# Aspect Oriented Programming for a component-based real life application: A case study

Odysseas Papapetrou and George A. Papadopoulos
Department of Computer Science
University of Cyprus
75 Kallipoleos Str., P.O.Box 20537, Nicosia, Cyprus
{cspapap,george}@cs.ucy.ac.cy

## ABSTRACT

Aspect Oriented Programming, a relatively new programming paradigm, earned the scientific community's attention. The paradigm is already evaluated for traditional OOP and component-based software development with remarkable results. However, most of the published work, while of excellent quality, is mostly theoretical or involves evaluation of AOP for research oriented and experimental software. Unlike the previous work, this study considers the AOP paradigm for solving real-life problems, which can be faced in any commercial software. We evaluate AOP in the development of a high-performance component-based web-crawling system, and compare the process with the development of the same system without AOP. The results of the case study mostly favor the aspect oriented paradigm.

## Keywords

AOP, Aspect Oriented Programming, evaluation, case study

## 1. INTRODUCTION

Aspect Oriented Programming, a relatively new programming paradigm introduced by Kiczales ([2]), recently earned the scientific community's attention. Having around six years of life, the paradigm was already presented in important conferences, and recently triggered the creation of several conferences and workshops to deal with it.

The paradigm is already evaluated for traditional OOP and component-based software development and is found very promising. Several evaluations consider it to be the continuation of the OOP paradigm. However, most of the published work while of excellent quality is mostly theoretical or involves evaluation of AOP for research oriented and experimental software. Unlike previous works, this study considers the AOP paradigm for solving real-life problems, which need to be faced in any commercial software. We evaluated Aspect Oriented Programming in the development of a high-performance component-based web-crawling system, and compared the process with the development of the same system without AOP. The results of the case study, mostly favoring the aspect oriented paradigm, are reported in this work.

This introduction is followed by an introduction to the AOP approach. We then describe the application that was used for our evaluation and proceed with a description of our evaluation scenario. We then present and comment our evaluation results. We continue with references to similar evaluation attempts, and, finally, we summarize the conclusions from our evaluation, and report on future work.


*SAC'04* March 14-17, 2004, Nicosia, Cyprus


## 2. ASPECT ORIENTED PROGRAMMING

Aspect Oriented Programming, as proposed by Kiczales ([2]), is based on the aspectual decomposition. Aspectual decomposition is somewhat complementary to functional decomposition, and tries to overcome the limitation of functional decomposition to capture and represent crosscutting functionality. After separating the system to functional constructs, aspectual decomposition is applied to the design in order to catch the crosscutting concerns. Crosscutting functionality usually includes extra-functional requirements (e.g. timing constraints or logging facility to all the system components). This functionality is usually replicated a number of times, spread over the whole system. There is no single point of reference where the developer can say that the aspectual functionality belongs and should be implemented.

The main purpose of AOP is to capture the crosscutting concerns throughout the system, and promote them as first-class citizens, in order to enable the modeling and reusing of them. The high level goals of such an approach, as reported in various publications, follow:

1. AOP makes programming easier and faster, closer to the human perception ([2, 3, 7]). Developers understand the concept of crosscutting concerns and crosscutting functionality, and they use it in understanding the whole system. However, apart from AOP, there is no easy way to implement such a crosscutting concern. With AOP, aspects are closer to the human perception for crosscutting concerns and simplify the design and implementation of systems with such requirements. Aspects can even allow code reuse for the extra-functional requirements they implement, which usually crosscut the whole system. Thus, they make system implementation easier and faster.

2. AOP makes programming less error-prone and easier to debug and maintain ([2, 3, 7, 6]). Not only the code becomes more modular, thus, easier to maintain and enhance, but also the goal for debugging is more easily gained (offered from the AOP inherent ability of automatic aspect invocation). Furthermore, AOP favors reusability and modular representation of crosscutting concerns, which make the code more readable and prevent tangling code.

The AOP approach is already used in the implementation of several academic-oriented systems such as [4], but there is not much work reported on AOP relating with commercial environment. However, we strongly believe that AOP can enter the industrial environment, and that it has much to offer. We expect to witness that in the near future.

# 3. THE HIGH PERFORMANCE COMPONENT-BASED WEB CRAWLER

To evaluate the AOP paradigm, we chose a high performance component-based web crawler, which would serve the needs of our laboratory. However, it was important for us to make the crawler easily extensible and changeable in order to be able to reuse it in different projects. Furthermore, the crawler should not be characterized as experimental (e.g. unstable or with extremely complicated configuration) since it should be reusable in a number of different projects, and without needing to know the complete infrastructure. We also needed the crawler to be easily adjustable to different configurations, hardware, and network situations, because of the variety of our hardware, as this would be desired in a real-life application.

This application was found suitable for our AOP evaluation, since it was of respectable size, which would give us the opportunity for better results. Furthermore, the non-experimental characterization of the current application, which is rarely the outcome in the academic environment, would ensure a more practical approach of our evaluation. For the same reason, the extra-functional requirements implemented for the evaluation, were carefully selected. It was important for us to keep the whole implementation and, consequently, the AOP evaluation not far from the commercial field, which we feel to be the important end-user of the programming paradigms.

Having these points in mind, we decided to use the following design, comprising three basic multi-threaded components: (i) the database component, (ii) the crawling component, and (iii) the processing component.

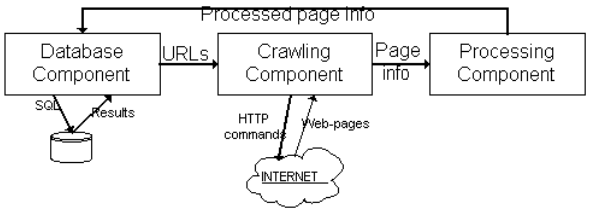


**Figure 1: The architecture of a high-performance component-based web-crawling system.**

The database component was responsible for two tasks: (a) updating the database with the processed information, received from the processing component, and (b) feeding the crawling component with the necessary URLs to be crawled. Furthermore, as in all the components, a number of threads were running in parallel in each component, so that the fast devices like CPU and memory (as opposed to the usually slow devices like I/O and network) would be more efficiently utilized. The number of threads running in parallel in each component could be selected from the user, and also adjusted dynamically from each component for optimal performance. Selecting a very small number of threads, the user would let fast resources like processor and the memory rather unutilized, while selecting an overly large number of threads would result to large context switching overhead.

The crawling component's responsibility was to download the URLs from the web and provide the processing component with the page information for further processing. Page information included the page's URL, IP address, and page text. Again, the crawling component ran a number of threads to maximize resource utilization.

Finally, the processing component was responsible for receiving the page information from the crawling component and processing it, and passing the results to the database component for permanent storage. As in the other components, this component was also multi-threaded, thus utilizing the resources better.

# 4. EVALUATION SCENARIO

To evaluate AOP in the crawling project, we ran the following scenario: First, we set our metrics for the evaluation of AOP, trying to keep them as objective as possible; then, we designed the component-based web crawler and located the different functionalities that could be modeled as aspects. Following that, we implemented and tested the three components independently. The implementation up to that point did not include any of the functionalities identified as aspects in the earlier step. Finally, we tried to integrate the three components, and also include the extra functionality, implemented with and without AOP.

Our selection for the metrics was mostly to favor (as much as possible) objective results. Our goal, as Murphy in [5] suggests, was to answer two important questions: (a) if AOP makes it easier to develop and change a certain category of software (usefulness), and (b) what is the effect of AOP in the software development (usability). For these reasons, we selected the following metrics:

1. We measure effectiveness of AOP for implementing the extra functionality, compared to traditional OOP.
2. We measure the learning curve of AOP methodology.
3. We measure time that took to complete the project with the two approaches, AOP and traditional OOP.
4. We measure complete lines of code for the added functionality with AOP and with traditional OOP.
5. We compare code tangling in the AOP and the traditional OOP model.
6. We report on the stability of the AOP model for creating component-based software.

The types of functionality that we identified as being best modeled as aspects were the following ones:

**Logging** : This functionality requires saving extended program execution trace to a file or printing it to the screen. The trace should include entrance and exit messages from the methods, exceptions thrown, and time of each event.

**Overloading checks** : Since the crawling function is expensive in resources, we must constantly check for overloading in any of the resources, in order to avoid driving the machines to collapse. The two resources we had to monitor were the DNS server that was serving our crawler and the machine that was hosting our crawling database.

**Database optimizer** : Even with the combination of the expensive high performance hardware and software that was used for the database server, we still needed to follow some optimization techniques to minimize the need for database connectivity. This was due to the heavy load that our database server experienced from the crawling function.

The Logging aspect, the most common aspect in AOP, was mostly to help debugging during the developing stage of the application, but it would also be used for identifying bottlenecks (profiling) and performing optimizations to the components in a later stage. When the logging aspect was enabled, entering or exiting a method would print (to `stderr`) the method's name, the exact time, and some other useful information. Moreover, a method throwing an exception would result in invoking the logging aspect to print the exception with the method's name in `stderr`.

The overloading checks were broken in two aspects, the DNS monitoring aspect and the database monitoring aspect. The DNS monitoring aspect was trying to adjust the number of active downloading threads according to the DNS server status. More to the point, the problem we faced was that the DNS server that was serving our crawler was shared with other machines, some of them running experimental software, doing extensive use of the DNS server for DNS resolution. This practically meant that the efficiency of the DNS server was dependent of the number of software clients using it in parallel. Running more than the appropriate (for each moment) downloading threads in our crawler (that were doing the DNS resolution) resulted in more DNS resolution requests that our DNS server could handle, and eventually, collapsing of our DNS server. On the other hand, underestimating our DNS server's abilities in low-usage hours would result in significantly lower crawling speed. For these reasons we constructed and used the DNS monitoring aspect, which would adjust the number of the downloading threads according to the running DNS load. Each DNS resolution was timed, and when discovering latency higher than expected, we were temporarily pausing some of the downloading threads (the pause time and the number of the threads that we were pausing were analogous to the latency), thus, causing less DNS lookups in a specific time.

The database monitoring aspect's goal was to disable overloading in the database machine. A similar approach to the DNS monitoring aspect was used. We were monitoring the responses from our database server and when we were detecting overloading of the database we would pause some of the downloading threads. The reason that we could not predict the ideal number for the database component threads from the beginning was because of the variety of the web-pages. For example, a web-page with many new words (words that are for first time parsed from the crawler) would result in much database load, while words that are seen before from the crawler would result in much less (due to some optimizations, similar to those proposed in [1]). For these reasons, we constructed the database monitoring aspect to monitor database queries. The aspect would time every interaction with the database server and try to detect overloading. When the time demanded for the query was bigger than a threshold (all the queries we were executing were having the same average time for execution in normal circumstances), we would pause some of the downloading threads for some time, in order to allow the database server to complete its work without extra work added at the same time. Later on, the downloading threads would resume their work.

These two last aspects would not contradict each other, since they were both doing the same action, pausing some of the downloading threads. However, the pause time and the number of the downloading threads to pause were not the same in the two cases. Each of the aspects was calculating the time and the number of threads to pause with a different algorithm.

Finally, we also constructed the database optimizer aspect which acted as a database cache and released some of the database load. More specifically, for the parsing function we were making heavy usage of the crawling dictionary table from the database. That dictionary was matching every word we found up to the moment with its id number. The choice was to avoid needless and costly replication of data and enable saving the page text as numbers (smaller in storing size and faster in seeking). By keeping a memory cache of that table as in Brin's implementation ([1]), we would manage to get important workload off the database server and speed things up. More to the point, prior addressing the database for a word's serial number, we were querying an indexed structure in the local memory. If the query failed, we were then inserting the word in the database and in the RAM dictionary and continuing our work. This minimized the database interactions and boosted the complete process, since RAM access was enormously faster than access to the database. Processing English language pages with an average-size dictionary of 1 million words would result to around 99,9% success from the RAM table, thus, it would prevent querying the database very efficiently.

## 5. EVALUATION RESULTS

As already mentioned, these four aspects were implemented in two distinct ways: (a) injected in the program code, using standard OOP approach, and (b) modeled and implemented as aspects. The two versions were then compared and evaluated in the described metrics. The results from the evaluation were mostly in favor of the AOP methodology. While the developers were not long experienced in AOP, the new model boosted the implementation speed and helped in more modular software.

Regarding effectiveness of the AOP approach compared to the traditional OOP approach, the two approaches were the same. We managed to add the extra functionality in both versions of the software (however, it was not always trivial to do so). In short, for the presented aspects there was always an AOP-oriented and an OOP-oriented solution

available, and there was not a noticeable performance difference between the two.

Regarding the time demanded to learn the AOP methodology, this was not significant. Both the developers that were working on the project were very experienced with OOP, but did not have previous practical experience with AOP. Fortunately for the project, they were able to learn AOP sufficiently without tutoring using only publicly available online sources in a single week. There was also another short overhead of one day for installing and getting familiar to an AOP-aware IDE (we used Eclipse with the AOP modules).

The complete time that was required to finish the crawler was shorter in the AOP version (this time did not include the time spent for learning AOP however). Both the versions used the same core already developed (the three components demonstrated earlier) but they were continued completely independently, without reusing knowledge or code from one version to the other (the nature of the two versions prohibited reusing knowledge or code anyway). The time demanded for completing the crawler with the aspects in the AOP version was 7 man-hours, while the OOP version demanded 10 man-hours in order to design and develop the code. Most of this time, in the case of the OOP version, was needed for locating the methods and putting the necessary code to them. For implementing the logging functionality for instance, in the OOP version there were 73 such methods counted, while AOP did not demand this task since the pointcuts were found automatically from the aspect definition. It was the developers' feeling that most of the man-hours spent in the OOP version of the crawler were *wasted*, because they were repeating trivial code in the application. Furthermore, as they said, the result in the OOP case was not satisfactory for them since, if they needed to change something in an aspect, they should relocate the aspect code from the beginning and this would be difficult to be done.

We also measured the number of lines we needed to add in the two approaches to implement the extra functionality. For the logging aspect with the AOP approach, we needed less than 20 lines in one single file, while the same functionality for the traditional OOP version required 126 lines of code spread in eight different files (the number of lines for the AOP code also include the pointcuts definitions and the java include directives). This, apart from a time-demanding approach, also reveals important code tangling since we had to modify eight classes for a simple logging requirement.

The other two aspects, the DNS monitoring and the database monitoring aspect, needed roughly the same number of lines for the two versions. To implement both the DNS monitoring and the database monitoring functionality, we needed around 30 lines for the pure OOP solution: (a) four lines for timing the DNS or the database query, (b) ten lines for checking for overloading, and proceeding in alternate behavior if overloading occurs, and finally (c) one line for invoking the check wherever needed. In the AOP solution, we were able to join the two concerns in a single aspect - something that we were unable to do in the OOP version - and reuse some of the code. The AOP version of the solution demanded roughly the same number of lines, around forty for both the concerns (the additional code was because of aspect and advice headers and the pointcuts definition).

Finally, the database optimizer needed the same number of lines in the two versions, that is forty lines. These lines in the OOP version were split in three different places in the original database component file, while at the AOP approach the original file was kept intact and all the new code was in a single aspect-definition file.

We also tried to capture the code tangling that occurs in the two versions, after the extra functionality is added. To do that, we found the distribution of the added code in the eight affected files. The OOP version of the logging aspect, as expected, was spread in all the eight files in seventy-three different places. The OOP version of the DNS monitoring aspect resulted in addition of code in one file only, the downloader component file, in three different places. Similarly, the OOP version of the database monitoring aspect resulted in addition of code in the database component file, in three different places. Finally, the database optimizer aspect implementation, without AOP, also resulted in code addition in three different places (again, in the database component file).

On the other hand, implementation of the four aspects with the AOP approach, as expected, created no code tangling. The complete code for the extra functionalities was included in the three (instead of four, since the DNS and the database monitoring concerns were implemented as a single aspect) aspect files. For the case of the database optimizer, this offered us another important advantage since we often needed to disable the database optimizer due to hardware (memory) limitations in weaker machines. While we did not take any provision for that, the AOP version, unlike the OOP version, enabled removal of the optimizer without changing any code. In the OOP version, the developer had to remove or modify some of the original code.

Table 1 summarizes the results for the code size and code tangling for the four aspects:

Finally, the implementation of AOP we used, combined with the IDE tool, were stable and did not cause us any unexpected problems (such as bugs in the compiler). While the crawling system was not extremely big, it did make extensive use of the machines' resources, and AspectJ compiled files did not face any trouble with that. AspectJ compiled files proved to work fine under pressure with the standard virtual machine, and aspects introduction was not causing a noticeable overhead to the machines.

## 6. RELATED WORK

Several publications try to evaluate AOP. Almost all of them report results similar to ours. However, while of excellent quality, most of the previous work we are aware of follow a theoretical approach or limit their hands-on evaluation for academic or experimental software. We will now briefly comment on some of them.

Walker in [8] constructs several experiments and a case-study to evaluate AOP. The outcome of the evaluation is that AOP can help faster software development (programming, debugging, etc.) under certain conditions, while other cases make development of AOP less attractive. While of superb quality and significant importance, this work is limited to the evaluation of AOP based on a preliminary version of AspectJ, version 0.1. Since then, AOP and especially AspectJ, changed significantly confronting most of the limitations detected in the evaluation, and also powering the users with more functionalities. Furthermore, CASE tools and powerful IDE environments were developed to assist the

| Aspect | # Lines of code | | # Places to add | | # Files to add | |
|---|---|---|---|---|---|---|
| | OOP | AOP | OOP | AOP | OOP | AOP |
| Logging | 126 | 19 | 73 | 1 | 8 | 1 |
| DNS Monitoring | 15 | 40 | 3 | 1 | 1 | 1 |
| Database Monitoring | 15 | | 3 | | 1 | |
| Database optimizer | 45 | 45 | 3 | 1 | 1 | 1 |

**Table 1: Size of added code and code tangling for implementation of the aspects. The two aspects, DNS and Database monitoring were easily joined to a single aspect in the AOP version**

developers in the process.

Mendhekar in [4] also presents a case study, evaluating AOP in an image processing application. Although AOP was then still in infancy, this case study presents results very similar to ours. However, Mendhekar, being in Xerox labs where AOP was born, follows a more research-oriented approach during the evaluation. The evaluation uses an AOP implementation that cannot easily be used from people outside the Xerox environment. Also, being interested in performance, this work does not elaborate on various other important measures, such as the learning curve and the time that took the developer to complete.

Several other important publications ([2, 3, 7, 6]) evaluate AOP from mostly a theoretical approach. Most of them also report results that favor AOP programming. Some of their results are reported in section 3 of this report.

## 7. CONCLUSIONS

During the construction of the component-based high-performance web crawler, we had the opportunity to evaluate the relatively new aspect oriented paradigm for building component-based systems. Having defined our extra functionality, we implemented and compared the two versions of the web crawler, the AOP and the OOP one. For the required extra functionality, both the paradigms proved able to implement a correct solution. The quantity of code (number of lines) that the developer needed to implement in the two versions was not of much difference, with the only exception of the logging aspect where the OOP implementation was much larger than the AOP one. Furthermore, in both the versions of the application there was no apparent performance difference. Both the versions were stable, even when working under high load and in varying system environments. The significant difference however between the two implementations was in the time required to develop and debug each of them, and the quality of the produced code. The AOP approach not only completed the system faster, but it also produced modular high quality code, while the traditional approach was creating the well-known spaghetti code. More specifically, the AOP version was having all the extra functionality apart of the code implementing the standard functional requirements. This not only kept the original components reusable in different implementations, but also prevented tangling the code, thus, making future maintenance easier. Furthermore, this enabled us to easily enable and disable the extra functionality, depending on the hardware resources available and on our requirements.

Concluding, we have to report that the AOP model in general appears to favor the development of quality component-based software. The AOP model itself is able to boost the implementation speed without negatively affecting quality of the software. Moreover, the learning time of the model, judging from our experience, is not long. While not having much experience of AOP implementation languages, we were able to produce AOP-based code in no time. Finally, while AOP cannot offer any solution to problems unsolvable from traditional approaches, and while AOP does not always target to less code, it can offer better and easier solutions to programs that are otherwise difficult to be implemented. Therefore, we can safely arrive to the conclusion that AOP has much to offer in component-based software development. We strongly believe that integration of AOP with component-based software is going to be the target of important research attempts in the near future and can produce some very interesting results, and we await for the introduction of AOP software in commercial component-based software products.

## 8. REFERENCES


[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, pages 220–242, Springer-Verlag, 1997.

[3] C. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, November 1997.

[4] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox Palo Alto Research Center, Palo Alto, CA, USA, February 1997.

[5] G. C. Murphy, R. J. Walker, and E. L. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. Technical Report TR-98-10, Department of Computer Science, University of British Columbia, 1998.

[6] A. Navasa, M. A. Perez, J. Murillo, and J. Hernandez. Aspect oriented software architecture: a structural perspective. In *Proceedings of the Aspect-Oriented Software Development*, 2002, The Netherlands.

[7] D. Shukla, S. Fell, and C. Sells. Aspect-oriented programming enables better code encapsulation and reuse. *MSDN Magazine*, http://msdn.microsoft.com/msdnmag/, March 2002.

[8] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. Technical Report TR-98-12, Department of Computer Science, University of British Columbia, Sept. 1998.