Using AspectJ to Eliminate Tangling Code in EAI-Activities

Arno Schmidmeier

Sirius Software GmbH
Lohweg 9
91217 Hersbruck
+49/91 51/ 90 50 30

Arno.Schmidmeier@web.de

ABSTRACT

Enterprise Application Integration imposes various non-functional requirements on integration teams and application manufacturers, which are hard to separate with OO languages and tools. This paper describes how the overall integration effort has been dramatically reduced by using AspectJ [1] to integrate different Sirius EOS Service Monitors [16] in a NGOSS [18] compliant EAI architecture realized with Vitria BusinessWare [22].

Keywords

Aspect Oriented Software Development, EAI, enterprise application integration, AOP, Aspect Oriented Programming, AspectJ, NGOSS, coding guidelines

1. INTRODUCTION

The economic competition and challenge to launch and support new services requires a constant integration of new applications into current legacy systems. Today these integration projects are normally solved as expensive professional service tasks. The majority of the money is spent on solving non functional, technical concerns and problems. Reducing the integration cost is a major strategic objective in many business domains.

This paper first describes a predominant solution concept, and how this concept affects current object oriented "off-the-shelf" software products and existing applications. It explains why this approach creates a number of crosscutting concerns at the application level; and, how these crosscutting concerns were easily modularised and then localized by using the general purpose aspect language (GPAL) AspectJ. Sirius Research used this approach very successfully in several real world projects.

Sections 4 and 5 discuss our most important coding conventions and the features we found lacking from AspectJ 1.0. Finally we discuss modifications to our development process and development activities as a result of adopting AspectJ.

2. Current Solution Approaches

The naive approach to integrate the n^{th} application with $n-1$ other applications has an unacceptable drawback: it requires $O(n^2)$ integration projects.

The predominant solution is to integrate all applications via a common message oriented middleware platform used as an information bus, to use common shared infrastructure services, and to use common business modelling tools like Business-rule engines, state machines, etc. to model and realize the process flow and define a common shared data (object) model. If such architecture is in place the initial integration cost is reduced to $O(n)$. New applications can be added with cost $O(1)$, not $O(n)$. Several domain groups have standardized such architectures, for example the Tele Management Forum (TMF) [17] with NGOSS. The NGOSS architecture is left technology neutral. It can be realised with a combination of two extreme approaches: the first is to use one of the multiple "off-the-shelf" EAI products (e.g. Vitria BusinessWare), and the second is to combine standard middleware platforms with multiple standard infrastructure services and commercial off the shelf applications, which are responsible for modelling and realising the business flow (e.g. ILOG JRules [6]).

The TMF defined several realisation guidelines for the first approach. Currently a guideline exists for CORBA [19], and XML [20]. A separate guideline [11] was released for J2EE from Sun Microsystems in cooperation with TMF.

The decision to leave the NGOSS architecture technology neutral offers the possibility to more easily adopt new protocols and new technologies as well as to integrate legacy applications. The drawback is that application providers must support an individual customized EAI environment for each NGOSS adopter. Normally these environments differ dramatically in their deployed infrastructures, their process flows, and shared object model. There is no easy way for interoperation. For example, products implementing Java Specification Request (JSR) JSR 89 [7], JSR 90 [8], JSR 91 [9] or JSR 142 [10] cannot be directly integrated in EAI architectures based on CORBA [19].

As a result, an application provider must realize the following technical or architectural requirements for each of his customers separately:

- Merging internal information buses with the information bus from the EAI architecture.
- Integrating information exchange sequences from the EAI project into the existing sequences from the

application with the expected quality of service. (For example, a pull based application may be required to be changed to an application which implements the observer pattern [4])

- Adopting common infrastructure services as opposed to those of the native enterprise application.
- Using a central security service, which often requires a different security protocol and different security concepts,
- Supporting a common shared data model that differs from the native data model of the application.
- Dealing with different transactional policies and behaviour
- Finally, implementing excessive crosscutting logic, such as caching of stewarded data to meet expected non-functional requirements for customer specific EAI support.

These requirements are currently fulfilled:

- As expensive professional service tasks, on top of the standard APIs, where the realisation of these requirements does not pollute the application core with customer specific elements, but faces challenges to overcome performance penalties and limitations of the available APIs
- Or, as a bunch of customer specific, tangled code fragments cluttered all over the applications
- Or, a combination of both.

However, none of these approaches is satisfying. They are too costly, provide a maintenance nightmare for the application providers, or violate the architectural integrity of the application. For this reason we used AspectJ to modularise and consequently solve these technical and architectural requirements in the EOS application suite.

3. Overview of our solution

In the EAI-projects discussed in this paper, all application providers were required to integrate their whole application or submodules of their enterprise applications with Vitria BusinessWare. In the project “SLM for Wireless IP” [21] for example, Sirius Software integrated their QoS-mediation and their service monitor modules, Amdocs their IPDR billing, Contract &Order Management, and their billing module, Cvidya their optimiser, Sodalia their provisioning application and Edocs their eCRM-System via Vitria BusinessWare. BusinessWare was responsible for providing the common EAI integration infrastructure.

None of these projects required a central transaction policy or a common security infrastructure. A central transaction policy in the EAI-projects was replaced by excessive use of pre and post conditions. It was a trivial task to realize these checks using AsectJ’s advice. . Implementing a central transactional policy was practically not doable, because the majority of the underlying network equipment, used network and element management systems and design patterns are based on best effort policies.

The requirement for a common security infrastructure was dropped, because a sufficient secure computing infrastructure could be more easily and cheaply created by the operating telecom companies with a closed WAN.

3.1 Using Aspects to Merge Our Internal Information Exchange with the Information Exchange on the Vitria Information Bus

In the projects, we had three different scenarios:

In the first, we had to distribute internal observer notifications to the Vitria bus.

In the second, we had to implement a kind of observer, for entities which strictly entailed a create select, update, delete lifecycle, where the change notification and parts of the changed data must be placed on the Vitria bus.

In the third scenario, we received information from the Vitria bus, which we needed to process and then put the original information together with the results of the computation back on the bus.

The first scenario was implemented by adding advice around the execution of the internal update methods responsible for distributing the change notifications. We decided to implement this approach as aspects and not as standard OO-solutions for following reasons:

- We had verified empirically in previous projects the theoretical result from [13 that aspects are superior to the OO-observer solution.
- We did not need to insert to any code in the original code base. So we could avoid creating any dependency from the EAI-code to the code base of the application.

For the second scenario, we used around advice on all join points which could cause relevant changes. The advice could easily extract the critical data (which may be changed) via inexpensive internal method calls. First the advice fetched and stored the old values, then proceeded with the calculation. Finally the advice fetched the potentially updated values and compared them with the original one; and, if they found a difference, they dumped the change notification to the Vitria bus.

The third scenario was implemented by objects listening as CORBA-objects on the bus. Upon receiving notification, they invoked the relevant internal methods, extracted the result, and dumped the information back on the Vitria bus.

We could have use basic object orientation for that task, but we needed aspects for following reasons:

- The creation of the CORBA objects was initiated by advice, which was bound to some methods during the start-up phase of the application. Thus we could easily ensure that the application was correctly connected to the bus as soon as it was ready to perform its work, and we did not need to change the bootstrap source code off the application.

- Some data structures should piggy-back some extra data, which was needed to perform the transformation from the common shared data model to our internal one and back again.
- We had to extend the functionality of some objects, to satisfy several interfaces for the Vitria integration.

The last two tasks were realized by introduction.

3.2 Using Aspects to Integrate the Information Exchange Sequences from the EAI Project into the Existing Sequences of the Application.

This task was solved for free with the aspects, which integrated the information exchange sequences. The change from a pull to a pull-push scenario was performed without any of the traditional problems. For example, we did not need to write any change logs, which could be pulled or observed in short intervals, we did not use any caches, and we did not need to work with any tricky functionality of the of the application infrastructure (e.g. triggers, stored procedures, etc.) Still the actual performance of the implementation exceeded the performance expectation of the customers by a factor of ten. A previous project based only on standard OO performed only half as well.

3.3 Use of Common Infrastructure Services Instead of those Native to the Enterprise Application.

Access to infrastructure services is performed via proxies in our application suite. The proxies are implemented as singletons. All changes were isolated inside these singletons. All but one of these singletons were reusable, with only configuration parameters needed to be changed. The remaining singleton needed to be fully replaced. By using a plain OO-solution we would have had to introduce a more sophisticated bootstrap mechanism, change all static *getInstance* calls to the new class, or replace the original one, with the modified new one. The first two options would require changes in the existing code base, and the last option would break our build system. We decided to use a simple around advice, around the getInstance() calls, which returns the correct singleton.

3.4 Using Aspects to Support the Common Shared Data Model.

The individual tasks of each application in an EAI project are such that the common-shared data model of the project is normally fairly close enough to the object models of the applications. Therefore a transformation is usually possible. There are at least two transformations necessary.

The first one is the data representation, (e.g. Java-Object to CORBA-struct and vice versa, or Java-objects to XML and vice versa) and the second one is the transformation of the class structure. For a more efficient or easier to implement transformation it is often necessary to piggy-back additional data, which is normally not used inside the application. E.g. detailed customer information, like contact information, etc are not used inside provisioning or service monitoring applications, however this information needs to be shared between CRM and billing applications, so it will be stored on the bus. In a classic OO-framework, this information must be additionally stored in the data transformation layer, or it must be retrieved expensively via the bus from the application which stewards the data. Alternatively, the existing class hierarchies have to be touched, so that this information can be piggy-backed inside the application.

Often we could use static crosscutting from AspectJ to add the piggy-backed data to the existing class hierarchy. If static crosscutting was not an option, we used aspects declared as perthis, pertarget, or percflow to store the additional information. The rest of the translation from the common shared data model to the data model of the application was performed using object oriented programming.

3.5 Using Aspects to Implement Non-functional Requirements

Finally as in most enterprise projects we encountered non functional requirements, like logging; auditing of relevant changes; sophisticated exception handling; retry of failed requests; fail over; caching of stewarded data by other applications, etc.

Each of these requirements can be implemented in a straightforward way with simple OO. Unfortunately, a simple OO-approach creates a bunch of tangling code. A high reliable and well performing plain OO-implementation requires an overhead of 50 to 100 lines of code for each method which accesses the EAI infrastructure. This implementation is normally added by a high error prone, hard to maintain and extend copy and paste session. Often the macros facilities of modern IDEs are also used for this task. Several projects and tools also use custom compilers, often based on commercial or freely available CORBA compilers with an extensible backend (e.g. omniorb [14]), to create customized stubs which realize these requirements. With the use of aspects we were able to reduce the required lines of code dramatically and still avoid implementing our own customized compiler.

4. Lessons Learned for Coding Guidelines

AOP is a very young programming paradigm, so most of the team members did not think in aspects when the projects started. We needed therefore several coding guidelines. These guidelines helped us to maintain a more manageable codebase.

All of these guidelines, with the exception of one, deal with the application of aspects. We made one modification to our java-coding guidelines: a programmer can access private member variables directly; he does not need to use the set and get methods. We liberated these regulations based on the experience from previous projects that we could implement on demand all of the flexibility-benefits of the set and get methods by applying specific advice.

We were very restrictive on the use of cflow and cflowbelow. It is prohibited to use it, to narrow a set of pointcuts. E.g. it was not allowed to use following pointcut statements:

```
pointcut pc():
   pc1()&&!cflowbelow(pc1());
```

These types of statements were prohibited, for the rule of least surprise. Because, what the developer really wanted in our scenario and codebase is:

```
pointcut pc():pc1()
    &&!cflowbelow(
        firstaroundadvice(pc1())
  );
```

where firstaroundadvice is a non existing pcd, which defines the execution of the most dominating around advice at this pointcut. We may drop this coding convention as soon as we have a pointcut discriminator for advice and a more powerful advice precedence concept.

Additional we did not permit the use of cflow-pointcuts, if the flow of execution leaves the actual compilation unit, in which some of the pointcuts are defined. We emulate the behaviour with (thread) local variables and if-pointcuts.

In addition, pointcut declarations should not rely on coding conventions. We recognized that for large-scale projects and application, with a great amount of legacy java-code, the coding conventions are not as reliable and consistent as required for the use in pointcuts. In our projects, this was caused mostly by different coding conventions in different technologies, (e.g. difference of mapping of attributes, Java-CORBA binding [15] vs. java-Beans), changing style guides and change of knowledge of the developer over the time.

Pointcuts, which are used in advices, shall be as restrictive as possible and reasonable.

In addition, we do not permit advice on global pointcuts which use patterns in a way that may result in a globally matching pointcut. These types of pointcuts are only allowed for declare error and declare warning. Each non-excluding pointcut declaration, used in a pointcut, must be narrowed by a class, or a package. The debugging overhead of accidental joinpoint clashes resulting in wrongly "advised" code, outweighed the benefits from the automatically fitting to newly added code in other modules.

As an example, the advice

```
before(): call(* *(..))&&!within(A){
      System.out.println("before");
}
```

must be written as

```
before(): call(* MyObject.*(..)){
      System.out.println("before");
}
```

to avoid potential double advice recursion.

In addition, we limited the use of the body of advice. The body of advice should contain as little code as possible. All this code should be refactored to functions of the aspect. This offered us the possibility to advise the body of the advice. So changes could be performed more easily and different concerns, which crosscut advice, could be separated more easily. We may drop this coding convention as soon as we have a pointcut discriminator for advice.

Advice that must be woven to multiple classes in different packages is defined in an abstract aspect, where each abstract aspect implements at most one concern. The advice is bound to abstract pointcuts. The abstract pointcuts are made concrete in final aspects - each of them containing only the pointcuts for one specific module, which implements one business concern.

During our first AspectJ-based project, we recognized that most of our aspects were advising the same pointcuts, so we traded initially tangling code against tangling pointcuts. We believe now, that tangling pointcuts are not better than tangling code. We recognized soon, that the joinpoints, which were captured by the tangling pointcuts, reference specific architectural places, e.g. methods, which are exposed to or using Vitria. All of these tangling pointcuts have been refactored according to our style guide.

This required the creation of one aspect for each module of a business concern, which defined all relevant architectural pointcuts of this business concern. Each of these aspects had only public atomic pointcuts. We call these aspects reference-aspects and the pointcuts reference-pointcuts.

This also required, that the pointcut declarations of the concrete sub-classed aspects were changed to reuse as many of the pointcuts of the reference aspects as possible.

With this concept, we could eliminate tangling pointcuts and tangling code. We are only left with tangling aspects, but this is at least a magnitude better than tangling code.

If we re-factor code, e.g. we rename a method, we have now only to update the relevant reference-aspect. We hope for extensible pointcuts as proposed on the AspectJ mailing list, because this approach would eliminate also the tangling aspects.

5. Missed Language Features in AspectJ 1.0

The projects were based on the language specification from AspectJ 1.0. This version lacks several features, which would be quite useful for this job. The AspectJ-team plans to address several of this features in AspectJ 1.1 and they are currently heavily discussed on the AspectJ mailing list [2]. We feel that all of these issues are in the composition of advices.

A big obstacle is the current dominates precedence rule. With dominates alone, a very concrete aspect cannot specify, that it should be dominated by a very general aspect. For example,. an observer-aspect must be dominated by a general transaction handling aspect. As a result of this the general aspect must specify that it dominates the very concrete aspect, which violates the need to know principle.

We missed a pointcut descriptor for advice. Several advice-bodies should be advised for a clean design. We could use our coding style guides as a workaround of this missing language feature for before and after advice. Around-advice caused us the biggest trouble. We could inline them manually, which breaks the concept of separation of concerns, or we had to emulate them with dominating advice. The approach with dominating advice resulted in tangling pointcuts or tangling aspects.

The lack of extensible pointcuts forced us to emulate that feature with abstract aspects and concrete aspects, which contain only pointcuts. Our codebase showed us that an aspect language with both of these concepts, extensible pointcuts and abstract aspects, would be better.

6. Adjusting the Development Activities and Process

As in all of our current AspectJ based projects, we faced the same general early adopter obstacles. We had to adjust the development process to cover several of these obstacles and had good workarounds available for most of the issues. Most of the remaining obstacles are early adopter tooling and language feature issues, which we have already discussed. We are quite sure they will be fixed as the tools and the language matures. Several may be already fixed when this paper appears.

One of the biggest obstacles we faced in the past is the lack of good non-trivial examples, good articles and books, education programs and training courses. The number of good examples increases through the ongoing projects. We kept a sample reference file, where we listed locations in our code base, which were simple and declarative enough to work as good samples. Additional good samples were the test cases, which tested the functionality of the aspect.

As soon as we have identified a new and interesting use of an aspect, we created a short talk, which covered that usage idiom. We named them communication units. As new members entered the team, we just “replayed” a selection of the most used and most important communication units. As a result, we had our developers up to speed in a very short timeframe, based on the individually tailored training and mentoring courses. Several communication units helped us to identify and separate general non-functional requirements, inside the process flow of the common shared data model. The most famous ones were for asynchronous acknowledgement, and for error messages. These crosscutting concerns could only be modularised by a very dense combination of object-oriented patterns and aspect-oriented idioms. For each new type of combination, we created internally a new communication unit.

The increased understanding about aspects, the new idioms and their applications was necessary to overcome the lack of a good debugger. Debugging aspect-object interactions was and is the greatest challenge during the development process.

Many of the problems we encountered arose from the incorrect application of advice. E.g. a wrong order of advice at a specific set of pointcuts, an advice was weaved at code fragments, where it should not, or missing advice caused by wrong pointcut descriptions. The rest were conflicting behaviour of aspects and objects at some specific pointcuts.

Developing, debugging and testing a concern were simple and easy. In addition, the separation of concerns was finally simple enough. The biggest challenge was the assembly of all the different concerns. We feel a need to officially keep track of when, how and by whom the different concerns should be integrated. We have a great need for simple to use diagrams, which can easily represent and list the interactions of the concerns, in a non-tangling way. We would like to use these diagrams to document and track the integration.

Standard refactoring activities required slight modification to the classical refactoring process defined by [3]. After the developer knows which code fragments he or she wants to re-factor, the programmer asks the ajcbrowser (integrated in his IDE) how the code is influenced (and by which) aspects. After the re-factoring he compiles the codebase and verifies manually with the help of the ajcbrowser, that all influences are still the same, if they differ he has to fix some of the pointcut declaration. For instance, to ensure that no advice is lost or duplicated. We wish for more sophisticated tool support here. We want an aspect aware re-factoring browser.

Summing up, our main modification to our current process was an extensive emphasis on coaching, training and pattern/idiom hatching.

7. Results, Conclusions, and Next Steps

We could modularise and separate all non-functional concerns regarding the integration of a large-scale enterprise application into an EAI scenario by the joint application of OOP and AOP. This approach was a total success. The number of required lines of code for these EAI tasks was approx. 95% less than on previous projects, which had a comparable complexity and used only object oriented techniques. Additionally the non-functional requirements were more significantly fulfilled. For example, the downtime of the EAI bridge could be reduced by a factor of 100 with the AOP solution and the performance increased by a factor of two. We could reuse most of the abstract library aspects between the two projects. Despite the expensive coaching and idiom hatching activities, we needed only one quarter of the time than comparable projects, with equal developer resources. We are confident that the realisation time can be still reduced approximately by a factor of four, if there is no need for additional education and the integration aspect library is in a final state.

We currently recommend strongly having an AOP experienced developer on board for mentoring and training activities.

We concluded that the current EAI integration approaches require AOSD for successful and efficient EAI projects. We expect that the approach of this paper will also work with other GPAL, e.g. Hyper/J [5].

We also suspect, that there are many crosscutting concerns in current EAI architectures, and shared object models. We believe that these crosscutting concerns can be modularised by AOP. This would enable a new breed of EAI architectures and EAI products.

8. Biography

Arno Schmidmeier is Chief Scientist at Sirius Software GmbH, where he is responsible for the commercial adoption of new technologies like AOP. He and his team has successful deployed several large-scale projects based on AspectJ. He and his team offer consulting services for use and introduction of AspectJ in commercial projects.

He is an independent expert on JSR 90 and represents Sirius Software in the TeleManagement Forum.

9. Literature

[1] AspectJ Team, The AspectJ Programming Guide, http://AspectJ.org/doc/dist/progguide/index.html , September 2002

[2] AspectJ-mailinglist, http://aspectj.org/pipermail/users

[3] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, Refactoring, Addison-Wesley, 1999

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Addison-Wesley, 1995

[5] HyperJ, http://www.research.ibm.com/hyperspace , Dezember 2001

[6] ILOG Rules, http://www.ilog.com/products/rules , October 2002

[7] Java Specification Request 89, OSS Service Activation API

[8] Java Specification Request 90, OSS Quality of Service API

[9] Java Specification Request 91, OSS Trouble Ticket API

[10] Java Specification Request 142, OSS Inventory API

[11] Java Specification Request 144, OSS Common API

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Ch., Lopes, Ch.V., Loingtier, J.-M., Irwin, J., Aspect-Oriented Programming, in: Proceedings of ECOOP 1997, Jyväskylä, Finland, June 9-13, 1997, pp. 220-242, in: Lecture Notes in Computer Science, vol. 1241, Springer, 1997

[13] Martin E. Nordberg, Aspect-Oriented Dependency Inversion, Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA, 2001

[14] OMNIORB Team, http://omniorb.sourceforge.net/, October 2002

[15] OMG, IDL to Java Language Mapping Specification Version 1.2, August 2002

[16] EOS Monitor Units, http://www.sirius-eos.com, October 2002

[17] TeleManagement Forum, http://www.tmforum.org , October 2002

[18] TeleManagement Forum 2002, TMF 053 v2.5: TM Forum NGOSS Technology Neutral Architecture

[19] TeleManagement Forum 2001, TMF 055 v1.5 NGOSS Phase 1 Technology Application Note – CORBA

[20] TeleManagement Forum 2001, TMF 057v1.5 NGOSS Phase 1 Technology Application Note – XML

[21] TeleManagement Forum 2002, TMF 837 Service Level Management for Wireless IP, Interface Implementation Specification

[22] Vitria BusinessWare, http://www.vitria.com/products/platform, October 2002