AOP and the DataServices Project

Adrian Kosmaczewski

2007-03-27

Five years ago I worked as a Software Engineer for a startup, based in Geneva, Switzerland, which had the goal of creating a web-based systems management console, to control and monitor the status of large computer installations, much like Microsoft SMS (Systems Management Server) does. This tool would eventually benefit from being a web-based application, and as such it could be used from anywhere, without having to install a "fat client"; just launch a browser, point to a particular URL, and you are done.

During the project, I was able to work towards the creation of the first AJAX application I've ever seen (this was 2002!), and also, to use Aspect-Oriented Programming techniques for the first time.

The Architecture

This tool was designed as a three-systems project:

- On the server side, the back-end: it used "software spiders" that crawled your local network for information about the domain computers, using their WMI (Windows Management Instrumentation) services to gather useful information such as processor type, network card address, memory size, CPU usage, any kind of information. This information was sent to the server, where it was stored in a SQL Server 2000 database, for quicker retrieval and reporting.
- On the client side, a web based application, running on top of IIS (Internet Information Services) and implemented as an ASP application (Active Server Pages). This tool was the first AJAX (Asynchronous JavaScript and XML) application I had seen at the time (2002!), giving users a real-time experience of what was going on in the network; for example (this was impressive) you could just plug a new computer on the domain, and a couple of seconds later, the information would appear in your web browser automatically, without having to refresh the page (the client AJAX component polled the back-end every 5 or 10 seconds for updates).
- In between both ends, the ASP application connected to the SQL Server backend using a COM+ "proxy" component, which was the only means of communication between both subsystems.

The development team was split in three teams, one working on the spider+database backend, and the other (where I was) working in the ASP client application. Finally, a highly-skilled C++ programmer had the task of building the "proxy" component that allowed both systems to communicate. This component was critical, and was designed for extreme high performance. Its design would deserve an article just for itself... just to mention, it featured transactional requests, complex event and feedback models and had extensive queuing capabilities, via MSMQ (Microsoft Messaging Queue).

Just for the record, regarding the methodologies used in the project, it is worth noting that the whole of the project was managed using the Rational Unified Process (RUP). It must be said that in the case of this startup company, this methodology was an "overkill", and documentation was never kept in sync when changes were done to the architecture. Being a startup company required to be able to adapt the software as quickly as possible, and often this meant skipping the documentation process.

The DataServices Project

The COM+ component had extremely high throughoutput and small response times, but for the sake of interoperability, it required the ASP client application to send and consume complex XML requests and responses, using an industry standard called "CIM" (Computer Information Model). This XML standard, designed by the DTMF (Distributed Task Management Force) is part of the WBEM initiative allowing for XML-based interoperability of computer management systems. It is worth noting that the Microsoft Systems Management Server (SMS) is based on this standard as well, and that the Windows WMI services are also based on these standards.

The problem, for the web applications team, was then to create long and complex XML requests to send to the COM+ proxy component. The web developers, already struggling with a complex AJAX-based UI, had real trouble to figure out the different parameters for these calls, and to create the proper XML structure (The CIM XML format is an extremely complex tree structure, with several layers of information and a high level of redundancy). This led to longer development cycles, since every new use case implementation required a good deal of new requests to be created to the server, with low reuse of already existing requests.

This is how the "DataServices" subproject was born: to create a layer of abstraction between the COM+ component and the dynamic HTML application. The goal was to provide a simpler way to connect the backend with the user interface, so that the development team in the UI layer could concentrate in the user experience, rather than in creating long XML documents. This was dragging development times, and was a well-known bottleneck for the project.

To avoid impacting the current ASP application, and to leverage XML capabilities, this project was created as an ASP.NET application, using the recently released .NET Framework, version 1.0 (released in February 2002) and the C# language.

This ASP.NET application would handle requests from the dynamic HTML application, but providing an API that would be much simpler for developers; for instance, instead of having to create long XML strings to retrieve the list of processes running in a given computer of the domain, the developers could just call “ComputerSystem.GetRunningProcesses(machineName)” (“facade” methods) and that would do it.

Aspect-Oriented Programming

Once the DataServices subproject began growing, we noticed that we were performing the same operations everywhere, for each “facade” method call:

- Deserialization of requests
- Security checks
- Instrumentation (mainly logging, but also performance management and throughoutput)
- Back-end connection management (opening, pooling, closing)
- Error management
- Serialization of responses

The same lines of code were repeated around almost every internal operation done by the DataServices infrastructure, but this code could not be properly encapsulated in an external component, to be reused instead of duplicated.

More or less at the same time, one of the team members read a paper about the newly-born idea of “Aspect Oriented Programming” (AOP), and about the basic AOP capabilities of the .NET Framework (Dharma, Fell and Sells, 2002).

Gregor Kiczales from Xerox PARC, one of the creators of AOP, explains that

> “We have found many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in”tangled” code that is excessively difficult to develop and maintain. We present an analysis of why certain design decisions have been so difficult to clearly capture in actual code. We call the properties these decisions address aspects, and show that the reason they have been hard to capture is that they cross-cut the system’s basic functionality. We present the basis for a new programming technique, called aspect-oriented programming, that makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code.”

(Kiczales et al., 1997)

The following two images highlight the location, in the Apache Tomcat project, of different sections of code that handle URL pattern matching and logging:

(Source: Hilsdale, Kiczales, 2003)

(Source: Hilsdale, Kiczales, 2003)

The idea of AOP is to provide a mechanism to centralize these lines of code in a single location, which would provide better management and maintainability in the future; Jacobson further explains that

> “Even though peer use cases are separate during requirements, their separation is not preserved during implementation. The realization of a use case touches many classes (scattering), and a class contains pieces of several use-case realizations (tangling). As a serious consequence, the realization of each use case gets dissolved in a sea of classes.”

(Jacobson, Ng, 2005)

(Source: Jacobson, Ng, 2005, page 33)

Regarding DataServices, suffice to say that AOP seemed to us as a good solution for our problem. Tempted by the discovery of a new paradigm, we decided to give a try to AOP, and the results proved to be worth the effort. In our case, we intercepted each message sent to the classes implementing the use cases (like the ComputerSystem.GetRunningProcesses(machineName) example above), “injecting” behavior before and after the associated method calls, providing the services enumerated above (instrumentation, error handling, etc).

Using DataServices, you could define methods like this:

```
[Log()]
[RequiresRole(Role.Admin)]
[Database(ConnectionType.SQLServer)]
public bool CreateRecord([RegExp("[a-z]*")] string name,
[Range(1, 99)] int age)
{
// just open the connection and insert,
// no further checking needed!
}
```

As you can see the `CreateRecord` method contains .NET attributes that define the valid ranges of execution for the parameters, and has some other attributes that define pre- and post-conditions to be checked prior to execution. This allowed us to centrally manage a quite large framework (around 20 classes, or around 100 quite complex methods) and centralize the debugging, security, logging and parameter checking routines into a single location.

Moreover, using configuration files, one was able to “inject” (“weave”, using AOP terminology) more behaviors into this mechanism without having to recompile the application (this is known nowadays as “Dependency Injection” or “Inversion of Control” - Fowler, 2004), while the core “facade” methods only contained specific instructions related to the business rules that they implemented. The code was lighter, easier to read and maintain, and largely self-documenting.

It is worth noting, however, that we used a very small subset of the functionality that AOP provides, that is, the runtime interception of messages sent to objects and weaving aspects before and after method execution:

> “Another way to understand AOP is in terms of how it works. A common misconception associated with this perspective is to equate all of AOP with just one part of the supporting mechanisms. This kind of error is analogous to saying that OOP is just abstract data types. Probably the most common mechanism error is to equate AOP with interceptors.(…) It’s true that AOP does use functionality like interceptors and Lisp advise. It also incorporates techniques from reflection, multiple inheritance, multi-methods and others. However, AOP has an explicit focus on crosscutting structure and the modularization of crosscutting concerns. The rule of thumb to avoid mechanism errors? Remember that AOP is more than any one mechanismâ€”it’s an approach to modularizing crosscutting concerns that’s supported by a variety of mechanisms, including pointcuts, advice and introduction.”

(Kiczales, 2004)

It is also interesting to note some drawbacks about our approach:

- Performance: the runtime-based interceptions of method calls was heavy, adding a 10-factor more processing time at each method execution. However, since our system was not designed for high number of users or transactions per second, this was not a showstopper, at least not in that phase of the project.
- Debugging: the Visual Studio .NET 2002 debugger (used at the beginning of the project) had some trouble figuring out the “jumps” between the “normal” methods and the AOP code (which we called “hyperspace” in our jargon, since the call stack of the weaved code seemed to appear and disappear completely, almost magically, while debugging). The workaround was to set up breakpoints in strategic places around the code.
- Microsoft support: while .NET provides basic AOP capabilities, such as message interception and context-bound objects, these features were undocumented and explicitly non-supported; however, the code created back in 2002 still compiles perfectly well in later versions of the .NET Framework.

Conclusion

AOP helped us to have an extremely high productive environment, and in only 4 months we had a working system (4 people were working full-time in the project), providing an enourmous amount of functionality, and enhancing the current system.

Unfortunately, though, the project could not be finished completely, since the venture capital group that financed the company cut all financing unexpectedly and all development tasks stopped in 2003. Nevertheless, it was a highly rewarding experience, that gave me a practical insight into complex OOP, AOP and architectural design, as well as into project management techniques.

References

Fowler, Martin, “Inversion of Control Containers and the Dependency Injection pattern” [Internet] http://www.martinfowler.com/articles/injection.html (Last accessed July 2nd, 2006)

Hilsdale, Erik; Kiczales, Gregor et al., “Aspect-Oriented Programming with AspectJâ„¢”, Xerox Corporation, http://www.ccs.neu.edu/research/demeter/course/w03/lectures/lecAspectJ-w03.ppt (Last accessed July 2nd, 2006)

Jacobson, Ivar; Ng, Pan-Wei, “Aspect-Oriented Software Development with Use Cases”, ISBN 0-321-26888-1, Addison-Wesley, 2005.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J, “Aspect Oriented Programming”, Springer-Verlag, 1997, [Internet] http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf (Last accessed July 2nd, 2006)

Kiczales, Gregor, “Common Misconceptions”, Dr. Dobb’s Magazine, February 10th, 2004, [Internet] http://www.ddj.com/showArticle.jhtml?articleID=184415113 (Last accessed July 2nd, 2006)

Shukla, Dharma; Fell, Simon; and Sells, Chris, “Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse”, MSDN Magazine, March 2002 [Internet] http://msdn.microsoft.com/msdnmag/issues/02/03/aop/ (Last accessed July 2nd, 2006)