Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe

Awais Rashid, Thomas Cottenier, Phil Greenwood, and Ruzanna Chitchyan, *Lancaster University, UK*

Regine Meunier, *Siemens AG, Germany*

Roberta Coelho, *Federal University of Rio Grande do Norte, Brazil*

Mario Südholt, *École des Mines de Nantes, France*

Wouter Joosen, *Katholieke Universiteit Leuven, Belgium*

Aspect-oriented software development techniques provide a means to modularize crosscutting concerns in software systems. A survey of industrial projects reveals the benefits and potential pitfalls of aspect-oriented technologies.

The past decade has seen the increased use of aspect-oriented software development (AOSD) techniques[1] as a means to modularize crosscutting concerns in software systems, thereby improving a development organization's working practices and return on investment (ROI). Numerous industrial-strength aspect-oriented (AO) programming frameworks exist, including AspectJ, JBoss, and Spring, as do various aspect-oriented analysis and design techniques.[2] The "Major Industrial Projects Using AOSD" sidebar highlights notable applications of AOSD, of which the most prominent is the IBM WebSphere Application Server.[3]

Developers considering AOSD techniques must ask three fundamental questions:

- *How is AOSD being used in industrial projects today?* Developers must determine whether AOSD techniques are suited to the problem at hand and the particular project context.
- *Does the improved modularity yield real benefits when engineering and evolving software?* Developers must understand whether the potential benefits outweigh the costs of introducing a new technology and, if so, be able to convince management of its long-term profitability.
- *What do developers need to be aware of when using AOSD techniques?* Developers must avoid known pitfalls and deploy design strategies and tools to help counter their potential threat to product quality.

Answers to these questions are not readily available, and gleaning knowledge from existing literature on the topic is difficult, but we have obtained some insights by analyzing several medium- and large-scale projects employing AOSD techniques. These projects have been accessible to us both directly within AOSD-Europe (www.aosd-europe.net), a large-scale academia-industry collaboration funded by the European Commission since 2004, as well as indirectly through its liaison channels with interested researchers.

Our experience indicates that production (as opposed to pilot) projects mainly rely on basic features of AO languages to modularize well-known crosscutting problems; developers introduce AOSD concepts incrementally,

MAJOR INDUSTRIAL PROJECTS USING AOSD

IBM WebSphere Application Server is a Java application server that supports Java Enterprise Edition (EE) and Web services. WebSphere is distributed in various editions that support different features, with AspectJ (www.eclipse.org/aspectj) used to isolate these features.

JBoss Application Server (AS) is a free, open source Java application server that supports Java EE. The core of JBoss AS uses JBoss AOP (www.jboss.org/jbossaop) to deploy services such as security and transaction management.

Oracle TopLink is a Java object-to-relational persistence framework that is integrated with the Spring application server (www.springsource.org). TopLink achieves high levels of persistence transparency using Spring AOP.

Sun Microsystems uses AspectJ to streamline mobile application development for the Java Micro Edition (ME) platform. Aspects are used to simplify the development of mobile applications for deployment to different operator decks and different mobile gaming community interfaces.

Siemens' Soarian is a health information system (HIS) that supports seamless access to patient medical records and the definition of workflows for health provider organizations. Soarian uses AspectJ and fastAOP (http://sourceforge.net/projects/fastaop) to integrate crosscutting features into an agile development process.

Motorola's wi4 is a cellular infrastructure system that provides support for the WiMax wireless broadband standard. The wi4 control software is developed using WEAVR (an aspect-oriented extension to the UML 2.0 standard) for debugging and testing.

ASML, a provider of lithography systems for the semiconductor industry, uses Mirjam, an aspect-oriented extension to C, to modularize tracing, profiling, and error-handling concerns.

Glassbox is a troubleshooting agent for Java applications that automatically diagnoses common problems. Glassbox Inspector uses AspectJ to monitor the Java virtual machine's activity.

MySQL is a widely used relational database management system. The logging feature in MySQL is implemented using AspectJ.

initially addressing developmental concerns and not core product features. In addition, AOSD techniques improve design stability over a system's evolution and can substantially reduce design model size. Finally, pointcut fragility in mainstream AO programming technologies can cause ripple effects during system evolution, with aspects unintentionally influencing program exception flows. This requires developers to carefully consider known bug patterns and introduce effective testing strategies.

TYPICAL USES OF AOSD IN INDUSTRY

Table 1 summarizes the projects we studied.

Basic AO features

Most of the industrial projects use basic AO features. The biggest hurdle to overcome in adopting such features is their incorporation into existing development processes. Crosscutting concerns are a fairly well-understood problem—most architects and developers must regularly manage the complexity of tracing, auditing, persistence, and so on. While the potential of AOSD techniques for this purpose is recognized, their introduction into the development process is nontrivial.

A typical example is Soarian, a large-scale hospital information system developed by Siemens using AspectJ and fastAOP in an agile development context.[4] In this case, a team of AO programmers initially had to be trained. The training comprised five one-hour sessions. Given the limited amount of time, the focus naturally was on mastering basic AO concepts to keep the learning effort low, allowing team members to see the benefits of using AOSD techniques without the need to become experts. AO programming evangelists helped introduce advanced features when the need occurred.

The other projects listed in Table 1 exhibit a similar pattern, as the constraints on introducing any new technology into the development process are comparable. Team leaders and managers must be convinced of AOSD's benefits while ensuring that substantial effort is not deflected from existing development activities.

The most frequently used AO features in industrial projects tend to be those that are relatively simple to apply, yet provide a high dividend. Examples are call and execution pointcuts. Inter-type declarations, which allow the introduction of new methods or attributes as well as inheritance and interface implementation links, are another typical usage; developers can easily see the benefits of modularizing the static structure of their programs to cater to crosscutting concerns. Another common feature is the ability to statically declare global constraints on the program that the system can check at compile time to raise a warning or error. These `declare error` and `declare warning` features (available in AspectJ) enforce architectural constraints; both Soarian and WebSphere use them as part of the build process.

Advanced AO features

Advanced AO features and new tool and language prototypes are mainly used in industry-academia pilots or controlled lab experiments for three reasons.

First, advanced features and new prototypes typically require expert guidance from the R&D personnel engaged in their development.

Second, incorporating such features into a shipped product can be very risky, as is true for any untried technology. In the Soarian project, even for a product as well-established as AspectJ, developmental concerns such as validation of architectural constraints had to be resolved first to show its potential. Only after the tools' reliability had been demonstrated was aspect-orientation incorporated into core product development.

Table 1. Overview of industrial AOSD projects studied.

Project	Domain	Size/complexity	Type	AO technologies used	AO features used	Crosscutting concerns modularized
Soarian (Siemens)	Healthcare	300 modules, 400 developers, 500 concurrent users per server	Industrial	AspectJ, fastAOP	Basic	Architecture validation, caching, auditing, performance monitoring
IBM WebSphere	Enterprise systems	WebSphere Product Center—3,700 classes, 43 aspects; Enterprise Service Bus—1,300 classes, 6 aspects	Industrial	AspectJ	Basic	Tracing, logging, first failure data capture
MOTOwi4	Telecom	12 components, 100 developers, 200,000 sessions per server	Industrial	WEAVR	Basic	Tracing, timeout
Toll system (Siemens, Lancaster University, EMNantes)	e-transport	90 classes, 20 aspects	Industry-academia	AspectJ, AWED, EA-Miner	Advanced	Charging, error handling, persistence, logging, monitoring, compatibility, security
Wit-Case (Katholieke Universiteit Leuven, SSEL-VUB, Alcatel-Lucent)	Telecom	8 Prolog modules, 1,200 lines of code, 1 technical architect, 2 developers	Industry-academia	Padus	Advanced	Fixed billing, duration billing, platform customizations, logging
Health-Watcher (Recife, Brazil)	Healthcare	130 modules, 3 developers, 1.5 million people served	Controlled experiment	AspectJ, CaesarJ	Advanced	Concurrency, distribution, exception handling, persistence, design patterns
AJHotDraw	Graphics	279 classes, 31 aspects, 1 AO developer, 25 OO developers	Controlled experiment	AspectJ	Basic	Persistence, design policies, contract enforcement, Undo command
MobileMedia	Imaging	3 developers, 51 classes, 36 aspects; a software product line for cell phones	Controlled experiment	AspectJ	Advanced	Exception handling, alternative and optional features

Third, industry adopters typically develop experimental solutions that evaluate the core know-how that the research community has generated, then they customize these core principles to create technology to meet the business needs of a specific industrial partner. Such an experiment provides not only a practical validation of general principles but also a concrete vehicle for further experimentation and practical evaluation and, last but not least, inspiration and insight for the challenges ahead.

This trend is evident in the various industry-academia pilots listed in Table 1. For example, in the Wit-Case (Workflow Innovations, Technologies, and Capabilities for Service Enabling) project, which partnered Katholieke Universiteit Leuven, Vrije Universiteit Brussel's System and Software Engineering Lab (SSEL-VUB), and Alcatel-Lucent, industry requirements for composition in workflow languages drove the development of an AO extension to WS-BPEL, Padus.[5] The project provided insights into the feasibility of AO concepts in the context of workflow systems.

A prototype toll system jointly developed by Siemens, Lancaster University, and the École des Mines de Nantes (EMNantes) used AWED (Aspects with Explicit Distribution),[6] a language and runtime system for distributed aspects, to flexibly manage toll processing pipeline distribution and synchronization issues, including toll data collection, processing, and billing functionality. A comparison with an object-oriented (OO) solution showed widespread code replication and scattering of distribution functionality within the OO toll processing pipeline. In addition to AWED, project developers applied AspectJ to support different data models during toll data processing. The pilot thus provided evidence that developers can exploit synergies by using different AO frameworks in one application to handle different types of crosscutting functionalities in industrially relevant applications.

The toll system also utilized EA-Miner,[7] a tool that uses natural-language processing techniques to identify potential crosscutting concerns in requirements documents. EA-Miner revealed aspectual concerns that the developers had originally overlooked during the AspectJ-based implementation. These concerns included legal constraints, connection/reconnection between onboard units and the back office, transaction management, and communication triggers.

Applying EA-Miner led to the reevaluation of some architectural decisions related to the use of AO technologies. The experience also revealed interesting insights into the potential adoption barriers for AO requirements engineering techniques. Any such adoption would necessitate a change to the established process of requirements document preparation. It would also require additional effort on the part of domain experts to specify aspect compositions, along with the need for evangelists and trainers to introduce the techniques into the development process (in a fashion similar to the Soarian project).

Incremental introduction of AOSD

Our observations about production projects and pilots point to an incremental adoption strategy of AOSD. In both Soarian and WebSphere, for instance, AO programming was first introduced into the build process as a means to enforce architectural constraints. This did not require any aspects to be included in the shipped product, while at the same time providing a solution to a fundamental problem that faces any large development team.

Our observations about production projects and pilots point to an incremental adoption strategy of AOSD.

Soarian, a very large system structured using a layered architecture, offers a good sample case. In this application, certain architectural constraints must be obeyed—for example, data access objects (DAOs) should be accessed only from application tasks, and only the DAOs can use the Java persistence API. However, enforcing such architectural constraints on large development teams is difficult. Soarian uses the AspectJ `declare error` construct to enforce such constraints as part of the build process, using specifications such as:

```
declare error: noATCallsFromDAO():
"A DAO should not call a business layer";
```

WebSphere uses the same technique to prevent unwanted API calls that may lead to unnecessary components being shipped. Such an approach also justifies the additional overhead of introducing a new technology in a broad context. The AO programming solution guarantees a valid architecture by ensuring that each layer only calls other authorized layers or APIs. Further, there is no additional risk as AO programming is not part of the production code.

Once they have established an initial confidence factor, developers can incorporate AO techniques into production code. When this occurs, concerns such as tracing, logging, and performance monitoring take precedence as the crosscutting challenges associated with these concerns are familiar to developers, and the benefits of an AO programming solution are immediately obvious. We see this trend in numerous industrial projects. For instance, Motorola developers used an AO modeling tool, WEAVR,[8] in MOTOwi4 production to modularize tracing for debugging and fine-tune the request timeout mechanism.

Debugging this type of application is particularly difficult because of race conditions that occur between distributed nodes and the asynchronous communication model. These race conditions require correlation of traces produced by the system components, thereby necessitating the coordination of different development teams. A tracing profile in WEAVR provides a configuration interface to declare the granularity and scope of tracing functionality. This information is then used to generate tracing aspects from predefined templates, which are transparently applied to the different system components before code generation.

A recurring communication problem in asynchronous distributed systems arises when the system processes do not define time frames for request responses, which causes the system to be blocked. A request timeout profile is defined in MOTOwi4 that lets engineers configure the timing constraints on the system requests and to declare specific actions to be taken when they are not obeyed. These exception handlers are then transparently deployed in the system by generating aspects according to the profile configuration.

MOTOwi4's debugging and timeout features have minimal interactions with the system's core functionality, and their aspect-based modularization does not require significant changes to the system architecture or the development process.

ROI ON ENGINEERING AND EVOLVING SOFTWARE

To promote smooth adoption of AOSD techniques, it is critical to empirically analyze their benefits and limitations.

Improved design stability

Lancaster University has established an AOSD testbed (www.comp.lancs.ac.uk/~greenwop/tao) as part of AOSD-Europe to provide a set of artifacts including benchmarking applications (both AO and non-AO implementations), metric suites, and previously collected empirical results. From these artifacts, both researchers and practitioners can configure their own empirical studies. Figure 1 shows samples of empirical data from the testbed.

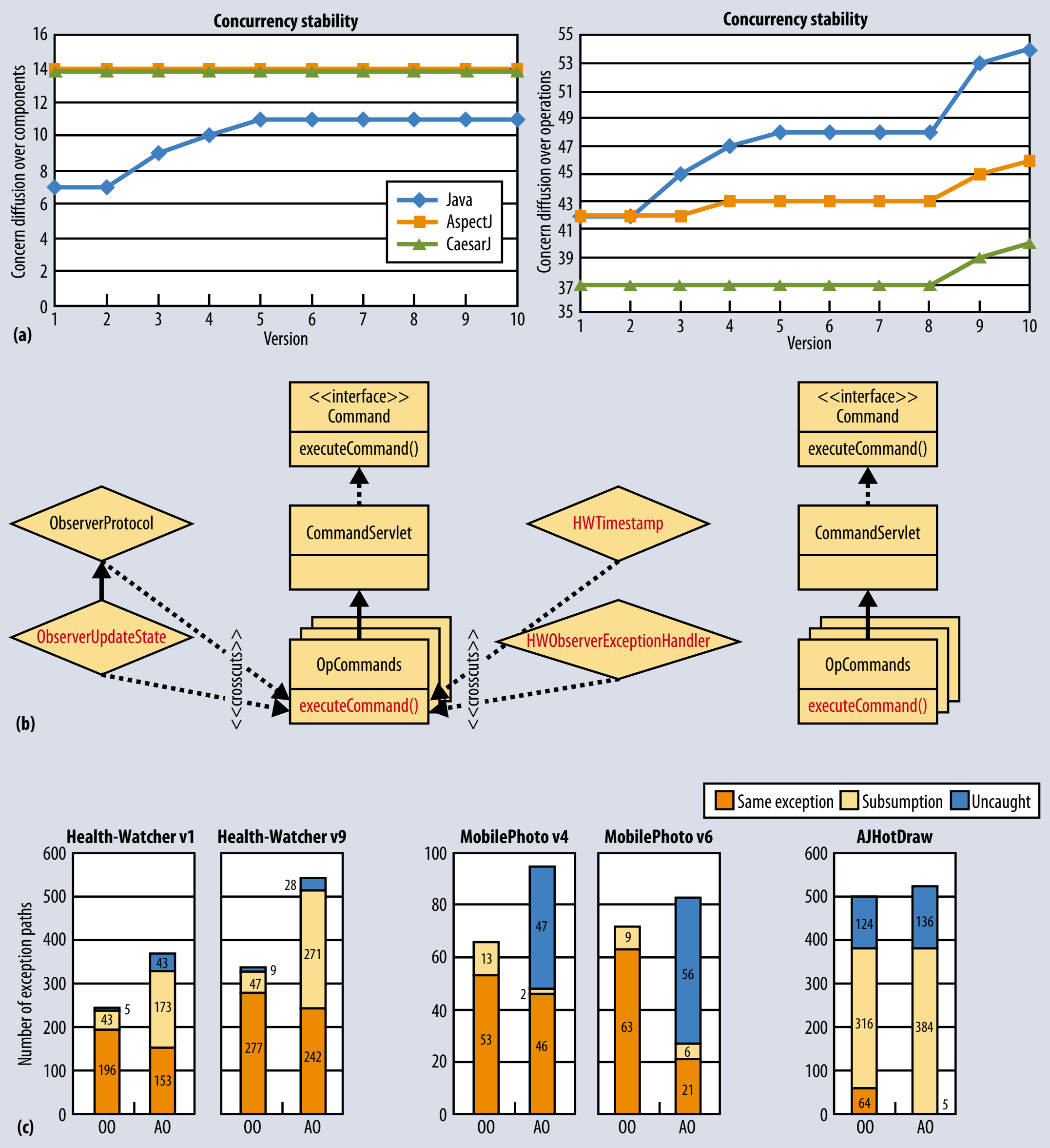

Figure 1. Samples of empirical data from the AOSD testbed at Lancaster University. (a) Improved stability offered by AO for the concurrency concern in Health-Watcher. (b) Increased scope of changes in AO design and more localized changes in OO design of Health-Watcher. The aspects are denoted by a diamond; the modules affected by the change are highlighted in red. (c) Uncaught exceptions, subsumptions, and specialized handlers in AO versus OO implementations of three systems. Note that MobilePhoto is a version of the MobileMedia product line that deals only with images.

One study[9] conducted with these artifacts involved using AspectJ and CaesarJ (http://caesarj.org) to compare the design stability of two AO implementations against an OO implementation (using Java) of Health-Watcher, a system designed to monitor public-health-related complaints and notifications in Recife, Brazil. The study applied numerous common maintenance scenarios to both the AO and OO versions of this benchmark application.

The analysis revealed that concerns modularized up front using AO techniques, such as concurrency (Figure 1a), showed superior design stability, and modifications tended to be confined to the target modules. In addition, the AO designs prevented more intrusive modifications, even when the change focused on a noncrosscutting concern.

The AO designs also respected the open-closed principle more effectively—that is, a module should be open for extension but closed for modification—by enabling modifications to be made through the introduction of new aspect modules rather than modification of existing artifacts. This increased stability is achieved via pointcuts and inter-type declarations, which absorb changes that would otherwise

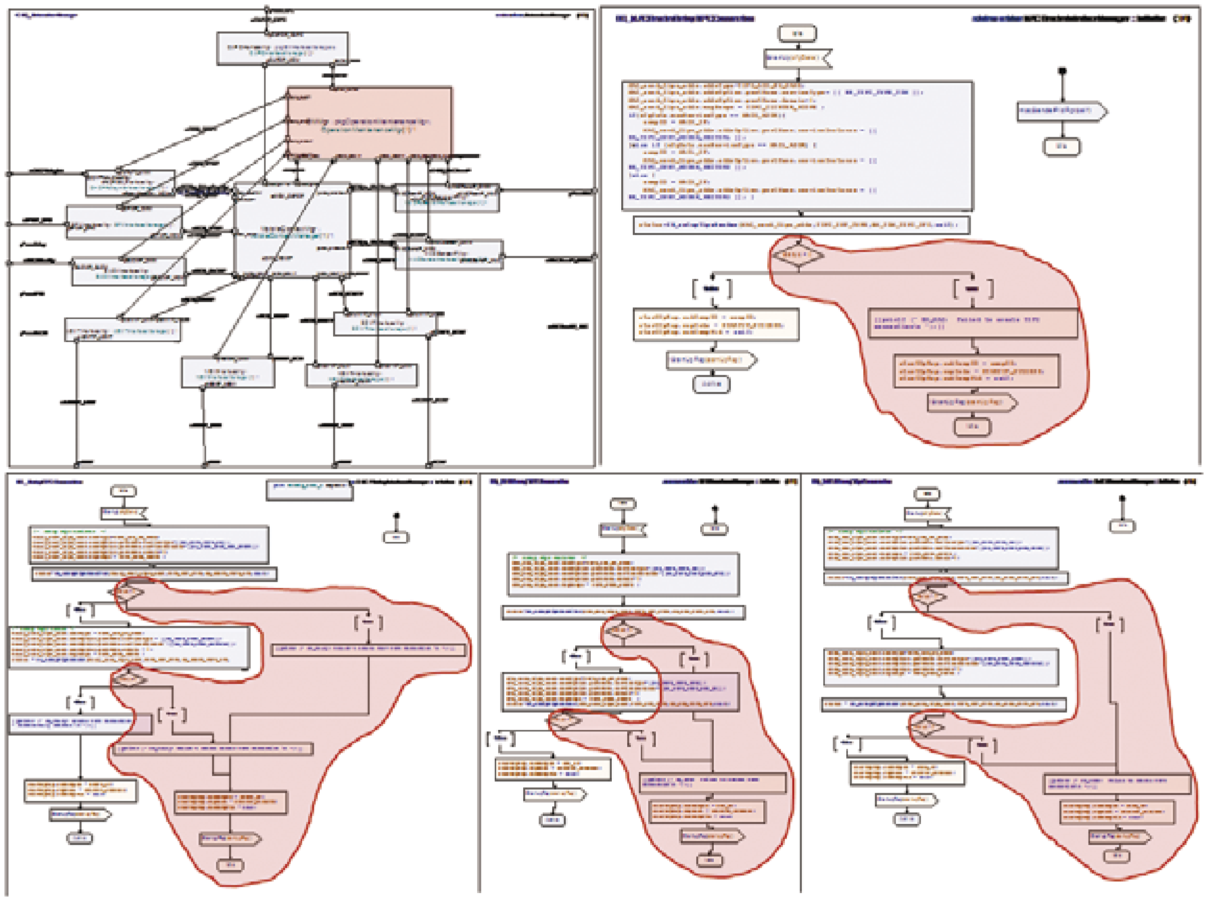

Figure 2. High-level view of two subcomponents of a large communication system. The upper areas are composite-part diagrams that describe the system's physical architecture and the connections between its parts. The lower area represents the state machine diagrams that implement the system parts' behavior. The highlighted sections represent the impact of a fault-tolerance requirement on the implementation of system parts.

be invasive, leading to increased scattering and violation of the open-closed principle.

Substantial reduction in model size

Experience with design models for large-scale communication systems in MOTOwi4 shows that AO techniques can substantially reduce model size and thus make it easier for developers to reason about the models.

Figure 2 gives a high-level view of two subcomponents of a large communication system. Each subcomponent comprises multiple parts. The system is subject to the consistency requirement "When the system starts up, all parts must initialize successfully, otherwise the system must shut down." As the system contains hundreds of parts, this requirement's impact on the implementation is massive. It implies that each part of the system must be able to detect a failure condition upon initialization and notify a coordinator. The implementation of the failure detection and notification concern cannot be modularized in a separate state machine using existing language mechanisms because it interacts with each part's control flow. The Unified Modeling Language's abstraction and composition mechanisms do not allow specification and implementation of the concern in a separate module.

A Motorola study of the MOTOwi4 infrastructure software models showed that more than 40 percent of the state diagram models implement crosscutting behavior related to process coordination and fault tolerance.[8] A version of the same models produced using WEAVR demonstrated that fault tolerance can be refactored into a separate model. The ability to modularize these concerns enables their representation in conceptual models that capture their functionality's essence. Such models allow the reuse of solutions to complex problems such as process coordination and fault tolerance in different contexts. Improved modularity also significantly reduces the size and complexity of the system's models.

POTENTIAL PITFALLS

Along with their advantages, AOSD techniques have some potential pitfalls that practitioners must be aware of.

Ripple effects caused by pointcut fragility

AO techniques typically utilize a syntax-based joinpoint model, whereby programmers specify pointcuts in terms of class names, method names, field names, and the like. If a maintenance change causes any of these elements to be renamed or removed, pointcuts that quantify over them can be potentially invalidated.[10] Researchers are developing semantics-based pointcuts to, among other things, address this fragility,[11] but it remains a problem.

The design stability study of the Health-Watcher system highlighted pointcut fragility problems and various resulting ripple effects (Figure 1b). Programmers had to correct pointcuts to remain compatible with changes made to the base code, and in certain cases this was nontrivial. However, the pointcut fragility revealed a more fundamental issue. When applying maintenance changes, programmers hope to contain any necessary modifications only within modules that are directly related to the concern being modified. For example, in a layered architecture, if the user interface layer is being modified, then the changes should not propagate beyond the view layer. In the AO design, we uncovered significant evidence of ripple effects, whereby changes propagated to seemingly unrelated modules. This was caused by interdependencies, created by pointcuts and inter-type declarations, between the base code and aspects, while the increased tangling in the OO design reduced this scope.

This is not to say that ripple effects did not occur in the OO design, but the scope of the ripple effects differed. The improved separation of concerns within the AO versions caused the changes to propagate to more unrelated modules, making the changes less obvious as unexpected modules were affected. In contrast, the OO version required more extensive changes within each affected component, making these changes more obvious. These differences led to a notion of "deep" and "wide" ripple effects. AO ripple effects tend to go "deeper" in that they propagate to unrelated modules, while OO ripple effects tend to go "wider" in that they more extensively affect the modified modules.

Effect on exception flows

The inversion of control provided by AOSD techniques helps separate crosscutting concerns. At the same time, however, the "new" aspectual modules must be tested. This is particularly problematic when trying to test aspects' effects on a program's exception flows. As aspects extend or replace existing functionality at specific joinpoints in the code execution, their behavior may raise new exceptions that can flow through the program execution in unexpected ways.

AO developers attempt to ensure that aspects do not create exceptions that impact applications, but unexpected behavior in aspect code such as referencing unanticipated null values or calling a library method that throws undocumented runtime exceptions often occurs.[12] Such exceptions may remain uncaught, causing the software to crash in an unpredictable way, or may be mistakenly handled by an existing handler in the base code, introducing an *unintended handler action*.[13]

In all of the AspectJ versions of one Lancaster testbed study (Figure 1c), we observed a significant increase in the number of uncaught exceptions and a decrease in the number of exceptions caught by specialized handlers. The number of exception *subsumptions*—exception types associated with the handler that are a supertype of the exception type being caught[13]—also increased in most AspectJ versions.

During code inspections, we discovered a set of recurring programming anomalies in the exception-handling code of AspectJ programs,[12] which we categorized into three bug patterns:

- bugs related to aspects that signal exceptions (for example, *solo aspect signaler*—no handler is defined for the exceptions signaled by the aspects);
- bugs related to exception-handling aspects (for example, *late binding aspect handler*—the aspect intercepts a point in the base code in which the exception was already caught by another handler on the base code); and
- misuses of the `declare soft` construct in AspectJ, a construct that wraps any checked exception into an unchecked exception.

These bug patterns can impair a system's robustness. In the short term, AO programmers must learn to recognize and avoid these patterns. However, the problem can only be tackled effectively by improving the design of exception-handling mechanisms in AO programming languages and building verification tools and techniques tailored to improve the reliability of exception-handling code in AO systems.

The software systems using AOSD that we have studied are medium- to large-scale and span a wide range of domains including enterprise systems, e-health, e-transport, telecommunications, Web-based information systems, multimedia applications, and workflow systems. Our analysis highlights typical usage patterns of AO techniques—for instance, they are mainly used for modularizing well-known crosscutting problems and incrementally introduced, addressing developmental concerns and other noncore product features first.

The benefits of modularity, such as reducing software model size and improving design stability over a history of changes, are substantial. As with any new technology, however, there are also potential pitfalls, especially in regard to testing execution paths, that developers must be aware of to ensure robustness of AO applications. Pointcuts also appear to be a double-edged sword: While they enable certain changes to be absorbed and thereby increase a design's stability, they are also the source of ripple effects that reduce stability. Developers must account for this tradeoff when incorporating AO technologies in a project.

References

1. R.E. Filman et al., eds., *Aspect-Oriented Software Development,* Addison-Wesley, 2004.
2. E. Baniassad et al., "Discovering Early Aspects," *IEEE Software,* vol. 32, no. 1, 2006, pp. 61-69.
3. A. Colyer and A. Clement, "Large-Scale AOSD for Middleware," *Proc. 3rd Int'l Conf. Aspect-Oriented Software Development* (AOSD 04), ACM Press, 2004, pp. 56-65.
4. D. Wiese and R. Meunier, "Large Scale Application of AOP in the Healthcare Domain: A Case Study," keynote address, 7th Int'l Conf. Aspect-Oriented Software Development (AOSD 08), 2008.
5. M. Braem et al., "Isolating Process-Level Concerns Using Padus," *Proc. 4th Int'l Conf. Business Process Management* (BPM 06), LNCS 4102, Springer, 2006, pp. 113-128.
6. L.D. Benavides Navarro et al., "Explicit Distributed AOP Using AWED," *Proc. 5th Int'l Conf. Aspect-Oriented Software Development* (AOSD 06), ACM Press, 2006, pp. 51-62.
7. A. Sampaio et al., "EA-Miner: Towards Automation in Aspect-Oriented Requirements Engineering," *Trans. Aspect-Oriented Software Development III*, LNCS 4620, Springer, 2007, pp. 4-39.
8. T. Cottenier, A. van den Berg, and T. Elrad, "The Motorola WEAVR: Model Weaving in a Large Industrial Context," Industry Track, 6th Int'l Conf. Aspect-Oriented Software Development (AOSD 07), 2007; http://aosd.net/2007/program/industry/I3-MotorolaWEAVR.pdf.
9. P. Greenwood et al., "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study," *Proc. 21st European Conf. Object-Oriented Programming* (ECOOP 07), LNCS 4609, Springer, 2007, pp. 176-200.
10. A. Kellens et al., "Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts," *Proc. 20th European Conf. Object-Oriented Programming* (ECOOP 06), LNCS 4067, Springer, 2006, pp. 501-525.
11. R. Chitchyan et al., "Semantics-Based Composition for Aspect-Oriented Requirements Engineering," *Proc. 6th Int'l Conf. Aspect-Oriented Software Development* (AOSD 07), ACM Press, 2007, pp. 36-48.
12. R. Coelho et al., "Assessing the Impact of Aspects on Exception Flows: An Exploratory Study," *Proc. 22nd European Conf. Object-Oriented Programming* (ECOOP 08), LNCS 5142, Springer, 2008, pp. 207-234.
13. M. Robillard and G. Murphy, "Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems," *ACM Trans. Software Eng. and Methodology,* vol. 12, no. 2, 2003, pp. 191-221.

Awais Rashid *is a professor of software engineering in the Computing Department at Lancaster University, UK, and the Pays de la Loire Regional Chair at École des Mines de Nantes (EMNantes), France. He received a PhD in computer science from Lancaster University. Contact him at awais@comp.lancs.ac.uk.*

Thomas Cottenier *is a senior research associate in the Computing Department at Lancaster University. He received a PhD in computer science from Illinois Institute of Technology. Contact him at thomas.cottenier@hengsoft.net.*

Phil Greenwood *is a senior research associate in the Computing Department at Lancaster University. He received a PhD in computer science from Lancaster University. Contact him at p.greenwood @lancaster.ac.uk.*

Ruzanna Chitchyan *is a senior research associate in the Computing Department at Lancaster University and coleads the work on aspect-oriented analysis and design in AOSD-Europe. She received a PhD in computer science from Lancaster University. Contact her at r.chitchyan @ lancaster.ac.uk.*

Regine Meunier *is a research scientist and software engineer at Siemens AG, Germany, where she leads aspect-oriented software development activities. Contact her at regine.meunier@siemens.com.*

Roberta Coelho *is a professor in the Department of Informatics and Applied Mathematics at Federal University of Rio Grande do Norte, Brazil. She received a PhD in computer science from PUC-Rio in cooperation with Lancaster University. Contact her at roberta@dimap.ufrn.br.*

Mario Südholt *is an associate professor in the Department of Computer Science at EMNantes and heads the ASCOLA group, a joint research team with INRIA that investigates aspect and composition languages. He received a PhD in computer science from Technische Universität Berlin, Germany. Contact him at sudholt@emn.fr.*

Wouter Joosen *is a professor in the Department of Computer Science at Katholieke Universiteit Leuven, Belgium. Contact him at wouter.joosen@cs.kuleuven.be.*