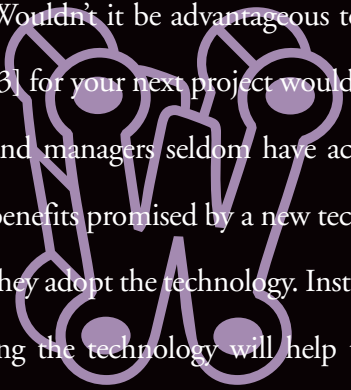
DOES ASPECT-ORIENTED PROGRAMMING WORK?

Determining the best method for evaluating the effectiveness of a new technology.

Gail C. Murphy, Robert J. Walker, Elisa L.A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten



Wouldn't it be advantageous to know in advance that the use of AOP [3] for your next project would be successful? Unfortunately, developers and managers seldom have access to evidence assuring them that the benefits promised by a new technology, such as AOP, will be achieved if they adopt the technology. Instead, they must take a leap of faith, believing the technology will help them overcome problems encountered previously.

To thoroughly evaluate the usefulness of AOP, multiple software development organizations would need to build their products both with and without AOP and compare the results. Such an approach is unrealistic. Is the situation then hopeless? Can software engineering researchers provide any help to determine if it is beneficial for software development organizations to adopt AOP for building their software products? We believe researchers can provide help. Toward that goal, a

number of studies have been conducted to assess the usefulness of AOP and similar technologies.

There are two basic techniques for assessing a programming technology: experiments and case studies. Experiments provide an opportunity for direct comparison, allowing researchers to investigate how the technology performs in detailed ways. Case studies take many forms, from small-scale comparisons to longitudinal studies of the technology in action. Case studies tend to provide broader knowledge about the use of a technology.

Experiments: Debugging and Change

We conducted two semicontrolled experiments to investigate whether AOP, as embodied in AspectJ, made it easier to develop and maintain certain kinds of application code [5, 8]. One experiment considered whether AspectJ enhanced a developer's ability to find and fix faults present in a multithreaded program. The second experiment focused on the ease of changing a distributed system. Each experiment involved three trials that compared the actions and experiences of two groups: one group worked with AspectJ; the other group worked with code implemented in a control language—Java for the first experiment and Emerald [1] for the second experiment.

An early version of AspectJ, which had two domain-specific aspect languages, was used for these experiments. The COOL aspect language expressed synchronization code; the RIDL aspect language specified code related to remote data transfer in a distributed program.

The first experiment showed that when locating a fault involved reasoning within a single class or an aspect, programming teams using AspectJ were able to correct a seeded program fault faster than the Java teams. Furthermore, teams using AspectJ had fewer discussions about the semantics of the base code. The second experiment showed that the RIDL aspect language did not allow developers to implement changes to a distributed program faster than developers working in the control language. We attributed this result to the fact that the developers did not spend enough time analyzing the code prior to beginning coding; they were led astray by assuming the change could largely be implemented using the aspect language.

Analyzing the results of these two experiments led us to two key insights. First, the range of effect the aspect code has on the system code matters. The COOL code affected limited, discernable parts of the rest of the system: The participants could look at COOL code and understand its effect without having to analyze vast parts of the rest of the code. The RIDL code did not have this property. Second, the presence of aspects in the code changed the way in which participants tackled the tasks. Participants first looked for a solution that could be modularized in an aspect. When this solution was appropriate, AOP was beneficial. However, when the solution could not be encapsulated within an aspect, the participants took longer to reach a solution.

We also conducted both large and small case studies to investigate the usefulness of AOP. A large case study affords the opportunity to examine how a programming technology affects multiple issues concerning the building of a specific system, such as the design of the system and the development process. Smaller case studies afford the opportunity to investigate specific development issues on smaller systems.

Case Study: Web-based Learning Environment. Atlas is a Web-based learning environment we implemented as a multitier distributed application with AspectJ 0.2 [2]. Atlas allows students to register for courses and to navigate through course material using personalized views. Atlas comprises approximately 11,000 lines of code spread over 180 classes, and 17 aspects. The versions of AspectJ that were used supported a general-purpose aspect language.

Atlas uses aspects for several different purposes. One set of aspects supports different tier configurations in which Atlas can run, such as on an application server or as an applet. Other aspects encapsulate design patterns. A final set of debugging and tracing aspects supports the development process.

This case study taught us several lessons about designing and programming with aspects. For example, we learned one must carefully consider the different kinds of "knows-about" relationships between classes and aspects. As another example, we found it useful to use aspects as factories to simplify the extension of an object's behavior.

Case Study: Comparing Separation of Concern Technologies. We have also undertaken a small, comparative case study in which three different mechanisms for separating concerns—AspectJ, Hyper/J, which supports the hyperspaces concept [7], and a lightweight, lexical means of separating code—were applied to pieces of two existing programs: `jFTPd` and `gnu.regexp` [6]. Using these mechanisms, we attempted to separate features discernible by users, such as whether or not the regular expression package could match expressions over lines.

Applying the three mechanisms to the same programs and concerns allowed us to compare the effect of each mechanism on the coding of concerns, the restructuring necessary to expose and extract a concern, and the process required to perform the separation. Not surprisingly, no single mechanism was

perfect: Each had a different set of tradeoffs. For example, some mechanisms are likely easier than others to use when concerns are to be developed independently by different teams.

Insights

Reflecting on the results of the assessment efforts we have performed, three areas emerge as important in supporting the use of AOP:

- *Exposing join points*. Many concerns in a system will likely not be designed from scratch as aspects. Rather, many concerns will emerge as a system evolves. From our experience, capturing such concerns as aspects requires restructuring of the base code to expose suitable join points. For instance, we have extracted code from existing methods into a new method to expose a method-level join point. Tools to help in the restructuring would make it easier to introduce aspects into an existing system.
- *Managing aspect interfaces*. We have observed that it is easier to build and debug an AOP system when the interface between the aspects and the base code is narrow and unidirectional. By narrow, we mean the aspect code has a well-defined effect on particular points in the code. By unidirectional, we mean the aspect code refers to the base code but not vice versa. Lippert and Lopes have made a similar observation based on a study of using AspectJ to capture exception-handling constructs [4]. They have suggested that tool support for showing the local effect aspects have on classes might make it easier to use aspects.
- *Structuring aspects*. In several cases, we have found it easier to understand and manage aspects when the aspect code forms the glue between two object-oriented structures. For instance, when capturing a user-level feature as an aspect, we have found it beneficial to express the feature in its own object structure and to use an aspect to inject that feature into the base code. This style should be investigated further as AOP design guidelines are developed.

What Next?

Does AOP work? Based on the results of the experiments and case studies we have conducted, AOP shows promise. We have determined particular situations where AOP benefits developers, and we have characterized "aspects" of how AOP has worked, presenting design and process guidelines we have found helpful in developing AOP systems.

However, we have much left to learn about AOP. Does it work for large, multideveloper projects? To what kinds of problems is it best suited? What kinds of constructs are most usable for specifying crosscuts? To answer these questions, more experiments and studies are needed. In particular, as AOP mechanisms stabilize, longitudinal industrial case studies are needed to better qualify and quantify the benefits of AOP.

Assessing a new technology is not without its difficulties, but the effort of assessment is worthwhile because at least three separate groups of users can benefit. Early adopters can use the results to decide when to make a trial effort with the technology and can benefit from design and other guidelines resulting from the assessment activities. Researchers can build upon the assessment approaches used to broaden and deepen the studies conducted, and AOP technology developers can determine which parts of the technology are proving beneficial. C

References

1. Black, A., Hutchinson, N., Jul, E., and Levy, H. Object structure in the Emerald system. *ACM SIGPLAN Notices 21*, 11 (Nov. 1986).
2. Kersten, M.A. and Murphy, G.C. Atlas: A case study in building a Web-based learning environment using aspect-oriented programming. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, (Nov. 1999).
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.J. and Irwin, J. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming,* (June 1997).
4. Lippert, M. and Lopes, C.V. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering* (June 2000).
5. Murphy, G.C., Walker, R.J., and Baniassad, E.L.A. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering 25,* 4 (Jul./Aug. 1999).
6. Murphy, G.C., Lai, A., Walker, R.J. and Robillard, M.P. Separating features in source code: An exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering*, (May 2001).
7. Tarr, P., Ossher, H., Harrison, W., and Sutton, S.M. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, (May 1999).
8. Walker, R.J., Baniassad, E.L.A., and Murphy, G.C. An initial assessment of aspect-oriented programming. In *Proceedings of the 21st International Conference on Software Engineering* (May 1999).

Gail C. Murphy (murphy@cs.ubc.ca) is an associate professor in the Department of Computer Science at the University of British Columbia in Vancouver, Canada.

Robert J. Walker (walker@cs.ubc.ca) is a Ph.D. candidate in the Department of Computer Science at the University of British Columbia in Vancouver, Canada.

Elisa L.A. Baniassad (bani@cs.ubc.ca) is a Ph.D. candidate in the Department of Computer Science at the University of British Columbia in Vancouver, Canada.

Martin P. Robillard (mrobilla@cs.ubc.ca) is a Ph.D. student in the Department of Computer Science at the University of British Columbia in Vancouver, Canada.

Albert Lai (alai@cs.ubc.ca) is a Master's student in the Department of Computer Science at the University of British Columbia in Vancouver, Canada.

Mik A. Kersten (mkersten@parc.xerox.com) is a member of the research staff at Xerox PARC's Computer Science Laboratory.

This work was supported in part by grants from NSERC, Xerox, and IBM.