Aspect Oriented DSLs for Business Process Implementation

Arno Schmidmeier

AspectSoft
Lohweg 9
91217 Hersbruck
Germany
+49/91 51/ 90 50 30

Arno @aspectsoft.de
http://www.aspectsoft.com

ABSTRACT

Domain specific languages (DSLs) are a very important approach to raise abstraction and enable an efficient communication between business experts and application software developers. Some DSLs could benefit from the application of ideas from the AOSD world. Therefore, it is a natural idea to enhance an existing DSL with AOP based programming ideas. This paper describes such an attempt. The approach has been successfully applied in several commercial projects. Commercially available DSLs in the domain of application integration, service orchestration and business process management have been enhanced with composition filters.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features

General Terms

Management, Documentation, Design, Languages, Theory

Keywords

Aspect Oriented Software Development, AOP, DSL, BPM, BPEL, Enterprise Application Integration, Process Engines, Workflow Engines

1. INTRODUCTION

Domain Specific Languages (DSL) are either textual or graphical languages for a given domain. They should simplify the software development in a specific domain

- by using usual paradigms, concepts, terms and graphical representations of the domain
- and by creating the suitable abstraction and simplification for the domain.

DSLs are often crafted to a specific domain, to a family of projects. However, creating its own DSL is often not possible for commercial reasons, e.g.:

- A viable Return of Investment can not be justified for the creation of the own DSL.
- The big upfront effort to create a DSL may delay the time to market.

This is true especially for small to medium or for agile teams. The usual alternative approach in industrial settings is the selection of a suitable commercial of the shelf language and its interpreter or compiler, which provide a suitable abstraction and which can be easily handled by the domain specialist. Then this language is the DSL of the project by definition.

This approach is often performed in the domain of Enterprise Application Integration (EAI) and Service Orchestration as well as Business Process Management (BPM). Projects in these domains use often a graphical defined Finite State Machine (FSM) as their DSL. Such a DSL is often marketed as Workflow Engine, Process Engine or Business Process Engine. You can find several open source (e.g. [6]) and commercial implementations (e.g. [5], [7], [8]), and even several specifications (e.g. [20]), for this type of DSLs. The next section describes these DSLs, the architecture of their interpreters and the development platform.

These DSLs can significantly simplify the development of small solutions. High level business processes models (e.g. epc models developed in IDS Scheers toolset [9]), can often be directly transformed in the FSM. However, implementing larger or complex processes with these engines is normally a cumbersome and problematic task, for the reson of:

- typical process specific crosscutting concerns,
- typical domain specific crosscutting concerns,
- and typical technical crosscutting concerns

The third section provides some typical concerns and shows how these specific concerns can easily damage simple process models through scattered transition actions and functionality.

The forth section describes some simple enhancements to the FSMs and their interpreters, which enhance the FSM to a composition filter based FSM, which is capable to modularize these crosscutting concerns. The modular implementation of the sample from the third section is outlined in the fifth section. The sixth section contains a summary, conclusions and future work.

2. Description of the DSLs

The DSL for a project in the domains Enterprise Application Integration, Service Orchestration or Business Process Management are often consisting of a dialect of a Finite State Machine (FSM). The range of dialects has a great variety. It covers amongst others executable UML state diagrams [10], BPEL engines (e.g. [11]) and workflow engines [6][1]. A FSM is normally programmed with State Transition Diagrams. Transition conditions normally consist of the occurrence of an (external) event. If this is insufficient, these conditions can be further specified by Boolean expressions. Figure 1 shows a simple sample of such a FSM.

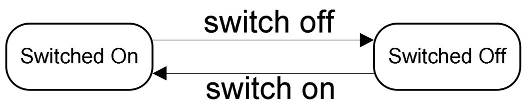

Figure 1

Each FSM has its state and set of attributes, which can be defined by the user.

The FSM can perform some actions. Each action can perform several tasks, e.g. sending some events, invoking a system, a service, some low-level language sniplets (e.g. Java or C#-Code fragments) or changing and setting the value of an attribute of the FSM. Following action types are usually available in such a DSL:

- Entry action: executed whenever a state is entered
- Exit action: executed whenever the FSM exits a state
- Transition action: executed, whenever a transition fires.

Typical available commercial FSMs may provide some "syntactic modelling sugar", such as Action States and Activity States, Action States fire automatically a transition to the next states. Activitiy States are like normal states, but they provide often some support and hooks for GUI-frameworks, which implement the technical infrastructure for manual activities within the business process. Another important type of states are Fork and Join states. These states are required to fork and join multiple parallel threads of execution inside one FSM instance. Some implementations provide nested states. A nested state encapsulates a complete "sub" FSM. These advanced features are mandatory for efficient commercial application. But, a detailed discussion of these features would lead to far for this paper, because they are not capable to modularize the concerns from the next section.

The rest of this section sketches a typical architecture for typical interpreters of these FSM based DSLs. The FSM interpreter normally acts as a kind of application server or is often deployed as a component in an application server. Several protocol gateways are installed in the application server. Each of the protocol gateways transforms events into invocation of popular communication protocols such as JMS, MQ Series, web services, IIOP, RMI and vice versa. Whenever a new event is created out of a "native protocol activity", the protocol gateway is responsible to forward this event to the FSM interpreter. The FSM interpreter forwards this event to an internal event dispatcher. This event dispatcher is responsible to identify the relevant FSM instance, and pass this instance over to the FSM interpreter, which finally processes the event and its caused transitions inside the FSM. This processing normally changes the state of the FSM and attribute values of the machine. Attributes and the state are normally stored transactional in an OODBMS, in a XML-database or in a relational database, which is often wrapped with an OR mapping tool such as Hibernate [12].

Figure 2 shows a typical architecture of such a system.

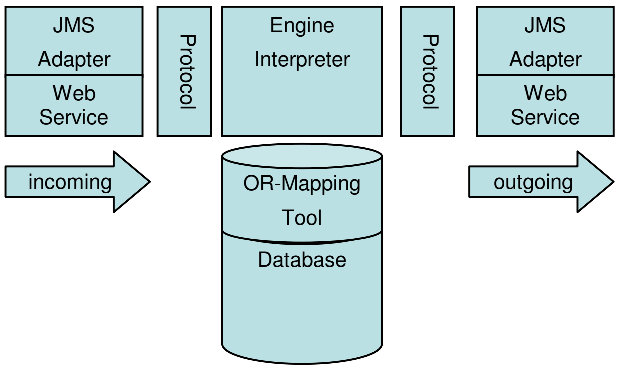

Figure 2

3. Typical Crosscutting Concerns in these Domains

The potential of crosscutting concerns to make even simple processes hard to understand can easily be shown on a simple but realistic sample. Assume, a bank wants to automate its credit approval processes. The reception of a valid approval request event puts the FSM in the state Request Stored. Whenever the Request Stored State is entered, a proposal evaluation process is started through a call to an external asynchronous service, e.g. through an event which is transformed through the infrastructure to a JMS message or to a Web Service call. This process may end with an acceptance, a refusal or manual investigation event, which indicates the need for manual evaluation. These events trigger a transition to the final Accept and Refused states or trigger a transition to the Manual Approval State. Any user decision in the Manual Approval State causes a transition to the Accept or Refused state. Figure 3 displays this process. Such models are often drawn and written by business analysts as a result of an initial design or requirement analysis.

[1] Please note: this difference is important in practical projects, but not for the content of this paper

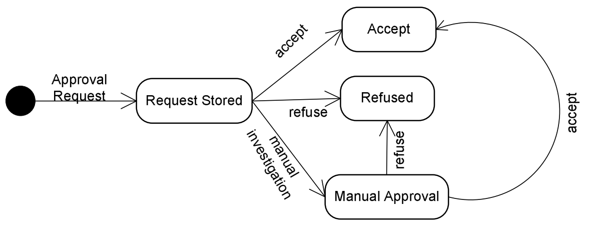

Figure 3

This process definition does not care about typical crosscutting concerns. Let's have a look at some typical crosscutting concerns and their implications to the process. Typical technical concerns:

- All communication protocols may cause technical errors. (Communication Failure)
- In case of technical errors duplicate transmissions of events may occur (Duplicate Transmissions)
- Typical domain specific concerns are soft real time requirements, e.g. the process must be performed in several minutes. (Timeouts)
- A typical process specific concern is the possibility for the customer to modify his approval request before it is approved. (Modification Proposal)

If the FSM receives a modified proposal event, then the concern Modification Proposal requires adding transitions from every state to the Request Stored State.

Timeouts require the definition of an additional timeout transition with some timeout activities from every state.

Duplicate Transmissions and Communication Failures require the definition of the process behaviour in case of technical errors. Such a scenario is solved in many times through the definition of an Error state. A process administrator has with this solution the possibility to restart the process, as soon as the error is fixed, which caused the transition to the error state. For each error scenario we have to add at least one transition from each state to the error state.

The FSM may receive unexpected events or outdated events because of each of these concerns. Detecting outdated events is tricky. It requires some support of messaging and data patterns such as Message Expiration, Correlation Identifier from [13] and Executional Context from [21]. Furthermore you have to guard each transition with a tangling invocation of a simple function, which performs the outdated or duplicate event detection based on the information in the event or its header. Finally, you have to add actions which maintain and add detection support information (e.g. time to life information [13], a transition count sequence number) into each event or its event header.

We started with a simple process which contained four states and six transitions and ended with a process consisting of five states and 17 transitions, each guarded by tangling code for outdated and duplicate event detection.

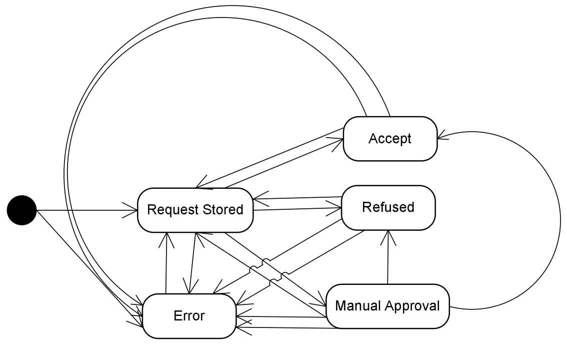

Figure4

Figure 4 shows the complete solution, Event annotation is hidden for a minimal graphical readability of the model. Handling such process models correctly just in the usual graphical DSLs is extremely difficult.

4. Composition Filter based FSM.

It is a natural idea for anybody, who is familiar with the concept of Composition Filter and has practical experiences with these DSLs [14], to define a composition filter based FSM to increase the modularity of the artefacts expressed in the DSL.

A Composition Filer based FSM works in the same way as the described FSM from the second section. Additionally the user has the possibility to define Composition Filters. Each filter can change or drop the event, forward it to the next filter, submit additional events, and change attributes and the state of the FSM. Each filter has complete access to the state and all attributes of the FSM as well as to all attributes to the event.

Each filter consists of two parts. The first part is responsible to identify, if the filter has to interact with the event, the other part is responsible to define what the filter does for the processing of this event. The first part works equal to a dynamically evaluated pointcut statement and is often named as When Clause, and the second part works like an around statement from the AOP platform AspectWerks [15] or the AOP language aspectj [1] and was often named as a then clause. It is obvious that a member of the AOSD community would have chosen different names. These names were taken from rule engines. Rule engines are other high level DSLs, which are quite popular and well known by the business process developers and their business analysts. These name choices simplified the understanding of the concept of Composition Filters for these important groups significantly.[2]

In most projects these filters have been implemented through the introduction of an interceptor framework between the transformation gateways and the FSM and between the FSM and the protocol transformation gateways for outgoing messages.

[2] It is also sensible to change the name of the concept Composition filter to the name "process rules" to make it easier to understand the concept and architecture in real life projects.

This interceptor framework is responsible to manage the filter, and forward the events to the filter. The interceptor framework is often based on a graphical rule management system, such as JRules [4]. Otherwise the interceptor framework is normally implemented by the application of the famous command pattern [3]. In this case, filters are implemented as simple Java or C# classes or in an Object Oriented script language like Ruby or Python. The developer simply has to subclass an abstract filterclass and provide the implementation of the when and then clauses as a template method [3]. Figure 5 provides the Composition Filter based FSM architecture. Even most closed source FSM implementations[3] provide sufficient public hooks for this architectural enhancement.

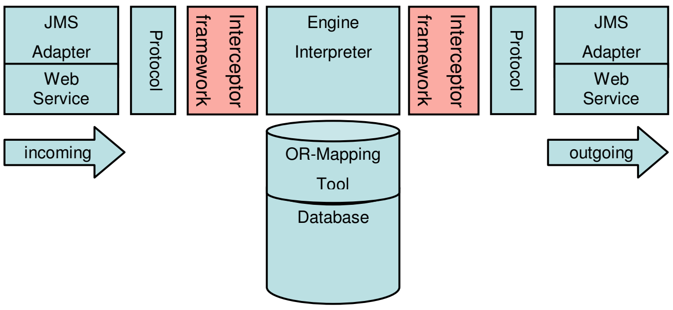

Figure 5

5. Example Resolved

This section outlines how the concerns could be implemented with composition filters and simple modifications from the initial FSM model, which was shown in Figure 3.

5.1 Concern Modification Proposal

This concern could be simply implemented in one composition filter. The When Clause just has to check, if the event is from the type modified proposal event. The then clause must simply change the state of the engine to the proposal created state.

5.2 Timeout concern

This concern could be implemented in just one simple filter for simple business requirements for handling timeouts. E.g if it is sufficient to proceed to the Refused or Accept State. In these cases the When Clause must only check, if the event is from the type "timeout". The then clause must simple change the state of the engine to the Refused or Accept State.

However, even very complex business requirements for this concern can be implemented with composition filter. Even quite complex logic such as retry twice, send each time an escalation email to the manager in duty, and if that did not work, refuse the proposal could be implemented in just one composition filter and an additional value attribute in the FSM, which contains the number of performed retries. The implementation of the then clause could look like:

```
If (fsm_instance.steps<2):
    sendEmail(getManagerInDuty())
    drop(event)
    increment(fsm_instance.steps)
else:
    fsm_instance.setState(PROPOSAL_REFUSED)
```

[3] This is at least true for all implementations, which are known by the author.

So this concern could be modularized in exactly one artefact. This enables an easy modification and enhancement of the solution, because the concern is fully modularized and orthogonal to the business code.

5.3 Concern Handling of Communication Failures

A typical implementation of this concern consists of one additional state in the FSM model and an Error handling composition filter. The error handling composition filter has a when clause, which selects all events. The implementation of the then clause depends on the business domain and some of its restrictions. However there exists a set of standard policies, which are normally applied. One common policy is a retry policy, A typical implementations of such a policy is the retry once policy. Its implementation in case of a failure could look like:

- use some repairing and auditing activities, (e.g. do a rollback, and retry, log the exception messages, causes, events, notify an administrator etc.
- and finally if none of the repairing attempts has succeeded, it might change the state of the process to the error State.

5.4 Concern Handling of Outdated Messages

This concern does not appear normally as a direct concern from the business drivers of the project. The business drivers normally assume that outdated messages can not occur. However they occur in practice caused either by business related concerns such as time outs and modification proposal or technical related concerns such as Communication failure.[4] Most non trivial FSMs suffer therefore easily from outdated events,.

Composition filters can not be used to apply the architectural data structures patterns. But, all the guarding activity and all activity related to maintain and add detection support information can be easily solved with two composition filter. Both filters must share some process specific values, e.g. a transition count sequence number. These values should be added to the process attribute set.

The first filter is only responsible for adding the information into outgoing events. It has a when clause which selects all outgoing events, and a then clause which adds decision support attributes such as the transition sequence number to the context of outgoing events.

[4] In real projects this concern is not recognised as a complete independent concern and often ignored, because there are several marketing myth, which create the illusion that these scenarios can be neglected. I reviewed several projects, where developers fixed more than 150 independent bug reports, till such a business process was reliable enough to be released in production.

The second filter is responsible for updating the decision support attributes and dropping old attributes. It has typically a where clause, which implements an algorithm such as:

```
If (isOutdated(
            event.context.transition_nr):
      drop(event)
      return
proceed(event);
Increment_suportinformation(fsm_instance)
```

5.5 Comparison with the solution without Composition Filter

Table 1 shows the different numbers of the modelling elements. The lowest numbers has naturally the model from the business analyst, as this is just a design model and far away from an executable model. The traditional solution with the four simple concerns (in practical projects there are several more to be handled) has a dramatic increase in the number of transitions and activities

.

	Without additional concerns	Traditional implementation with concerns	With composition filers
Number of states	4	5	5
Number of transitions	6	17	7
Number of activities	~6	~35	~7
Number of guarding statements	0	17	0
Number of Composition Filter	0	0	5

Table 1

The solution with the composition filter has only a slight increment in the number of transitions, activities and state. As a trade off this solution adds five simple composition filters. Three of the filters can be reused in lots of different projects, nearly in all FSMs of a given organisation. The filter for the timeout handling might be derived (e.g. parameterized) from a library of filters for company policies.

Furthermore we have now the possibility to track a concern to the major module, which is handling the concern. This is a great additional benefit for practical process improvement projects, which have strong legal audit requirements. We have another big plus in the fact that all new concern can be implemented in a modular way, enabling a fast improvement when necessary. Even our minimal realisation of the timeout concern could be dramatically improved by some lines of code, (e.g. by basing the transition on a simple statistical estimate) than the directly modelled solution, where only one transition is normally modelled.

However additional typical additional requirements can be realised with minimal effort, as example Aris PPM [17] monitoring could be added by just one library filter, which could even be reused between different projects and processes.

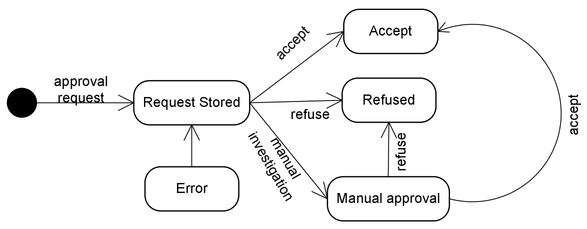

Figure 6

Figure 6 shows the modelled process. This process looks pretty close to the original process without any of the concerns, as a simple comparison between Figure 6 and Figure 5 suggests. It did not explode in graphical complexity like the traditional solution. This enables business analysts and sometimes even business owners (e.g. Line of business manager, process manager) to reason over the process implementations in production and identify improvement potentials. This is a clear enabler for agile business process management and Domain Driven Design [16].

6. Summary, Conclusion and future Work

This paper shows a practically proven approach how composition filter can be used to enhance FSMs dialects, which are very often used as DSLs from the shelf for Business Process Management, SOA orchestrations or application integrations process models. The samples demonstrate:

- that this approach increases the modularity of typical domain specific concerns
- that the AOP enhancement of the DSL simplify the implementation of concerns important to the business and domain specialists.

In addition this paper demonstrate, that AO based domain specific languages are not only relevant for classical technical problems such as synchronisation, they are also relevant for business oriented DSLs.

The experiences in the actual projects, where this approach was used, demonstrated great improvements against similar projects, which used only plain FSMs. Also there have been significant hints, that there exists a complete AOSD tool and methodology stack ranging from process analysis and design over service implementation to improved infrastructure adaptation. The potential to adapt and use existing infrastructure such as CORBA and J2EE more efficiently with AOP languages such as AspectJ has been demonstrated in [18]. Improvements in the effort to craft services out of exisiting java applications have been shown in [19]. It is now one of my major interests to generalize and formalize the improvements through the application of Aspect oriented techniques in the analysis and design activities of business process management projects.

7. Literature

[1] AspectJ Team, *The AspectJ Programming Guide*, http://www.eclipse.org/aspectj , December 2006

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

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

[4] ILOG, *Rules*, http://www.ilog.com/products/rules , January 2006

[5] Vitria, *BusinessWare*, http://www.vitria.com/products/platform, October 2005

[6] JBoss, *JBoss jBPM*, http://labs.jboss.com/portal/jbossjbpm December 2006

[7] Tibco, *Business Process Management Software*, http://www.tibco.com/software/business_process_management/ December 2006

[8] IBM, *Websphere MQ Workflow*, http://www-306.ibm.com/software/integration/wmqwf/ December 2006

[9] IDS Scheer, *Aris Toolset*, http://www.ids-scheer.com, January 2006

[10] OMG, *Unified Modeling Language*, http://www.uml.org/, January 2006

[11] Active endpoints, *ActiveBPEL Engine Overview*, http://www.active-endpoints.com/active-bpel-engine-overview.htm January 2006

[12] JBoss, *Hibernate*, http://www.hibernate.org/ January 2006

[13] Gregor Hophe, Bobby Woolf, *Enterprise Integration Patterns*, Addison Wesley, 2004

[14] University of Twente, *Composition filter research homepage* http://trese.cs.utwente.nl/oldhtml/composition_filters/ January 2006

[15] AspectWerkz, *AspectWerkz*, , http://aspectwerkz.codehaus.org/ January 2006

[16] Eric Evans, *Domain Driven Design*, Addison Wesley, 2003

[17] IDS Scheer, *Aris Process Performance Manager*, http://www.ids-scheer.com/germany/products/aris_controlling_platform/49532 January 2006

[18] Schmidmeier, A: *Using AspectJ in Component-Based Architectures on the Server Side*, Invited talk at AOSD'02, Enschede, April, 2002

[19] Schmidmeier, A.: *Using AspectJ to Eliminate Tangling Code in EAI Activities*, practitioners report at AOSD´03, Boston, 2003

[20] IBM, et.al, *Business Process Execution language for Web Services*, http://www-128.ibm.com/developerworks/library/specification/ws-bpel/ January 2006

[21] Allan Kelly, *The encapsulate Context Pattern*, Overload 63, Oktober 2004