ASPECT ORIENTATION VS. OBJECT ORIENTATION IN SOFTWARE PROGRAMMING

An Exploratory Case-study

Anna Lomartire, Gianfranco Pesce
Università degli Studi di Roma "Tor Vergata"
Centro di Calcolo e Documentazione, via O. Raimondo, 1 - 00177 Roma, Italy

Giovanni Cantone
Università degli Studi di Roma "Tor Vergata"
Dipartimento di Informatica Sistemi e Produzione,via del Politecnico, 1 - 00133 Roma, Italy

Keywords: Aspect-oriented Programming, Object-oriented programming, Empirical Software Engineering.

Abstract: Aspect orientation is a software paradigm that is claimed to be more effective and efficient than Object orientation when software development and maintenance interventions are taken in consideration that affect transversally the application structure, namely Aspects. In order to start with providing evidence able to confirm or disconfirm that opinion in our context - software processes that we enact, and products that we develop at our University Data Center - before launching a controlled experiment, which would require the investment of large effort, we conducted a preliminary explorative investigation that we arranged as a case study. We started from a Web-based object-oriented application, which engineering students in Informatics had constructed under our supervision. We specified new user needs, which realization was expected to impact on many of the application's classes and relationships. Hence, we applied another student to realize those extensive requirements by using both Aspect orientation and Object orientation. Results show that, in the average, both the completion time and the size of the additional code advantage significantly the Aspect orientation, for maintenance interventions that are transversal to the application's structure, with respect to the characteristics of the experiment object utilized, the specified enhancement maintenance requirements, and the subject involved with performing in the role of programmer. Although the exploratory nature of the study, the limited generality of the utilized application, and the fact that just one programmer was utilized as experimental subjects, the experiment results push us to verify the findings by conducting further investigation involving a wider set of programmers and applications with different characteristics.

1 INTRODUCTION

Object Orientation (OO) is a software paradigm, which nowadays has become a classic of software analysis, design, and programming, so that we do not mind to recall his characteristics in this paper.

Aspect Orientation (AO) is a software paradigm too; it constructs on OO (Kiczales et al., 1997) (Laddad, 2002) (Bonin, 2002) (TAODS, 2003) (Eclipse, 2007) (AOP, 2007) (AODS, 2007), and tries to make effective and efficient software development and maintenance, when interventions are transversal to, act on most of the structure of, the software application, its classes, relationships, and execution scenarios.

AO is based on the concept of Aspect, which deals both with mechanisms, like security and persistence, and utilities, like control-path tracing, e.g. for structural testing and logging.

While nowadays the AO paradigm is spreading to software architecture, analysis and design (Keuler, Naab, 2007), this paper focuses both on programming - particularly the impact on software code of enhancement maintenance interventions - and process efficiency. Some tools support the Aspect Oriented Programming (AOP), including programming languages, like an Aspect oriented version for Java (AspectJ, 2007).

Based on the AO paradigm, the classes of a program are not requested to be aware of the program's aspects. For instance, in order to extend a program by using aspects, the work of an AO programmer in not to change program classes but consists of identifying and coding the needed aspects, as specific programming units, and defining the points of those classes and methods, and

Lomartire A., Pesce G. and Cantone G. (2007).
ASPECT ORIENTATION VS. OBJECT ORIENTATION IN SOFTWARE PROGRAMMING - An Exploratory Case-study.
In *Proceedings of the Second International Conference on Software and Data Technologies - PL/DPS/KE/WsMUSE*, pages 47-54
DOI: 10.5220/0001349800470054

the conditions (object states and events), which claim for the activation of an aspect.

While there are works that are assertive of the AO, or also present concepts and practical examples about the usage of that paradigm (Baccan, 2004) (RCOST, 2007) (Merlo, 2007), it is quite limited the number of studies that investigated AO *empirically*, i.e. using controlled experiments, case studies, or surveys.

At our University's Data Center ("Centro di Calcolo e Documentazione", CCD), where we design and develop software for providing services to the Administration and students, we have been reflecting on the utilization of AOP for maintaining or re-engineering our applications. In order to become confident with AO and related technologies, and in the aim of understanding what advantages and disadvantages might derive from using AOP, in which circumstances, and in what extent, our decision was to start first an exploratory study, and then, in case of successful results, to launch a controlled experiment to conduct initially with students of engineering and then with our professionals.

The scientific conjecture for this our work is that the development and maintenance of software with structurally pervasive characteristics should find relative advantage in using AO rather than OO.

Many points, which are in our research agenda but this paper cannot afford, include but are not limited to: (i) Pros and cons of AO vs. OO, when software requirements are provided, which have, and respectively have not, a transversal impact on the software architecture, for development from the scratch or maintenance interventions, respectively; (ii) Impact of AOP on readability, comprehensibility, efficiency, testability; (iii) Debugging and static analysis of AOP vs. OOP software applications. Moreover, in what extent: (iv) The extension points that AOP provides can be utilized to implement and manage dependence relationships between use cases; (v) The application characteristics (e.g. its structure) have influence on the utility of applying AOP rather than OOP.

In the remaining of the present paper, Section 2 recalls previous work, reasons on AO and AOP mechanisms; Section 3 sketches on AspectJ and its AOP constructs; Section 4 presents the case study, Section 5 and Section 6 present and discuss the related results, respectively. Some final remarks and forwards to future work conclude the paper.

2 ASPECT ORIENTATION

In the traditional OO approach, it is complex or impossible to utilize modular entities like classes, and methods to model behaviors, which spread anywhere in the application, do not need explicit invocations, and are specified and defined in one point of a software artifact. Memory management, security mechanisms, logging, fault and exception management are some of the characteristics that suffer for such a limit.

AO has been proposed as a solution for problems of that kind. In the view of the AO supporters, OO should deal with the business logics, domain entities, and interaction with the external world; AO should deal with all the remaining. Let us consider, for instance, a data-management system component, which is in the responsibility to "Insert a new customer in the DB". This component should also deal with some access-related auxiliary concerns, like authorization, tracing, and policies. These auxiliary concerns have pervasive impacts; we need programming supports, which allow having just a copy of those access checks in a program rather than duplicating them in all the impacted operations. The AO approach claims to have enough expressive power to meet all those constraints and requirements in the whole application.

AO provides design concepts, and programming constructs and mechanisms, which allow practitioners to isolate functionalities that impact transversally on the application system. AO calls these functionalities with *crosscutting* and groups them and related activation points in an *aspect*. For instance, with regard to the previous example of data-management system component, it should be a good choice to evaluate for crosscutting eligibility those functionalities that deal with DB access policies, authorization, and tracing.

In the AO context, the term *Concern* is used to denote a concept or area of interest, or also a requirement or functionality, eventually a behavior, which can be thought and developed autonomously.

There are AO Concerns of two kinds: *Core Concern* and *Crosscutting Concern.* OO is well able to model the former by using delegation and/or inheritance and polymorphism. The goal of AO is to capture the latter kind and provide simple means to represent it in a software artifact by minimizing dependencies between the involved entities.

In order to introduce a Crosscutting Concern into a program, a programmer is just requested to provide the desired new behavior (Concern) and the application points where s/he wants that the behavior is entered. Hence, two fundamental parts compose an aspect:

- *Advice*, which provides a full implementation of the Concern;
- *Pointcut*, which defines a family of points in the program flow where to exec the aspect.

A pointcut, in its turn, can be expressed by a combination of *Join points*, each expressing a

characteristic point of the program flow, like the invocation of a certain method or constructor, the access to an attribute, or the raise of an exception.

The construction of an AO software is structured in four sequential steps, and includes OO construction:

- Aspects identification and decomposition: here each transversal service, is detected.
- Class structure definition: the same as for OO.
- Independent implementations of classes, and of transversal services as apects.
- Aspects re - composition (*Weaving*).

3 USING ASPECTJ TO PROGRAM ASPECTS

AspectJ is a Java compatible AOP language. It adds concepts and constructs like joint point, aspect, pointcut, advice, and inter-type declaration to Java.

3.1 Aspect

An Aspect is the modular unit that AspectJ provides to express crosscutting concerns. Classes and aspects share syntax and structure less the keyword `class` that moves to `aspect`, and the fact that an aspect can also include pointcuts, advices, inter-type declarations, and other aspects. Of course, an aspect can also contain what a class can include: e.g., constructors, attributes, methods, and inner classes.

3.2 Join-point

In an AspectJ, a joint point is a well-defined point, which regards the program flow, including asynchronous events and exceptions that might occur at run time.

AspectJ provides support for implementing some types of join point, including:

- Invoking a method or constructor.
- Entering a method or constructor.
- Initializing an object.
- Reading from or writing into an attribute.
- Executing an exception handler.

3.3 Pointcut

A `pointcut` allows the definition of a one or more join points; in other words, in order to intercept an occurrence, a pointcut is specified in such a way to define join points.

AspectJ provides primitive pointcut designators that allow a programmer to define many types of join points. A pointcut designator is a matching tool for an application's identified event set (join points).

The syntax of a pointcut is: `[visibility-modifiers] pointcut name(ParameterList): PointcutExpression;` where the `visibility-modifiers` field allows to explicitly define the visibility of the pointcut in {`public, protected, private`}. The field `name` is for the user-defined pointcut's unique name. `PointcutExpression` indicates what the pointcut has to intercept; logical operators can be used to combine primitive pointcut designators.

For instance, it specifies the interception of all the invocations of the methods `setAnAtb(int)` and `setAnotherAtb (int)` in the class `AClass`, the expression: `call (void AClass.setAnAtb (int)) || call(void AClass.setAnotherAtb (int))`.

Finally, wild cards are allowed for use in a pointcut expression. For instance, it specifies the interception of any method in any class with name suffix "AClass", the following expression: `call(*AClass.*(…))`. Of course, if a naming convention is applied, wild card can be extensively utilized. Vice versa, an existing application might cause additional effort for defining the pointcuts if its naming is not consistent.

3.4 Advice

Advices allow users to define the code to exec when a join point is intercepted. There are three types of Advices that AspectJ provides:

- *Before advice* is invoked when a join point is reached, i.e. just before the call of the method that the join point specifies. An instance of such an usage follows, where a message is printed just before any occurrence of a specified event: `before(): anEvent(){<print msg>}`
- *After advice* is invoked when, following the interception of a joint point, the control flows goes again through to the intercepted join point, i.e. at the end of the invoked method, just when this returns the control to the caller. An instance of such an usage follows, where a message is printed just after any occurrence of a certain event: `after():anEvent(){<print msg.>}`
- *Around* advice is executed in the range that Before and After advices define. This is the most power type of advice. It includes all other advices; additionally, it may change the execution context to install on return from the method. The syntax for such an advice is: `ReturnType around (ParamList);` in order to specify when to return from an Around advice to the related

joint point, a call to proceed is used, which returns a value of the ReturnType.

3.5 Inter-type Declaration

The Inter-type declaration is an advanced and risky feature that allows changing the structure and behavior of a software module by adding attributes, methods or constructors, and modifying class relationships. The syntax of an inter-type declaration is:

- For an attribute: [Modifiers] FieldType TargetType.Id;
- For a method: [Modifiers] ReturnType TargetType.Id (Formals) [throws TypeList] {Body};
- For a constructor: [Modifiers] TargetType.new (Formals) [throws Typelist] {Body};

The default visibility of an attribute or method is private; it can be set to public. For instance, the statement: int AClass.anAtb=0; specifies that the class AClass is requested to include the private integer attribute anAtb and to initialize it to zero for any instance. The further statement: public int AClass.getAnAtb () {return this.anAtb;} specifies that AClass is requested to have an integer method that is called with getAnAtb() and returns the value of the receiver's data field anAtb. Finally, the following statement: declare parents: CC extends AClass; sets AClass as superclass of CC. Similar syntax is used for declaring the implementation of an interface: declare parents: AClass implements IC.

4 THE CASE STUDY

In order to start comparison of AOP and OOP, we made decision to design and eventually conduct an enhancement maintenance case study. The goal (Basili, Caldiera, Rombach, 1994) and the consequent underlining hypotheses that we assumed for this initial study were that for some types of enhancement maintenance interventions on small-medium size data-management Web Java OO applications, it should be significantly worth, effective and/or efficient, to use AOP rather than OOP in the context of an academic Data Center, with junior programmers, from the researcher point of view.

Based on the case study goal, in the design stage, we choose to add some orthogonal utilities and use-cases to a pre-existent student-made Java J2EE Web application for home-library management.

Additionally, because we made design decision to involve no more than a couple of subjects to perform as programmers in the initial exploration of AOP vs. OOP, our further decision was to proceed by a paired case study, that is to indicate each subject, one or two, to utilize both the treatments in random order, hence to develop each maintenance intervention two times, by using AOP and OOP, respectively.

4.1 The Application to Enhance

The use case object is FamilyLibraryMgt, FLM, i.e. a software application, which allows a family to manage musical, artistic, or reference materials (as books, manuscripts, recordings, or films) in terms of media catalog, search, and allocation in the available bookcases.

FLM is a Web application, which is Model-View-Control designed and includes 4 packages, 46 classes, 579 methods and 46 constructors. Beside the Internet access, two human Actors define the application's boundary: the Administrator and its super class User, who interact with FLM by 11 use cases.

Once the application owner registers himself or herself into the application system, s/he creates the personal virtual FLM, and is set as the FLM's Administrator. S/he can then store or search the FLM DB for media, and register other users.

For registration of a new User, the application asks the Administrator for some user attributes, including username and initial password.

Registered users are allowed to manage the library in the limits specified by the level of authorization that the administrator assigned them.

For including a new item in the system, this asks the user for attributes that any medium has, and some other ones, which are medium-type specific, like Book, Magazine, VHS, ACD, DVD.

Functionalities that the application provides are:

- Insert; the specification of a physical location in a bookcase for the real medium is also requested.
- Search; both simple and advanced types of searches are provided.
- Lend, concerning media that friends lend/ borrow.
- Cancel a media by providing motivation and confirmation.

4.2 The Maintenance Requirements

The enhancement maintenance requirements of FLM are reported in the following, as sketched in terms of a non-functional requirement (L) and two use cases (K, M) additionally requested for FLM.

- Logging (L): the system updates the Log file anytime a method is entered.
- Controlled access (K): the system is requested to manage the access to videos depending on the age of the user. In the user view, as a result of this intervention: Each registered VHS and DVD has the attribute usageLabel, which gets value into {Green, Yellow, Red}. Moreover, restrictions to access are reinforced, so that a medium is like not existent in the electronic library for people who have not the requested authorization and access level. Furthermore, the access to videos is restricted as in the following:
  In case of usageLabel:
  - Green: no restriction applies.
  - Yellow: for 14 or older.
  - Red: for 18 or older.
- Insertion management (I): the Administrator is the only user allowed to assign a usageLabel different from Red to, or change the usageLabel of, a video medium.

4.3 The Case-study Participant

When we were ready to start with the case study, one subject was available, and we made decision to begin the exploratory study with just that participant.

A bachelor engineer in Informatics less final dissertation was involved as subject in the case study. Previously: he had performed in the average, as a student of OOP and OOAD; moreover, he had never experienced as a software professional or been exposed to AOP.

4.4 Training

Before starting with the case study, the subject had to refine his Java OOP practical knowledge by analyzing and constructing an UML-documented small size data-management application under the supervision of professionals at the CCD.

Successively, he attended a briefing on AOP introductory elements.

4.5 Threats to Results Validity

Based on the training received, the subject should be considered much more expert with OOP than AOP.

Consequently, the validity of results from the case study should be considered as threaten, whether giving an advantage to OOP; vice versa, results should be emphasized, whether providing an advantage for AOP.

In order to keep in control the further threat to validity, which relates to the learning effect that is surely associated with using one subject in a paired case study, we instructed the participant: (i) to start the work with AOP; (ii) to change the treatment to use first (OOP, AOP, OOP, and so on) in the current pair when passing from a requested enhancement to the next one, and (iii) to not change the treatment before completing the current intervention. In practice, he applied three paired maintenance enhancements by using orderly treatments as in the following: (L: AOP, OOP), (K: OOP, AOP), and (M: AOP, OOP).

4.6 Case-study Operation

Eventually, both the available application and the maintenance requirements for extension L were given to the subject, who was then instructed to make the requested ordered pair (AOP, OOP) of interventions. When he had finished with these, the requirements for K and M were assigned him, and he was requested to develop first the pair of interventions (OOP, AOP) for extension K, and then the pair (AOP, OOP) for I requirements.

In order to allow replications of the study by interested scientists, if any, some further details are given in following.

The subject was invited to develop the maintenance interventions where and when he preferred, but to refer systematically to the involved academics: tutor and professionals.

The subject preferred to work at his own home and meet academics on demand, depending on his work advancement or the occurrence of blocking doubts. During the development, academics spent one man-hour per week, in the average, for meeting the subject.

The enhancement maintenance implementation is sketched in the following sub-sections. Cumulative results are given first for the whole interventions (Sub-Section 4.2.1). Subsequently, the interventions are classified per the requested non-functional requirements (L) and use cases (K and I), respectively, one per sub-section, and presented with some details.

5 RESULTS

5.1 Total Results

It follows the total size (numbers of methods, classes, aspects, and Source Lines Of Code, SLOC, default value is zero) of, and time spent for, AOP and OOP maintenance interventions on the pre-existent FLM application, respectively. It counted

one time per intervention an impacted artifact, e.g. a class was counted two times if it was modified by two interventions.

AOP:
- New artifacts:
  - 5 methods, 16 SLOC.
  - 2 aspects, total size 45 SLOC.
- Amount of time spent:
  - 135 minutes.

OOP:
- New artifacts:
  - 1 class.
  - 6 methods, 28 SLOC.
- Changed artifacts
  - 49 classes.
  - 582 methods, 721 SLOC.
- Amount of time spent:
  - 530 minutes.

5.2 Logging

The maintenance intervention was first realized in AOP and then in OOP for this utility.

AOP. The AOP implementation of this extension includes an aspect, which names `Logging`, where a pointcut is defined, which is named with `logPoint()`, and is able to intercept the invocation of any of the 579 methods that populate the application. Additionally, an after-advice is defined: subsequently to `logPoint()`'s interception, the aspect's method `addItem(String log())` is invoked, which eventually appends the identifier of the receiver and the signature of the invoked method to the Log file.

OOP. The OOP implementation of this extension includes the new class `LoggingPrint`, which is in the responsibility of updating the Log file. Additionally, each method has been extended to include the following Java instruction:

```
LoggingPrint.addLogging (obj&methodInfo);
```

Results. On the AOP side, 12 SLOC in an aspect (`Logging`) were sufficient to implement the maintenance intervention. The amount of time spent to enact this extension is 60 minutes.

On the OOP side, 1 new class, 6 new methods 28 SLOC; 46 modified classes, 579 modified methods (705 SLOC) are the numbers that characterize the size of this intervention, which impacted on all the previous methods and classes. The amount of time spent to enact this extension is 480 minutes.

5.3 Controlled Access

The maintenance intervention was first realized in OOP and then in AOP for this use case.

OOP. A class has been modified for managing the user's birth date. Additionally, in the class `PostgresqlDAO`, the method has been modified, which is in the responsibility of loading a video medium from the DB. Only those media are loaded which the current user is allowed to access.

AOP. As before for the OO intervention, a class has been modified for the user birth date management. Additionally, an aspect has been created and implemented, which names with `CheckForAdministrator`. In such an aspect, two pointcuts and related advices have been defined and called with: `receiveAdmrAttributes()` and `usageLabel()`, respectively. The former is in the responsibility of intercepting application system logins. Just *after* such an execution, an advice gets the birth date of the current user. The latter is in the responsibility of: (i) intercepting `PostgresqlDAO` invocations to the method `setUsageLabel (String)` in the class `Video`; such a method is also called whenever the system tries to load a video medium from the DB; and, (ii) calling the related advice. Only videos are made visible that satisfy the conditions that the use case defines.

Results. The same amount of new lines of code was necessary to implement the OOP and AOP interventions.

The amount of time that the subject employed to enact this extension is 60 minutes for the AOP intervention, and 30 minutes for the OOP intervention.

5.4 Insertion Management

The maintenance intervention was first realized in AOP and then in OOP for this use case.

AOP. The aspect `CheckForAdministrator` and a class (see Section 5.3) have been extended by 57 Java SLOC. The latter includes the pointcut `checkUsageLabel()`. This contains join points that concern: (i) `PostgresqlDAO` calls for the method `getUsageLabel()` in the class `Video`; such a call occurs whenever the system is requested to store new videos into the DB, and (ii) any execution of the method `getUsageLabel()` in the Java Bean class `InsVideoBean`; in fact, such a method receives attributes of the video medium to insert in the DB; the user defines values for those attributes by filling in the fields of the user interface. An advice is associated with such a pointcut, to apply controls that depend on the type of the current user (Administrator or not).

OOP. A class has been extended as for the AO intervention. Following that step, based on the use case requirements, a couple of lines of code have been added to two methods: the method which, as part of the class `PostgresqlDAO`, is in the responsibility of inserting video media into the DB; and the other method which, as part of a boundary class, manages the interaction with the user through the user interface.

Results: The AOP and OOP maintenance interventions are practically equivalent in term of adjunct size and effort spent. In both cases the same class was modified. Additionally, the OOP implementation impacted on two more classes.

The amount of time spent to enact this extension is 15 minutes for the AOP intervention, and 20 minutes for the OOP intervention.

6 EVALUATION OF RESULTS

Concerning the maintenance intervention as a whole:

- 61 SLOC for 5 new methods and 2 aspects, and 135 minutes plus testing time are the size and duration time data, respectively, which characterize the AOP intervention.
- 28 SLOC for 1 new class and 6 new methods; 721 SLOC for 49 modified classes and 582 modified methods, and 530 minutes plus testing time are the size and duration time data, respectively, which characterize the OOP intervention.

Thus, in order to implement the same functional changes in the given application, AOP needed 8% of the OOP SLOC, and 25% of the OOP duration time, in the average. This result shows a clear advantage of AOP versus OOP, for structurally pervasive maintenance interventions.

The efficiency is 27 SLOC/Hour for AOP, and 85 SLOC/Hour for OOP. Let us recalls that an OOP-expert AOP-naive junior programmer was involved to perform as subject in the case study. In practice, the former efficiency is affected by the time that the subject needed to learn AOP concepts and programming language. These results might also tell us that AOP requires more reflection time than OOP.

Additionally, we should take in account that the added aspects are full reusable and, in practice, reuse is quite for free; vice versa, in order to introduce the same functionalities in another application, the OOP programming should be mostly done again.

Moreover, it should be considered that this maintenance intervention was specified to manage and control just the video materials. The same aspects that the AOP interventions introduced would hold all the present and future types of medium. Vice versa, changes to many classes and/or the application structure should be enacted, in case of OOP interventions.

Last but not least, structuring AOP has not been yet in focus, e.g. introducing generalization and specialization of aspects. It is reasonable to expect that it would be easy for an aspect to subsume such concepts and related constructs whenever they should be included by AOP languages; vice versa, it is reasonably without hope any tentative aimed to give structure to code spreading application-wide.

Logging. For the FLM object, the following advantages were observed for AOP vs. OOP: (i) clear reduction of code-change diffusion (two very small localized groups of instructions were sufficient to enact the AOP intervention, while 705 SLOCS spread through 46 classes, i.e. 100% of the OOP application, in the OOP intervention), (ii) very large reduction of the development time (60 vs. 480 minutes), (iii) full reusability of the added aspects.

Concerning productivity: it is 12 SLOC/Hour for AOP, and 92 SLOC/Hour for OOP. Again, AOP could be much more complex to use than OOP. However, the AOP inexperience of the case-study subject should be taken in consideration, hence the time he employed to learn about that paradigm and related techniques and tools.

In conclusion, based on this case, it seems that using AOP rather than OOP should significantly improve the return on investment.

Controlled access. For this use case, no significant difference was observed between AOP and OOP interventions. In fact, concerning the code size, a very small advantage was observed for OOP; concerning the development time, OOP needed 30 minutes less of AOP, which lasted 60 minutes. This should confirm that AOP is much more complex to comprehend and enact than OOP. It should be also considered that the use case was just applied to video media; the extension to other types of media should come quite for free, when the aspect is taken in consideration, but the effort should grow, when OOP is considered. In fact, concerning the former, it should be enough: (i) to extend the pointcut `usageLabel()` so that it can intercept also invocations of other media objects, and (ii) to introduce a small change into the related advice. Vice versa, in the OOP approach, the code already written for videos should be replicated in the classes of the other media or else the application should be re-structured by introducing a further abstract class, e.g. Medium, as the super class of all the media.

Insertion management: Also for this use case, no significant difference was observed between the AOP and OOP interventions. However, while both the interventions did modify the same application class, the OOP intervention also modified two further classes. Such an OOP additional work should give again advantage to AOP, in case the enhancement is extended from video material to all the library media: in fact, such a further extension would utilize a common aspect.

7 CONCLUSIONS AND FUTURE WORK

After a brief recall of the Aspect Oriented (AO) philosophy, concepts, and programming languages, this paper has been presenting a case study aimed to explore advantages, if any, that AO can return when structurally pervasive maintenance interventions are applied to an OO designed and implemented data-management Web application.

Some results from the case study are very interesting, but need confirmation. We are now in the planning phase of confirmatory experiments to conduct first with some decades of students, and then with programmers.

Such prospective experiments should be also aimed to investigate:

- Pros and cons of AO vs. OO, when both the types of software requirements are provided, which have, and respectively have not, a transversal impact on the software architecture, for development from the scratch and maintenance interventions, respectively.
- The impact of AOP on readability, comprehensibility, efficiency, and testability.
- Debugging and static analysis of AOP vs. OOP software applications.
- In what extent, AOP extension points can be utilized to implement and manage dependence relationships between use cases.
- In what extent, the application's characteristics, e.g. structure, naming conventions, etc.) have influence on the utility of applying AOP rather than OOP.

REFERENCES

Baccan M., 2004. Introduzione alla programmazione AOP con AspectJ. In *Webbit*, (Slides), May 2004, http://www.baccan.it/webbit2004/aspectj.pdf, last access 7.04.2007 (in Italian).

Basili V., G. Caldiera, and D. Rombach, 1994. *Goal/Question/Metric Paradigm.* Encyclopedia of Software Engineering, John Wiley & Sons, Vol. 1, pp. 528-532.

AOP, 2007. Alliance, http://aopalliance.sourceforge.net/, last access 7.04.2007

AOP, 2007. JavaWorld, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html, last access 7.04.2007.

Bonin H. E. G. , 2002. Aspect-Oriented Software Development: A Little Guidance to better Java Applications, *University of Applied Sciences, Luneburg*, Germany, May 2002-Nov. 2006. http://as.uni-lueneburg.de/publikation/aosdall.pdf, last access 7.04.2007.

RCOST, 2007. Aspect Oriented Software Development: Analisi e Disegno. In *Research Center On Software Technology*,*Benevento*,Italy,(*Slides*), http://www.ing.unisannio.it/dilucca/LSISW/materiale0506/aosd-aoad.pdf, last access 7.04.2007 (in Italian).

TAODS, 2003. Aspect Oriented Programming. *TAODS'03, Ed. Technerdokan B., DCE, Bilken Univeristy, Amkara*, Turkey, http://trese.cs.utwente.nl/taosad/Papers/TAOSDProceedings.pdf , last access 7.04.2007.

AOSD-Europe, 2007. http://www.aosd.net/, last access 7.04.2007.

AOSD-Europe Project, 2007. http://www.aosd-europe.net/, last access 7.04.2007.

Eclipse org., 2007a. AspectJ Development Tools Project (AJDT), http://www.eclipse.org/ajdt/, last access 7.04.2007.

Eclipse org., 2007b. AspectJ Documentation and Resources, http://www.eclipse.org/aspectj/docs.php 7.04.2007.

Eclipse org., 2007c. AspectJ Project, http://www.eclipse.org/aspectj/, last access 7.04.2007.

Eclipse org., 2007d. AspectJ Programming Guide, http://www.eclipse.org/aspectj/doc/released/progguide/index.html, last access 7.04.2007.

Keuler T., M. Naab, 2007. Supporting Architectural Design by Early Aspects Identification, *Fraunhofer IESE, Kaiserslautern*. Germany, http://dawis2.informatik.uni-essen.de/events/AOM_AOSD2006/Keuler.pdf, last access 7.04.2007.

Kiczales G., Lamping J., Mendhekar A., Maeda ., Lopes ., Loingtier J.-M., and Irwin J, 1997. *Aspect-Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming, http://citeseer.ist.psu.edu/~cache/papers/cs/1860/http:zSzzSzwww-sal.cs.uiuc.eduzSz~kaminzSzdslzSzpaperszSzkiczales.pdf/kiczales97aspectoriented.pdf, last access 7.04.2007.

Laddad Ramnivas, 2002. Separate software concerns with aspect-oriented programming, *Java World*, Jan. 2002, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html

Merlo F., M. Miraz: Aspect-Oriented Programming, 2007. www.elet.polimi.it/upload/ghezzi/_PRIVATE/AOP.pdf, last access 7.04.2007 (in Italian).

Merlo F., M. Miraz, 2007. Aspect-Oriented Programming (*Slides*), www.diit.unict.it/~acalva/SE/slides/se2/aop_slides.pdf, last access 7.04.2007.

ObjectWay Gruppo, Aspect-Oriented Programming and AspectJ, (*Slides*), http://it.sun.com/eventi/jc04/presentazioni/parallela2/objectway.pdf, last access 7.04.2007 (in Italian).