# Whitepaper: The Value of Improving the Separation of Concerns

## EXECUTIVE SUMMARY

Microsoft's enterprise customers are demanding better ways to modularize their software systems. They look to the Java community, where these needs are being met with language enhancements, improved developer tools and middleware, and better runtime support. We present a business case for why Microsoft should give priority to supporting better modularization techniques, also known as *advanced separation of concerns (ASOC)*, for the .NET platform, and we provide a roadmap for how to do so.

### Goals

The whitepaper is targeted towards developers and managers at Microsoft. The goals of the document are

- Describe "crosscutting" as a natural, unavoidable, quantifiable, treatable, and, most importantly, expensive problem in software development.
- Describe ASOC in an unambiguous way and clearly explain its advantages and disadvantages without using hype-infused terms and "research-speak".
- Describe industry need and potential for ASOC.
- Provide a clear roadmap for supporting ASOC.
- Indicate risks and open issues, and outline risk mitigation strategies.
- Encourage Microsoft to investigate ASOC and help us better understand their issues with it.

# 1. Introduction

The complexity of software continues to increase. Our users, and the market, have an insatiable demand for new features. The Computer Science field has evolved to manage this increase in complexity. Programming languages are safer (e.g., C#/.NET, Java), yet more dynamic (e.g., Ruby, Python). Development tools that support Intellisense, refactoring, static analysis (e.g., FxCop, PREFix, Lint), and generative programming (e.g., unit test generation, proxy generation), enhance programmer productivity. Development processes have become more distributed, collaborative (e.g., open/shared source), and agile (e.g., XP, Scrum).

However, none of these innovations have been able to fully address a fundamental problem in our field: *crosscutting*. This problem limits our ability to manage software complexity, leading to long release cycles and increased development and maintenance costs.

# 2. Crosscutting is a Fundamental Problem

A *concern* is a feature, requirement, policy, issue, design rule of a software system. Consider the canonical "shapes" concern: we want to support various types of graphical shapes (squares, circles, etc.) that can all be moved and drawn on the screen. This concern maps nicely to a class hierarchy in the object-oriented programming paradigm where each distinct shape derives from a generic Shape base class.

### Crosscutting Concerns

Unfortunately, some concerns defy modularization using traditional object-oriented programming languages like C++, Java, and C#. These are called **crosscutting concerns** -- the concern's design or implementation is *scattered* throughout the system and *tangled* with the design and implementation of other concerns.

Sometimes a crosscutting concern is easy to spot. In the source code, a crosscutting concern manifests as code, often called "boilerplate code", which is repeated in multiple functions and files in order to satisfy a single concern (see Figure 1). While the crosscutting code may not be exactly the same each time, the purpose of the code is the same.

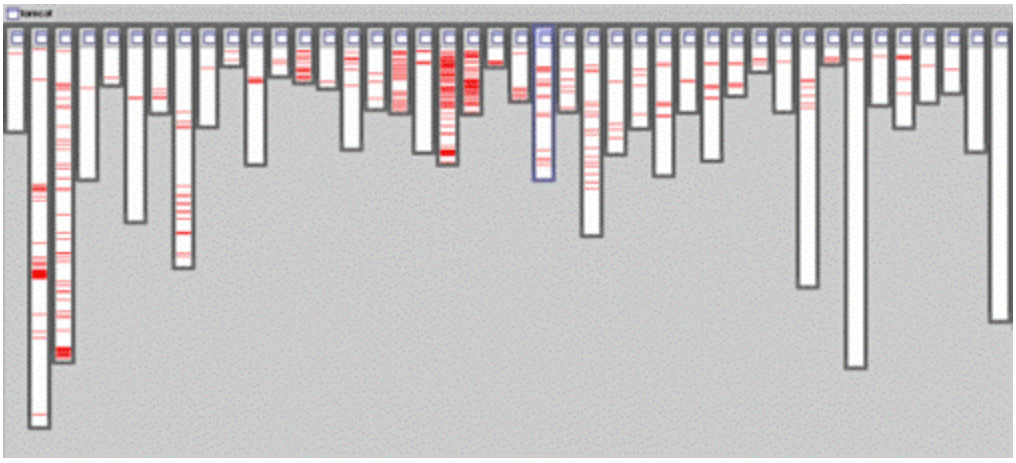

*Figure 1. Example taken from the Tomcat web application server. Each column represents a source file. The length of the column represents lines of code. The red lines indicate lines of code related to the logging concern. The logging concern is crosscutting because logging-related code is scattered across many source files.*

Crosscutting code typically invokes functions or members of another class. This shows up as a *uses* dependency between the two classes. Because crosscutting code is, by its very nature, repetitive and numerous, this results in one or more classes with a high fan-in of uses dependencies (see Figure 2).

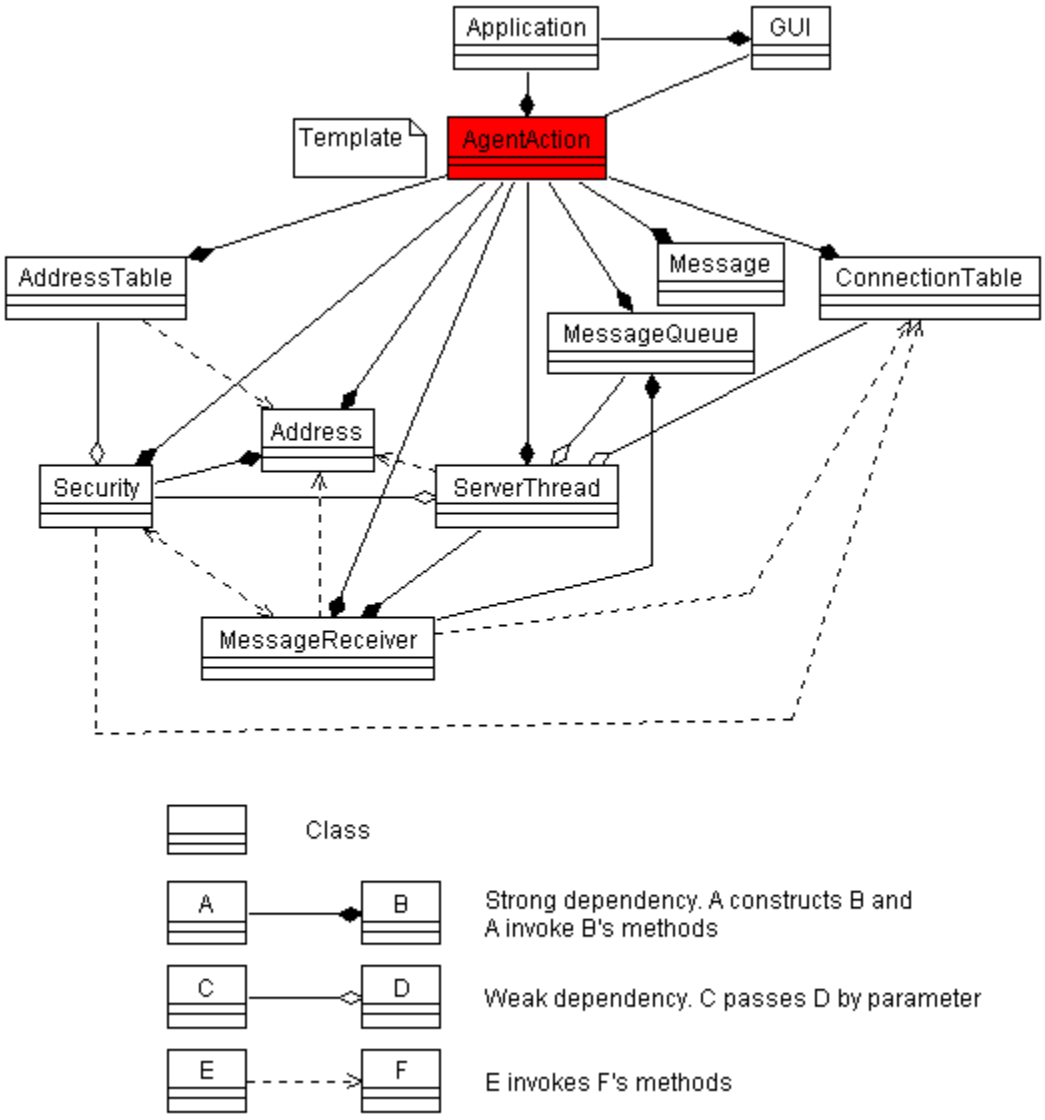


*Figure 2. Example taken from the JATLite multi-agent system. AgentAction is a crosscutting concern because many classes use it.*

*Image © Heecheol Jeon*

**An Example: Error Handling**

Another example of a crosscutting concern is **error handling**. A typical error handling policy requires that we

> "check return values of all functions and handle errors appropriately"

This concern is crosscutting because it requires error checking and handling code in every function in the program. The code must be manually written, a tedious and error-prone process, and one which inevitably leads to inconsistencies. The final result is not pretty: the error handling code is tangled with the original code making both hard to understand. Even when using state-of-the-art object-oriented design principles and patterns, this code cannot be fully extracted and encapsulated.

Not only does crosscutting negatively impact software design, implementation, and maintenance, it also hinders software evolution. Consider a change to the error handling policy:

> "all functions must be *hardened* to ensure they fail gracefully and deterministically in low memory situations"

Because the error handling concern is not modularized (a consequence of crosscutting), this policy change may require manual changes to every function in the program.

**Crosscutting is a Natural Phenomenon**

Crosscutting is not limited to the field of Computer Science. It is a phenomenon observable in the structure of organizations, city plans, and architecture blueprints, too name a few. For example, an organization is by-and-large structured as a hierarchy. However, its accounting department is crosscutting because it requires accounting information from each department.

City building design and construction is on the whole a distributed and independent endeavor. However, building codes, zoning rules, electricity supply, and trash and sewage service represent concerns that crosscut the construction of all buildings.

As a third example, the floor plan of a building can be designed independently to a large extent. During construction, however, one must punch holes for electrical outlets, provide adequate spacing between walls for insulation and plumbing, construct conduits for electrical wiring, and ducts for air conditioning. These all represent concerns that crosscut the building. Furthermore, these concerns may interact or constrain each other. For example, phone, networking, and electrical wires may share the same conduits.

**Crosscutting is Unavoidable**

We believe that some crosscutting is unavoidable. Given any decomposition of a software system, whether it be object-oriented, aspect-oriented, procedural,

functional, data flow-based, event-based, state machine-based, etc., there will always be concerns that crosscut the system. This phenomenon is called the "tyranny of the dominant decomposition".

However, it is not as bad as it sounds. First, some programming methodologies, for example, aspect-oriented programming (AOP), provide direct support for modularizing many crosscutting concerns. This has the effect of reducing crosscutting.

Second, we do not need to eliminate crosscutting entirely. Rather, we should only modularize a crosscutting concern when there is a perceived or expected benefit. Some simple heuristics:

- there is little benefit in modularizing a concern that only crosscuts a small number of system components; and
- there is little or no benefit in modularizing a crosscutting concern that is never subsequently modified. (This is not always true. Some companies derive great value from the consistency gained from modularizing a crosscutting concern even if the concern is not expected to change.)

It is not surprising that these heuristics mirror those for modularization in general. As the software evolves, we may need to revisit these modularization decisions and refactor crosscutting concerns that are new, more abundant, or less stable than initially assumed.

**Crosscutting is Measurable**

Several tools have been developed for mining and refactoring crosscutting concerns from existing (legacy) code bases; this is an area of active research. They use various techniques and metrics to identify crosscutting code and refactor it into separate modules. Other tools measure the general degree of crosscutting present in a software system. These measurements are useful for comparing system designs that seek to minimize crosscutting.

Recently, *net option value*, a term that originates from finance, is being used to measure the cost of crosscutting. The idea is that when designing a software system, each possible design of the system represents an option we can take. As we all know, some designs are better than others. This is captured by the idea that each design option has a value. We can determine the actual value of a design we chose at any given point in time by adding up the profits gained from the software system and subtracting the costs of building and maintaining it. However, by that time the information is not very useful; we would like to choose a high value design option at the start of the project.

How can we pick a high value design from a set of design options? We need to know what changes will be made to the system and what the cost of those changes are. Both of these must be approximated: there will always be unanticipated changes and change costs vary widely depending. ASOC

techniques provide a way to make unanticipated changes in a *disciplined* way, without requiring massive refactoring of the existing design and code base.

**A Solution: Advanced Separation of Concerns**

The goal of ASOC is to pick up where conventional programming methodologies leave off by providing direct support for modularizing crosscutting concerns. Crosscutting concerns can either be identified during the design phase or refactored from an existing code base. They are then placed into separate modules thus gaining all the benefits of modularity including independent development and compilation.

Readability is also improved. The other parts of the system are no longer cluttered with the code related to the concern. All the code related to the concern is in one place, making it easier to reason about. Recent advances in IDE support further improve readability and predictability by allowing the developer to see the points in the code where a concern applies.

Consistency is a priority for many companies. They want consistent enforcement of security policies, consistent error checking, and consistent logging, for example. Consistency is difficult to enforce for crosscutting concerns, since the code related to these must be manually written at many different points in the system (scattering). ASOC improves consistency and automates tedious and error-prone programming by allowing a concern to be specified in a single module and automatically applied to multiple points in the system.

There are many different techniques for enabling advanced separation of concerns including:

- Aspect-Oriented Programming
- Open Classes
- Runtime Reflection
- Compile-Time Reflection
- Generative Programming
- Metaobject Protocols
- Metaprogramming
- Hyperdimensional Separation of Concerns
- Subject-Oriented Programming

# 3. Real-World Case Studies

The following case studies give concrete examples of companies choosing ASOC techniques to meet their business needs. Aspect-oriented programming is the most popular ASOC technique so it is not surprising that it is used in all the case studies.

## 3.1 Case Study: Deutsche Post World Net

Deutsche Post World Net integrates the Deutsche Post, DHL and Postbank companies ("The Group") to offer tailored, customer-focused solutions for the management and transport of goods, information, and payments through a global network combined with local expertise. They are also the leading provider of Dialog Marketing services, with a unique portfolio of efficient outsourcing and system solutions for the mail business. The Group generated revenue of Euro 56 billion in 2005. With currently some 500,000 employees in more than 220 countries and territories, Deutsche Post World Net is one of the biggest employers worldwide.

**Business Need: Dynamic Reconfiguration**

The Deutsche Post has evaluated aspect-oriented programming (AOP) techniques and the Rapier-LOOM.NET aspect weaver to identify reconfiguration concerns for multithreaded component-based applications. Reconfiguration allows components to be updated without knowledge of the components' functionality and their interactions with other components.

The adaptation to changing environmental conditions, new product requirements, or software errors demands the ability to reconfigure software. Today's software systems often use threads to handle many tasks simultaneously. This makes the reconfiguration of such software a challenge. Inter-component interactions must be tracked in order to identify valid reconfiguration points. Furthermore, long running applications can not be restarted for reconfiguration so they must support online (dynamic) reconfiguration.

As a technology study our solution has been applied in the context of EPOS, a large retail application at Deutsche Post World Net. EPOS is currently installed on tens of thousands of computers distributed over several thousand post offices throughout Germany. One of the requirements for the application is that business logic can be exchanged dynamically. I.e., **if the marketing department plans special offers for certain products, an according business case component must be deployed on demand**. Another desire is that **software errors must be fixed immediately**, for example, if operations control detects a critical security issue within the credit card component. Both of these business needs require support for dynamic reconfiguration.

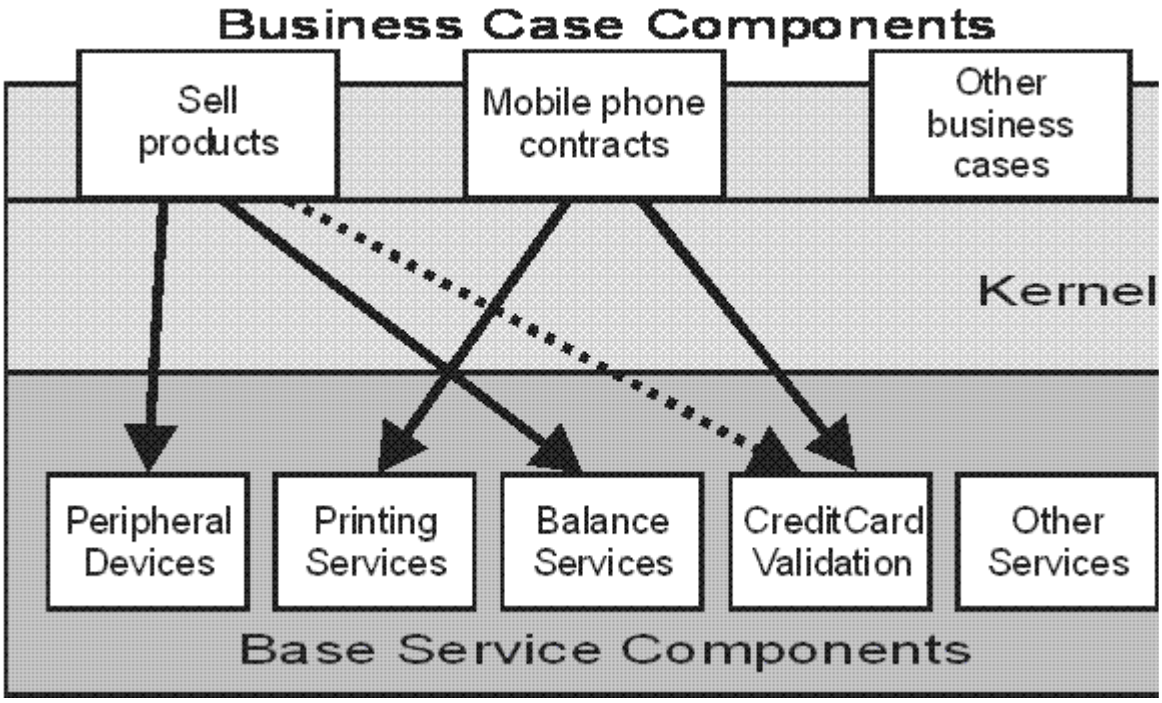


*Figure 3. The EPOS Retail Application.*

Figure 3 shows a skeleton of the EPOS application. Every business case (e.g., selling a certain product or making a cell phone contract) is implemented inside a business case component. These components are shown in the upper part of the picture. At the moment there are approximately 350 business case components available. Depending on their requirements, business case components can use several base services. This can be seen in the bottom of Figure 3. A base service typically implements common tasks, i.e., it provides functionality to print invoices, to gain access to a product scanner, or to authorize credit card payments. Base services can use other base services. A typical configuration includes about 50 base services.

**An AOP Solution**

*Shorter product cycles, new offerings, and the requirement to react almost immediately to cyber-attacks make an architecture that supports dynamic reconfiguration highly desirable.*

The technique of dynamic weaving offers a novel approach for the development of reconfigurable applications. In contrast to existing implementations using byte-code rewriting, our AOP approach does not require additional compilation steps. The seamless integration into the software development process is implemented by the transparent activation of the reconfiguration aspect during component instantiation through our framework using the factory pattern. (In the described retail application, the kernel instantiates new components and weaves them dynamically.)

**Results**

The management of inter-component interactions needed for the enforcement of valid reconfiguration points is handled by our aspect within the callee. This enables the tracking of component usage without requiring explicit knowledge about client components. With our approach, **no reconfiguration specific code at client side is required**. In contrast, most existing solutions for application reconfiguration depend heavily on extending the client logic.

## 3.2 Case Study: Philips

Royal Philips Electronics of the Netherlands is one of the world's biggest electronics companies, as well as the largest in Europe, with 161,500 employees in over 60 countries and sales in 2005 of Euro 30.395 billion. Philips is active in over 60 businesses and holds more than 115,000 registered patents. Philips is a global leader in electric shavers, lighting, medical diagnostic imaging systems, and video and audio products, such as televisions and DVD recorders. Dow Jones recently ranked Philips the global leader in sustainability.

**Business Need: Improve Modularity**

Philips Research is looking for improvements in software development in general, and resource limited applications, such as televisions, in particular. Software development benefits from separation of concerns, i.e., the ability to deal with the difficulties, the obligations, the desires, and the constraints one by one (Dijkstra, *A Discipline of Programming*, 1976), is very desirable, especially when the modularization of the reasoning can be reflected in the modularization of the software.

The software in Philips televisions is currently modularized using the proprietary component model *Koala*. Koala enables us to handle the diversity of our product family of televisions. Recently, we observed some limitations of Koala: the modularization of the reasoning cannot always be reflected in the modularization of software into components:

- Many (nonfunctional) concerns are not localised in one component but are **scattered** throughout our software, and
- Multiple concerns are **tangled** in one component.

**An AOP Solution**

To better support modular reasoning, Philips Research is looking for more effective modularization techniques, such as aspect orientation.

An aspect-oriented framework on top of Koala is currently being developed at Philips Research. We will now briefly describe our experiences with this framework.

**Results**

How to handle *access before initialization* is a concern that affects all components. Although one can easily describe how to handle access before initialization in general, it is currently handled per component. Even worse, this handling differs between components in the same software stack.

With aspect orientation, we were able to write three different strategies to handle access before initialization. The first strategy *asserts that a component is not accessed before initialization*; the second strategy *ignores accesses when the component is not yet initialized*; and the third strategy *calls the initialization code when the component is accessed before being initialized*. **With these strategies, we could ensure that all components handle access before initialization identically, reduce the lines of code by 3%, and postpone the decision for a strategy from implementation time to integration time.**

*Resource usage* is an important concern for resource-limited systems. The functionality to check that a component does not use more resources than specified can be localised in one component. Yet, for each component under test one still has to do a lot of plumbing:

- Instantiate a test component, and
- Change the connections to the component under test to pass through this test component.

**With aspect orientation, we were able to localise not only the functionality but also the plumbing in one aspect. This made the test process both easier and less error-prone.**

Many pieces of software may not be accessed multithreaded, but accidentally are accessed on multiple threads. Integration and testing benefit from *automatic detection of illegal multithreaded accesses*. **We have written an aspect that lists multithreaded accesses throughout the complete software stack. This list helps our architects to pinpoint illegal multithreaded accesses.**

During integration and testing, *understanding the dynamic behaviour of a system is crucial*. Tracing provides insight into this behaviour. **We have written and applied an aspect to trace the interface function calls made in the software of an already finished television set**. While manually adding trace statements requires programming effort linear with the number of interface function calls, **this aspect required only a small programming effort that is independent of the number of interface function calls**. Currently, we are reusing this aspect to see how the Nexperia platform is accessed by the applications running on top of it.

We focused on applying aspect orientation in the validation and verification phase to reduce the risks associated with introducing this new technology. Yet, **our experiences convinced us of the added value of aspect orientation for our products**. Consequently, **Philips considers native support for aspect orientation an important property of a software development environment.**

## 3.3 Previously Published Case Studies

In addition to the case studies presented in the previous sections, quite a few industrial application reports of aspect-oriented programming approaches have been published at ASOC-oriented conferences and workshops by big enterprises. As a selection, short summaries of three published articles will be given in the following (copyrights and trademarks apply as given by the papers):

- *Large-scale AOSD for Middleware* by Adrian Colyer and Andrew Clement of IBM (published at AOSD'04),
- *Applying AspectJ to J2EE Application Development* by Nicholas Lesiecki of VMS (AOSD'05), and
- *Creating Pluggable and Reusable Nonfunctional Aspects in AspectC++* by Michael Mortensen of HP and Sudipto Ghosh of Colorado State University (ACP4IS'06).

**IBM: Large-scale AOSD for Middleware**

International Business Machines Corporation (IBM) is a large technology company founded in 1888 and nowadays, with over 300,000 employees and revenues of over $90 billion in 2005, one of the biggest IT companies in the world. Among other products, IBM offers a series of middleware products, which are aimed to hide the complexity of building distributed applications with certain quality of service requirements.

Both in building and applying middleware, there is a significant amount of inherent complexity due to a typically high number of features (especially in mature products), a large group of technologies and products which need to be integrated (such as databases, message brokers, and web servers), heterogeneous environments, and possibly resource constraints of the target systems. To ensure consistency for different members of product lines, flexibility in creating members of such a line, and a simple programming model, IBM investigated the application of aspect-oriented software development.

Using the AspectJ aspect-oriented programming toolkit (including the AspectJ compiler and AJDT development tools for Eclipse), the research team **managed to encapsulate both homogeneous crosscutting concerns**, such as tracing, logging, first-failure data capture, monitoring, and statistics, **and the heterogeneous concern** of separating support for Enterprise Java Beans (EJBs) from the rest of the middleware, allowing support for EJBs to be configured at product build time (effectively creating two different versions of the product).

The result of their investigation was that **aspect-oriented software development is applicable to large-size commercial projects** (also from a performance point of view) and that it can be **successfully integrated into existing development processes**. It also showed, however, that AOP tool support (AJDT) still had to evolve before being accepted by mainstream developers.

**VMS: Applying AspectJ to J2EE Application Development**

Video Monitoring Services of America (VMS) is a corporation providing news media monitoring, advertising monitoring, and Integrated Media Intelligence solutions. In 2004, VMS started to integrate aspect-oriented programming into their *Adbase* product, a J2EE-based application for interfacing and searching advertising data. The Adbase team had identified a number of crosscutting concerns and was aiming to increase their source code's modularity, flexibility, and clarity.

Using AspectJ as a general-purpose aspect-oriented extension of the Java programming language, which could integrate existing object-oriented Java code without changes, selected crosscutting concerns (error logging and the application-specific concerns of performance statistics gathering and administration workflow management) were refactored and implemented in an aspect-oriented fashion. Since the effort paid off, a large number of limited-scope application-specific aspects followed, including a most advanced set of concerns comprising a persistence solution with *bidirectional relationship propagation*, which was missing in the Hibernate tool they were using at that time.

For VMS, the **adoption of AOP was very successful**: In the time of six months, the Adbase **team managed to adopt the concepts and tools** required for aspect-oriented programming and refactor their system in an aspect-oriented fashion, including the addition of new, complex features (like the persistence solution). Unit testing, pair programming, and refactoring proved to be valuable techniques for handling the challenges in this process. AspectJ and the AJDT development tools provided a relatively mature tool suite for the adoption of AOP, and VMS considers their adoption of AOP to be successful.

**HP: Pluggable and Reusable Non-functional Aspects in AspectC++**

Founded in 1939, the Hewlett-Packard Company today is one of the largest IT companies in the world with about 140,000 employees worldwide. Their products among many others also comprise very-large-scale integrated (VLSI) computer chips, which are designed using VLSI CAD frameworks. Such frameworks usually provide extension points for reuse of framework components in an object-oriented fashion. However, such extension points must be foreseen by the framework developers, which is not always possible.

HP therefore investigated the possibilities of using aspects to extend and enhance existing object-oriented VLSI frameworks. Using the AspectC++ aspect-oriented programming tool, they built a library of aspects representing crosscutting concerns of applications built on the frameworks, refactoring and *aspectizing* the applications rather than the frameworks themselves. That way, they **were able to circumvent any problems related to licensing or the large size of such frameworks** while still profiting by the SOC benefits provided by aspect-oriented software development.

After investigating non-functional aspects for caching and run-time configuration use cases, the research team **developed a set of recommendations for developing highly reusable aspects** which can be bundled in a library and applied in many different situations. According to their concluding statement, adherence to these guidelines makes aspects an **effective modularization mechanism for non-functional crosscutting concerns** in their field of operation.

## 3.4 Commercial Products

Several commercial products provide an aspect-oriented programming interface for configuring their product, including:

- JBoss AOP
- BEA JRockit JVM
- IBM WebSphere
- Jakarta HiveMind
- Senselogic SiteVision
- Near Infinity IntelliPrints

# 4. Recommendations for Microsoft

Support for ASOC is not an all or nothing affair. In fact, the .NET Platform already provides some basic ASOC functionality in the form of

- Reflection,
- Code generation/instrumentation (Reflection.Emit, Phoenix),
- Dynamic proxies (ContextBoundObject), and
- Interception and dynamic updating (Profiler and Debugger APIs)

However, these .NET technologies were not created with ASOC in mind which limits the power and elegance of the AOSC implementations that are built on top of them.

**Our overall recommendation is for Microsoft to include ASOC requirements in the design of the next generation of the .NET Platform.** We provide a detailed description of these requirements below.

**We also recommend that Microsoft (re)evaluate ASOC both internally and externally.** Several groups at Microsoft, for example, Phoenix, Singularity, Common Language Runtime (CLR), Microsoft Business Framework, and Prescriptive Architecture Group (PAG), have already adopted, or expressed an interest in adopting, ASOC techniques. They can serve as pilots to help solidify the ASOC feature set and to flush out issues. Some of Microsoft's industry

partners are also interested (in fact, many are already using ASOC techniques but do not call it that) and they can serve as early adopters.

# 5. ASOC Requirements for the .NET Platform

The following requirements were gathered from the AOP-.NET community.

## Improved Proxy Support

Proxies that intercept method calls can be used to implement a wide variety of interception-based ASOC techniques. Interception-based techniques are very attractive because they are simple, powerful, and require little or no instrumentation of the program.

- **Noninvasive interception** - Currently, injecting a dynamic proxy requires that a class derive from ContextBoundObject or MarshalByRefObject. This requires invasive changes to the client code base, which is itself a form of crosscutting and does not facilitate unanticipated proxying, both of which prevent ASOC techniques from being applied to third-party libraries or legacy code. This makes it necessary for adopters of ASOC to implement their own proxy facilities relying on dynamic code generation (e.g. generating subclasses of the proxies classes).

- **Ability to intercept instance construction** - Currently, when relying on code generation rather than ContextBoundObject-based proxies, clients must instantiate a class using a class factory in order for the class to be ASOC-enabled. This requires invasive changes and has the same associated restrictions.

- **Intercept self-calls** - Currently, proxying based on ContextBoundObject only works on cross-context calls. Therefore, an object that calls a method on itself, e.g., this.ToString(), bypasses the proxy.

- **CLR proxy performance** - Since proxies based on ContextBoundObject are originally intended for .NET remoting, not much effort has been put into runtime performance of proxied classes. Especially the performance of method calls is currently very bad.

- **Allow for interface introduction** - Aside from interception features, ASOC tools would benefit from proxies being able to add additional interfaces to the proxied objects.

- **Intercept non-virtual calls, field accesses, etc.** - Currently, proxies based on dynamically generating subclasses only support virtual calls, whereas proxies based on ContextBoundObject only support cross-context calls. This severely limits the ASOC functionality that can be enabled. A more powerful infrastructure must provide more interception functionality.

## Static Instrumentation API

Instrumentation is by far the most powerful and highly performance technique used to enable ASOC.

- **Improve CodeDOM support** - Currently the CodeDOM API is incomplete. Ideally, CodeDOM would be able to express all C# and C++/CLI language constructs and have parsers for C# and C++/CLI. Many ASOC tools also need transformations to be "roundtrip-able" which requires CodeDOM to be able to express the complete program syntax tree including comments and whitespace.

- **ASOC-aware IL instrumentation tools** - Microsoft Phoenix and ildasm can be used to manipulate MSIL. However, building an ASOC tool that leverages them is very difficult. The ASOC community has responded by creating ad hoc tools but these tools are not production quality, for example, they cannot keep the PDB file in-sync. Ideally, Microsoft would support an AOSC-aware instrumentation tool, for example, a static aspect-oriented programming weaver. This would make it easier to instrument third-party libraries and applications written in any .NET language.

## Dynamic Instrumentation API

In .NET 1.1, the Profiler API allowed efficient method interception and the ability to modify IL code at runtime (e.g., to inject method calls). However, profiler-based tools were not attractive for production scenarios because they required manual registration of the profiler and could not be used in conjunction with a real profiler.

The IL modification feature was disabled in .NET 2.0. The Debugger Edit-and-Continue API in .NET 2.0 does allow dynamic modification but it is extremely inefficient and unusable for production scenarios. The Debugger API also suffers from a similar drawback as the Profiler API in that it precludes the use of a commodity debugger.

- **Efficient IL updating**
    1. Swapping the IL code in a method body should be efficient. The cost of enabling this functionality should be < 5%. The update time for a single method should be less than 1 sec. (These performance goals were obtainable with the Profiler API v1.1).
    2. Ideally, the updating API would be a managed API as opposed to the unmanaged Profiler API.
    3. It must be possible to update optimized code and allow optimized code-gen. Updateable methods must also be inlineable.
    4. Currently, methods that have already been updated are not freed, resulting in a memory leak.

- **Efficient metadata updating** - Currently, this requires the application to be launched in edit-and-continue mode which slows it down by more than 10%. Ideally, enabling both IL and metadata updating would have a total overhead of < 5%. It is also not possible to extend NGEN'd assemblies.

- **More powerful metadata updating** - The ability to update metadata is currently very constrained. Only private members (fields, methods, properties) can be added. This makes it impossible to extend the C# Partial Classes feature to load-time or runtime scenarios, which is exactly the goal of the Open Classes ASOC technique.

- **Flexible attach/detach** - Currently, the application must be run under a specialized debugger to be dynamically updateable. However, this prevents commodity debuggers from being used to debug problems. Ideally, the .NET platform would provide a flexible way to attach to a running process and update any part of it dynamically, without involving a debugger.

- **Delta file generation** - Currently, Microsoft does not provide a mechanism to diff two assemblies in order to produce the delta files needed by the Edit-and-Continue API. Ideally, Microsoft would provide an API that would diff two assemblies and produce the IL and metadata delta file, which could then be used directly by the Edit-and-Continue API.

- **Improve Reflection API** - Currently, the Reflection API provides introspection and invocation functionality. Ideally, it would provide the ability to add/remove/modify program elements and IL code.

- **Managed Profiler API** - Currently, the Profiler API allows method calls, etc. to be intercepted but, unfortunately, it is an unmanaged API. Ideally, Microsoft would provide a managed Profiler API.

# 6. Risk Assessment & Mitigation

Depending on the way it is implemented, the adoption of ASOC might pose risks to the entire software development lifecycle. Support for ASOC varies from simple preprocessing tools to ASOC containers to specialized virtual machines. ASOC usage can go from merely doing performance analysis in a QA environment to building a whole enterprise application around an ASOC-oriented architecture. Taking into account this broad range of adoption possibilities, we suggest some ways that Microsoft can mitigate risk.

## 6.1 Support & Servicing

### Testing

ASOC can affect testing in positive and negative ways. On the plus side, ASOC techniques make it easy to inject faults and "test probes" into a program for white box testing. Contracts can be defined externally and then checked at test

time by using ASOC techniques to insert contract enforcement code. ASOC techniques make it trivial to automatically instrument applications with monitoring and profiling code. Testers can use this data for regression testing, to obtain field diagnostics, and ensure performance requirements are met. During debugging, testers and developers can leverage ASOC techniques to dump data structures.

On the other hand, testing is more difficult because the behavior of the program may not correspond with its source code. This problem also exists with more traditional techniques for separating concerns like Reflection and virtual functions. For example, it is not obvious from the source code which function will be called when you call a virtual function. A tester or developer that understands object-oriented programming can easily cope with this. However, without a good understanding of ASOC it may be more difficult for them to understand the behavior of an ASOC-enabled program.

*Mitigation strategy*

The same mitigation strategy that made understanding virtual functions easier can make understanding ASOC-enabled programs easier: training and tool support. Developers and testers must fully understand the particular ASOC technique used. IDE support for ASOC techniques is critical and is already available for some techniques.

## Customer Support

Microsoft's customer support infrastructure is built around the assumption that executable files are fixed, that is, an executable's size and modification date are unchanged after its been released. *Invasive* ASOC techniques modify the executable and therefore break this assumption. This makes it difficult for customer support to know exactly what bits the customer has on their machine. It also causes problems for Microsoft's automated utilities like Windows Update and for virus checking software.

Invasive ASOC techniques invalidate the assumption that executable boundaries signify code ownership. Code inside the executable may have been injected from a third-party library. When a fault occurs due to the injected code, tools such as Windows Error Reporting (Doctor Watson) may incorrectly assign blame to the owner of the executable.

*Mitigation strategy*

Avoid modifying the executable file on disk by using noninvasive ASOC techniques that modify program behavior at load-time or runtime.

## Security Considerations

Some ASOC techniques can be used to introduce security violations. For example, access control checks can be bypassed, sensitive data can be exposed, and arbitrary code may be executed.

*Mitigation strategy*

Security is probably the biggest reason why ASOC techniques need to be tightly integrated with the .NET Platform instead of grafted on using a variety of ad hoc techniques. The .NET Platform can ensure that only authenticated and secure ASOC tools can modify a program.

## 6.2 Developer and User Comprehension

### Semantics

Inheritance has undoubtedly helped developers in tackling difficult design problems more naturally. But the first time they tried to test a method that was "nowhere to be found" (i.e., it was actually declared somewhere up the inheritance hierarchy), they surely cursed object orientation. ASOC presents very powerful mechanisms for dealing with crosscutting which at times might make a developer wonder "where the magic takes place".

*Mitigation strategy*

Just as with object-oriented programming, ASOC techniques require training to comprehend and master, and better tool support. IDEs and debuggers must provide a view of the program that makes it easy to understand how the ASOC-enabled functionality is integrated with the program. This will help the developer understand the ASOC-enabled program, predict its behavior, and avoid ASOC-related mistakes.

### ASOC for Everyone

Microsoft developers are considered "Einsteins" and will have little trouble understanding and taking full advantage of ASOC. In fact, some groups at Microsoft are already using ASOC techniques. However, ASOC must be understandable by the "Morts" of the world, for example, business people whose job requires them to do a little programming.

*Mitigation strategy*

ASOC features and APIs must be designed with the intended audience in mind. A programmer that simply wants to turn on monitoring and logging never need know that it was implemented using ASOC mechanisms. ASOC-enabled middleware have successfully marketed ASOC-based extension mechanisms by limiting the extensibility power to message interception and filtering.

# CONCLUSION

We explained how mainstream languages like C++, C#, and Java provide limited mechanisms for modularizing software. This results in *crosscutting concerns* being scattered across multiple files and tangled with code related to other concerns. Advanced separation of concerns tries to capture this latent modularity potential.

We provided real-world examples of companies that are using ASOC techniques to solve real business problems and add revenue. However, with direct support from Microsoft, we can do much more. By adding first-class support for ASOC to the .NET Platform, Microsoft will be able to exploit ASOC to its fullest and allow others to do the same.

# ACKNOWLEDGEMENTS

We would like to thank the co-authors

- Marc Eaddy <me133@columbia.edu>, Columbia University,
- Alan Cyment <acyment@dc.uba.ar>, University of Buenos Aires,
- Pierre van de Laar <pierre.van.de.laar@philips.com>, Philips,
- Fabian Schmied <fabian.schmied@gmx.at>, Vienna University of Technology, and
- Wolfgang Schult <Wolfgang.Schult@hpi.uni-potsdam.de>, Deutsche Post World Net.

We also thank Pascal Dürr, Sonja Keserovic, and Christa Schwanninger, for reviewing the whitepaper.