to distinguish between three different security concerns of the ftp server: yellow for access control (authentication and authorization), red for auditing and blue for security in the remote administration.

An aspect implementation of the security requirements was achieved by migrating the security logic into aspects. We briefly sketch the result for one particular feature, access control, without showing any aspect code. The modularized access control concern consists of eight classes. Four of them are straightforward conversions of existing classes in the original model, namely those where the latter are fully colored classes in Figure 1 (FTPUser, FTPSecuritySource and its two extensions). Furthermore, four new aspects deal with crosscutting behavior: 2 aspects contain the security logic of the FTPConnection class, which includes keeping session state¹ and enforcing the access control policy, and 2 aspects have similar responsibilities with regards to the FTPHandler class. Figure 2 illustrates the migration of the FTPConnection class.² For the right-hand diagram in this figure, the advices in FTPConnectionSecurity require further explanation: ftpAuthProcessing() intercepts incoming user/pass commands and forwards them to the corresponding command ; authenticationChecks() verifies for sensitive commands whether the client has authenticated properly ; the remaining advices (...Checks()) enforce access control for sensitive operations (i.e., checking whether the user is allowed to perform the operation).

While performing this experiment, we were faced with two obstacles. First, the separation of access control frequently necessitated pointcuts referencing seemingly random, yet specific places within method boundaries of the original

¹The perthis() association of AspectJ is a useful construct to associate state with a particular object.

²The classes shown in this figure only contain those attributes and operations that are directly influenced by the migration –our apologies for the limited readability of the diagrams.

implementation. This resulted in "dirty" pointcuts that were not always related to the logic of the advice (e.g., a before advice on the next statement in the method) To give an example, the separation of the authentication phase (i.e., `doUserCommand()` and `doPassCommand()`) required breaking one monolithic case structure from which (among others) authentication was activated. Due to their relationship with unrelated statements, such pointcuts almost certainly require reevaluation in case of business logic modification. The second obstacle relates to quantification ([6]), one of the primary characteristics of AOSD. While we could sometimes take advantage of the quantification potential of AspectJ (e.g., for `authenticationChecks()`), the application was very limited for certain separations (e.g., all the access control advices differed only in subtly variations). In our opinion, the core problem for these two obstacles was due to the fact that the original code was (obviously) not developed to enable the experiment: the security logic has not been programmed with this modularization in mind, which makes some of the resulting aspects relatively complex – the net result is not the most elegant solution one could design. This probably is the reason why the code size increased³: the size of the original implementation was 154.3 kB, while the size of the code base after migration was 163.6 kB⁴.

In all, the migration of access control to aspects was not straightforward, but not insurmountable either. AspectJ offered convenient support (the necessary language constructs) to tackle this problem. One of the nice results of this case was the freedom of deployment: we could either choose to weave the aspects in the ftp server, which resulted in a functional ftp server similar to the original one, or we would choose not to weave in the aspects and as such retain a fully functional ftp server stripped from its security features.

3 Discussion

Our experiences with this case study support the well-known AOSD claims of improved modularity and comprehensibility. Regarding **modularity**, we were able to achieve a complete separation of the security features, which clearly testifies in favor of the technical capabilities of the AspectJ tool. The migration obstacles that were discussed earlier induce us however to slightly relativize this statement.

Modularization of crosscutting concerns improves the **comprehensibility** and **analyzability** of security features. In particular, since composition logic is also modularized, verification of the security deployment policy in an application (i.e., what security feature is used where) is considerably facilitated: given that the security requirements for an application are known, a security expert no longer has to check the whole codebase to make sure that all security require-

³We acknowledge that a LOC-like metric is a inferior way to compare two code bases in such case, since it does not take into account the advantages of a clean design and a good modularization. However, to our knowledge no metrics exist that take into account all such properties of a code base.

⁴Also, the extensive use of comments within the aspects also explains this growth partially.

ments are enforced; he can now only verify the composition policy. For security, this is a very important advantage, since it helps to control the frequent holes of the incomplete access mediation type (i.e., omitting a security check on a particular place).

Based on our experience, we do have some reservations with regards to another software engineering property: **verifiability**. Quality assurance is a crucial⁵ part in a typical software development process and verifiability is hence a very important software engineering property of any (AOSD) language that is part of this process. In the jFTPd case, we tested the functionality of the system by roughly comparing the behavior of the old and new implementations. For commercial systems however, such testing strategy is clearly insufficient. In general, the question raises whether the use of AOSD technology influences the verification of software.

A typical development process involves various sorts of tests (unit, integration, regression, ...) during different phases in the software lifecycle [3]. For unit testing, there is both a positive and negative impact on established testing strategies. On the upside, the advanced modularization capabilities of AOSD result in smaller and more focused units. Since only one concern is addressed per unit, the reflection on and specification of unit test criteria is facilitated. On the downside, the AOSD paradigm raises at least two important problems with regards to unit testing: the scope of a typical unit and how it should be tested. In Object-Oriented unit testing, a class is typically used as the basic test unit. AOSD offers a new concept to the programmer: an aspect. As a result, unit testing should be revisited to either take into account several unit types or to merge classes and aspects into a common unit. The second alternative is ruled out in AOSD technologies that use a dominant decomposition[9], since the asymmetric relationship makes unit testing for aspects and classes clearly different. In general, an aspect consists of both behavior and composition logic and aspect testing must hence include both the behavior of the aspect (which is comparable to but different from traditional object testing) and the binding within different environments. A good approach to tackle the latter is not straightforward, since it must include both information about the deployment environment (type information and possibly context information), as well as specific application binding information (specific state transferred between different aspects).

Integration testing is also influenced by AOSD. A common strategy for integration testing, uses-based testing, is to pursue an iterative process by first testing classes that depend on no other classes. Next, classes that use the first group of classes are tested, followed by classes that use the second group, and so on. The problem here is that the dependencies between different units are not easily predictable. Since aspect composition can break encapsulation for other units that are not aware of the aspect, it is hard to determine the exact set of aspects that must be tested for a particular subset of units. Moreover, especially

⁵According to various resources (e.g., [7]), more than fifty percent of the cost of software development is devoted to quality assurance.

in the context of dynamic composition, unpredictability of composition is a key challenge. Here, systems can change dynamically by adding/removing concerns, which makes it very hard to set up and run a rigorous test strategy.

It is clear that using AOSD technology in combination with today's testing systems is far from trivial. As for today, support for aspect testing is to our knowledge not available. A solution to this problem involves efforts of two sides: testing methods should be able to distinguish between different categories of units (a.o., classes, aspects), whereas AOSD technology should deal with the notion of unanticipated or unpredictable composition scenarios. Needless to say the latter is a core research topic.

4 Conclusion

The main contribution of this paper is twofold: it discusses a practical case study of the use of AOSD techniques for (some concerns of) security and it elaborates on several software engineering properties, with a focus on the relation between AOSD and software verifiability.

We can contribute to the debate of this workshop in several ways, among others by sharing our experiences with regards to various software engineering properties. We particularly favor discussing new possibilities to improve aspect testing and verification.

Finally, we believe that a general taxonomy of AOSD technologies, which is to our knowledge non-existent, would serve the discussion on the software engineering properties considerably.

References

- [1] FTP Server with Remote Administration. <http://homepages.wmich.edu/~p1bijjam/cs555/Project/>.
- [2] The AspectJ website. <http://www.eclipse.org/aspectj/>, 2003.
- [3] Robert V. Binder. Testing Object-Oriented Systems: A Status Report. <http://www.rbsc.com/pages/ootstat.html>, 2001.
- [4] Bart De Win, Frank Piessens, Wouter Joosen, and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. Workshop on the Application of Engineering Principles to System Security Design, Boston, MA, USA, November 6–8, 2002, Applied Computer Security Associates (ACSA), 2002.
- [5] Bart De Win, Bart Vanhaute, and Bart De Decker. How aspect-oriented programming can help to build secure software. *Informatica*, 26(2):141–149, 2002.

- [6] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000.
- [7] Mary Jean Harrold. Testing: a roadmap. In *ICSE - Future of SE Track*, pages 61–72, 2000.
- [8] 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.
- [9] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.
- [10] Bart Vanhaute, Bart De Win, and Bart De Decker. Building Frameworks in AspectJ. Workshop on Advanced Separation of Concerns, ECOOP2001.