XP + AOP = Better Software?

Michael Kircher
Siemens AG
Corporate Technology, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
Michael.Kircher@mchp.siemens.de

Prashant Jain
Siemens AG
4-A Ring Road, I.P. Estate
Delhi, 110002
India
Prashant.Jain@mchp.siemens.de

Angelo Corsaro
Electrical and Computer
Engineering Department
University of California, Irvine
CA 92697
corsaro@ece.uci.edu

ABSTRACT
Aspect Oriented Programming (AOP) [5] *is a paradigm that enables clean modularization of crosscutting concerns. AOP facilitates extensible architectures without requiring major refactoring of code.*

This paper presents a theoretical study about the influence of AOP on eXtreme Programming (XP) [3]. The paper analyzes the effect of AOP on the XP principles, values, and practices, and whether it makes sense for projects using XP methodology to introduce AOP.

Keywords
Extreme Programming, Aspect Oriented Programming

1 INTRODUCTION

Agile and lightweight software development methodologies are increasingly enabling development scenarios in which risk of failure can be assessed earlier and more easily. These methodologies make it possible to cope with rapidly changing requirements, thus enabling software projects to be re-targeted easily. Extreme Programming (XP) is one of the leading development methodologies in the agile process arena.

To enable the flexibility required by agile methodologies, the software that is written in these kinds of projects, more than in any other, has to be amenable to change. This mandates that the software be reusable, extensible, and flexible. Object-orientation (OO) provides some of the key concepts and mechanisms that facilitate software reusability and extensibility.

OO techniques work great in encapsulating concerns and responsibilities using artifacts such as a class. A concern includes a property or an area of interest such as security and quality of service. Using OO techniques for encapsulating concerns works fine as long as the concerns are isolated. However, often there are concerns that span multiple classes. Such concerns are commonly referred to as crosscutting concerns. For example, concerns such as tracing, security and logging are crosscutting concerns since they typically span multiple classes. Traditionally most of the OO Languages like C++ and Java have lacked any mechanism for encapsulating crosscutting concerns. For example, a concern like tracing is typically distributed over several, if not all, classes of a system. The lack of support for encapsulating crosscutting concerns in OO languages typically leads to tangled code that can become hard to maintain, debug and extend.

Some languages like Common Lisp Object System (CLOS) [2] contain a Meta-Object Protocol (MOP) that makes it possible to encapsulate separation of concerns. However, using such languages is usually quite complicated. Aspect Oriented Programming (AOP) provides a programmatic and encapsulated way of expressing crosscutting concerns. It can complement OO techniques in producing code that includes crosscutting concerns. Therefore, agile development process in general and eXtreme Programming (XP) in particular can take advantage of AOP. This paper presents an analysis of the influence of AOP on XP.

Section 1 of this paper presents an overview of XP and AOP. Section 2 elaborates on the advantages and disadvantages of combing the AOP paradigm with the XP principles, values, and practices. Finally, section 3 presents our conclusion.

eXtreme Programming

eXtreme Programming (XP) [3] is a lightweight methodology that has gained increasing acceptance and popularity in the software community. XP promotes a discipline of software development based on principles of simplicity, communication, feedback, and courage. It is designed for use with small teams who need to develop software quickly in an environment of rapidly changing requirements.

XP uses effective practices such as Refactoring, Pair Programming, and Continuous Integration. Each practice provides an important benefit to the development cycle. For example, Refactoring provides a gradual change of source-code to a more adaptable design. Pair Programming allows two programmers to share their thoughts and know-how working in front of a common screen. Finally, Continuous Integration ensures that there is always a running system executing all tests successfully.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a paradigm, which enables separation of concerns, and provides a clean way of encapsulating crosscutting concerns. The rationale behind AOP is that computer systems are better programmed by separately specifying and implementing the various concerns of the system and some description of their relationships. The mechanisms in the underlying AOP environment are responsible for weaving or composing the different concerns into a coherent program.

Concerns can range from high-level notions like secu-

rity and quality of service to low-level notions such as logging, caching, buffering and so on. They can be functional, like features or business rules, or nonfunctional, such as synchronization and transaction management.

The problem is that with conventional programming languages, there are certain concerns that are very hard to encapsulate in a single programming language entity (e.g., a class or function). These concerns are known as crosscutting concerns. Crosscutting concerns make programs harder to read, maintain, understand, and reuse. AOP focuses on mechanisms that enable clean modularization of crosscutting concerns. For example, using AOP the functionality of tracing can be treated as a concern and factored out of existing code.[1] Tracing serves a common purpose in the application but crosscuts multiple classes. Therefore, using AOP, the tracing code can be factored out from all the classes into an aspect. The locations in the application code from where the tracing code is factored out are known as join-points. Once the tracing code has been factored out, all the join-points are declared in a file using a special notation. A tool such as AspectJ [1] is then used to weave the aspect code that is the tracing code, at the join-points. However, the actual application code stays independent of any tracing code. Thus there is a clear separation of application logic and crosscutting concerns such as tracing.

2 COMBINING AOP AND XP

While XP is a methodology affecting overall software design and development, AOP is a technique aimed at separating and implementing crosscutting concerns. Therefore, although at first glance the two may not appear to have much in common, they do share a well-defined intersection where AOP can influence the way XP is done.

One of the key ideas at the heart of XP is refactoring. (Please note that it is beyond the scope of this paper to describe how to refactor crosscutting conerns into aspects.) Refactoring offers numerous benefits to software development and is therefore strongly supported by XP. However, refactoring is often a tedious process involving many repetitious steps of changing and testing code.

AOP, on the other hand, can allow for extensible architectures without major refactorings. Therefore, adding the AOP paradigm to the XP methodology can provide several benefits to system developers.

This section presents an analysis of the effect of AOP on the principles, values, and practices of XP as described in [3].

In the following discussion we assume that programmers can generally handle the complexity introduced by AOP.

[1] Note, that aspects can also be created from scratch.

Effect of AOP on XP Principles

The effect of AOP on XP principles is discussed below:

Assume simplicity – Introducing aspects separates the concerns, so that it gets simpler to understand the architecture. Since it is usually straightforward to work in terms of aspects, AOP therefore supports the programmers in their approach of assuming simplicity.

Embrace change – Aspects support change naturally since they make changing application code less intrusive than usual code changes. In addition, using AOP allows for the introduction of crosscutting concerns into the application code without requiring refactoring of the application structure.

AOP has no effect on the following XP principles: *Rapid Feedback*, *Incremental Change*, and *Quality Work*.

Effect of AOP on XP Values

The effect of AOP on XP values is discussed below:

Simplicity – Simplicity is improved with respect to the structure of the application code. Crosscutting concerns are untangled into aspects, which makes the code simpler to understand. In order to fully leverage this, however, strong support in visualization is needed to keep track of where the aspects are applied.

Communication – Since an aspect addresses crosscutting concerns, it can serve as an excellent source of documentation. In fact, using AOP can enhance communication and understanding of the code among developers.

AOP has no effect on the following XP values: *Courage* and *Feedback*.

Effect of AOP on XP Practices

The effect of AOP on XP practices is discussed below:

Simple Design – AOP leads to simple design by incorporating separation of concerns. The increased modularity of the code makes it easier to enhance the software and make modifications.

Studies [8] are under way to investigate the benefits and liabilities of using architectural means, versus aspects to develop certain features. Experience and practice will show what succeeds in the future.

Collective Ownership – Aspects help reduce the need for collective code ownership. This is because using aspects helps separate tangled code and thus allows a more refined separation of responsibilities of programmers. However, even with such a separation of responsibilities, the need for collective ownership cannot be totally eliminated. This is also true since aspects can cause global changes in weaved code, which again leads to collective ownership. One of the key benefits of collective ownership is the flexibility and the ability to deal with having a programmer become unavailable.

Another benefit of collective ownership is that it addresses the common need of a programmer to touch

multiple parts of a system. Typically, when a new feature is added to a system, multiple parts of the system are modified. Collective ownership of code among programmers makes it easier to modify multiple parts of the system. However, if the project were using AOP then the need for collective ownership would be weakened. This is because using AOP, the aspects untangle code that earlier forced developers to touch multiple parts of the system. Using AOP the developers would ideally touch only a single aspect to introduce a new feature to the system.

If AOP is introduced into a project that uses XP as a methodology, it is important that every programmer becomes familiar with AOP – slightly contradicting the value of Simplicity. If programmers were unable to understand AOP code, they would not be able to understand the code enough to change it. One suggestion could be to use pair programming techniques for communicating AOP knowledge within the team.

Refactoring – AOP can supplement tedious refactorings since AOP can add functionality that the original code is not prepared to do so. Refactoring can of course be applied at both levels, the actual source code and the aspects themselves as well.

On the other hand, aspects, as currently implemented by AspectJ can be misused for patching, instead of properly addressing the actual problem of separating entangled code blocks. Misusing aspects for patching undermines the principle of OO. For example, if a programmer realizes a method is incorrectly implemented, the programmer could redirect the control flow to a new method replacing the broken one. This surely might be convenient, but the derived code becomes so cumbersome that its maintenance will be a nightmare.

Testing – AOP supports testing in many ways such as by providing support for factoring out test code that is commonly glued to the application code. This can reduce the footprint and likelihood of unused code (dead code) causing any problems at run-time.

AOP can also be used for specialized test cases and spike tests as well as to develop test cases that include combining multiple test cases. For example, using AOP, test cases can be easily created that combine testing of different input values to variables. This can include testing various extreme (maximum, minimum) values as well as invalid values to see if the code is able to handle them. The test cases can be developed for testing the variables independently or in combination. Therefore, using AOP the number of test cases can be greatly varied and extended. Also, using AOP measurement and instrumentation are much easier and less intrusive to implement.

While AOP helps to automate tedious testing, the complexity introduced by the concept alone might be too overwhelming for programmers. Therefore, all programmers including testers that are involved in the project should become familiar with AOP.

Small Releases – The differences between small releases can be viewed as a set of aspects. This is related to the concept of Multi-dimensional Separation of Concerns (MDSOC) [6]. MDSOC provides many features including modeling of separation of concerns among releases. The rationale behind MDSOC is that any criterion for decomposition is appropriate for some contexts, but not for all. Similarly, AOP can provide a nice way of capturing differences between releases.

Continuous Integration – AOP can make continuous integration more difficult since with AOP it is necessary to determine which aspects to weave in and which not for every integration step. Nevertheless, AOP can be used for providing a flexible means of integrating configurable behavior.

Many projects require systems that are highly configurable. As a result, such systems require transparent removal of behavior that is not used or is not desirable. For example applications running in a single-threaded environment do not require synchronization. AOP can be used to allow for such flexibility.

However, even though AOP can make continuous integration more flexible, using it makes the code more susceptible for configuration errors or bugs that only occur under certain combinations of aspects.

Coding Standards – Because XP has no explicit design or architecture phase, it is important that implementation choices, that have a critical impact on the quality of the software, get coordinated by coding standards. For example, while using XP, the coding standards may require the usage of Guarded Locking [9] instead of using locks directly. Similarly, when integrating AOP with XP, the coding standards would need to require the usage of AOP.

Coding standards need to specify how concerns should be separated. In fact, coding standards for some concerns such as initialization or logging can be enforced by aspects since they focus exactly on these issues. In certain cases, AOP actually simplifies coding standards. For example, tedious tasks such as the coding of object factories can be taken over by aspects, which can then implement such functionality [7].

AOP has no effect on the following XP practices: *Pair Programming*, *Planning Game*[2], *Metaphor* and *On-Site Customer*.

To make effective use of AOP with XP, it is important

[2] A technology that is used in implementing a system influences the way that planning is done since the technology usually has a certain overhead associated with it. In the case of AOP the same rule applies - AOP affects implementation of a system and hence the Planning Game. However, the extent of the influence is similar to those of other technologies. Therefore for the purpose of this paper, we assume that AOP has no relevant impact on the Planning Game.

that the usage of AOP is properly controlled. One way to control it is by using the Coding Standards to provide the usage of AOP in the system.

3 CONCLUSION

AOP provides a programmatic and encapsulated way of expressing crosscutting concerns that is usually missing in OO languages. It can therefore complement XP methodology giving developers a powerful tool to take advantage of. As discussed in this paper, using AOP benefits many values, practices, and principles of XP. However, there are some areas where combining AOP and XP requires careful thought and planning. The following are some recommendations to developers who plan to introduce AOP in their systems that use an XP methodology.

Knowledge of developers – Since AOP affects all parts of a system; it requires most, if not all, developers to be knowledgeable of AOP. This in turn requires that the developers be willing to adopt a new paradigm and educate themselves on how to use both AOP and XP effectively.

Extensive use of AOP – AOP can be beneficial, but only if it is used in the entire software. Using AOP for only single tasks, such as testing, or implementing a single feature, will lead to additional overhead. The more AOP is used in a project, the more it will be worth investing the initial learning overhead.

Using right tools – If AOP is used to encapsulate crosscutting concerns into aspects, it becomes essential to use the right visualization tool to view how and where the aspects influence the base code. Therefore, it is important that the right set of tools is available to help the developers integrate AOP with XP. For example, if AspectJ is being used, there are extensions available for IDEs such as Emacs/XEmacs, JBuilder and Forte that provide visualization of aspects.

Awareness of changes – AOP influences many XP values, practices and principles. It is therefore necessary that developers be aware of all the changes that are introduced into the system as a result of using AOP. In particular, since AOP can result in crosscutting changes that affect a large part of the system, each change can in reality affect several developers working on the system.

Moreover, since AOP is still in its seminal phase, a lot of research has to be done in order to develop a complete understanding of the benefits and of the idioms behind AOP.

Just as OO techniques needed time to be mastered, the same applies to AOP. The role of researchers is very important since they need to provide examples and guidelines of good use of aspects, similar to those that exist for OO technology. Patterns, such as those documented in [4][9], are one form to share such experiences. Organizing a workshop at one of the pattern conferences, such as PLoP or EuroPLoP, could be one way to further investigate this.

This paper made a first step in bringing the techniques of AOP and XP closer together. Future work should include practical evaluation in real applications to prove the effectiveness of this analysis.

Thanks to Klaus Ostermann for providing us with excellent feedback on earlier versions of this paper.

REFERENCES

1. *AspectJ* homepage, http://aspectj.org, 2002
2. D. Bobrow, R. Gabriel, and J. White, *CLOS in Context*, 1991
3. K. Beck, *Extreme Programming Explained: Embrace Change,* Addison Wesley Longman, Inc., 1999
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, 1995
5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, *Aspect-oriented programming*. In Proceedings ECOOP'97, LNCS 1241, Jyvaskyla, Finland, Springer-Verlag , 1997.
6. P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, Proceedings of the ICSE'99, May, 1999.
7. K. Ostermann and M. Mezini, *Object Creation Aspects with Flexible Aspect Deployment*, 2002
8. R. Pichler, K. Ostermann, M. Mezini, *On Aspectualizing Component Models*, submitted to European Conference on Object-Oriented Programming, 2002
9. D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture - Patterns for Concurrent and Distributed Objects*, John Wiley and Sons, 2000