BPEL PATTERNS FOR IMPLEMENTING VARIATIONS IN SOA APPLICATIONS

Samia Oussena, Dan Sparks and Balbir Barn
Computing Department, Thames Valley University, Slough, Berkshire, UK

Keywords: BPEL, business process, patterns, Service-Oriented Architecture, process variation, reference model.

Abstract: The main purpose of the COVARM research project is to define a candidate reference model utilizing a framework of web services to support a key UK Higher Education business process. Any given business domain may offer a level of complexity such that process activities, terminology (the ontology) and business rules may vary between organizations belonging to same domain While a generic process can and has been built as part of the reference model, the flexibility (or variability) is afforded by the implementation strategy for the canonical model / generic process. We have implemented the following variations: activity ordering, cross-site terminology harmonization, and specific business rules to address the variability requirements. This paper presents our experience with explicitly managing the variability within the implementation technology. With the use of BPEL patterns, we describe how the management of these variations can be dealt with in an SOA application implementation.

1 INTRODUCTION

Reference models are used as an effort to define common terms, such as; a well defined framework for extending aspects of the specification; an attempt to define a general, overarching structure of the domain, and; a focus on interoperability and standardisation. Thus the reference model will provide a strong steer on how systems for a particular domain should be implemented. The COVARM project aims to define a candidate reference model utilising a framework of web services to support a canonical business process to support course validation within UK Higher Education institutions.

Service Oriented Architecture (SOA) is a strategy currently being pursued by the Joint Information Systems Committee (JISC), the sponsors of the COVARM project [Olivier B. et al, 2006]. The project therefore followed an SOA approach.

The trend of SOA development is to maximise the separation of concerns in software development. The idea of separating an interface from its implementation to create a software service definition has been well proven in J2EE, CORBA and COM (Emmerich 2000). Web Services provide the ability to more cleanly and completely separate a service description from its execution environment. The real value of this separation comes when these web services are deployed in the context of an SOA, making it easier to develop new applications by orchestrating web services. The main benefits provided by adopting an SOA can be summarised as follows (Newcomer et al 2005):

- **Reuse:** the ability to create services that are reusable in multiple applications
- **Efficiency:** the ability to quickly and easily create new services and new applications using a combination of new and old services
- **Loose technology coupling:** the ability to compose applications independently of the execution environment of the services

SOA applications should therefore provide flexibility and adaptability to respond to changes to requirements. Implementation of these changes should require localised changes to either specific services or to the orchestration of these services through a Business Process Execution Language (BPEL) process (Juric et al, 2006). The focus on flexibility and adaptability should mean that adapting an application to a new domain will be much faster and cheaper than a ground-up approach. Changes such as these will still require developer intervention with appropriate expertise in either the services implementation or the BPEL process. Minimising this developer intervention is consistent with the concepts of SOA. To do so requires that we deal explicitly with the variations within the implementation, should be externalised and parameterised by an interface to remove developer intervention.

In this paper we present a set of BPEL patterns that record our experience of dealing with three types of variations. The variations are not specific to our project domain; they are general variations that could occur in any SOA development.

2 BACKGROUND

The objective of COVARM is to develop a reference model for H.E. course validation. Course validation can include the specification of a new course at various levels, e.g. undergraduate and postgraduate level. Course specifications address areas such as rationale, appropriateness, resources required for the course, assessment strategy and so on. The specification is determined by local institutional constraints but there are other requirements imposed by national bodies. Therefore even though the process may be implemented differently in the different institutions, the constraints imposed by the national bodies such as the Quality Assurance Agency (QAA) provide some standardisation for the process and its outputs. The approach that we took was to build process models of four institutions and then derive a synthesised process including a canonical process and a set of options and extensions that would be required to enable customisation for any HEI (Barn et al 2006). Our approach was to create a synthesised model to represent an aggregation of concepts rather than a homogenised, re-engineered process.

3 RELATED WORK

Customization has often been dealt with in the implementation phase by providing a developer with abstractions. Abstraction hides and emphasises characteristics of a subject in ways that are relevant to a particular usage or purpose (Parnas 1976). There are two ways of providing the bridge between the abstract model and the implementation [Johnson 2000]. One way is by providing a software framework that specifically addresses a well defined, narrow problem domain, and using the abstractions in a model to define how requirements for variability points in the framework must be met (Greenfield 2004). This has worked specifically well in graphical user interface development; Java Faces (Bergsten 2004), for example, effectively employ abstraction in the domain of graphical user interface manipulation. Alternatively, patterns can be used to implement the abstractions. This becomes a very powerful technique when a collection of patterns are combined and implemented in a toolset, providing developers a way of applying patterns to solve similar kinds of problems (Johnson 2005).

Generative programming has emerged as one way of improving software productivity (Csarnecki et al 2000). Instead of building a single software system as a solution to a certain problem, generative programming involves designing a system from which configurable solutions can be generated based on an assembly plan or a configuration specification. Although a wealth of work has been undertaken in this area (Van Zyl 2002), these methods tend to focus on the early stages of the software life cycle and address product line issues at a high-level of abstraction. Connecting product-line concepts with established implementation technologies is thus largely left to the user, for example, in Feature–Oriented Domain Analysis (FODA) (Kang et al 2002), where mandatory and variant requirements are depicted in a graphical form as trees. We can therefore find out which variants have been anticipated during the domain analysis, but there are no guidelines for how the domain analysis with the variant requirements can be realised. To address this, two main approaches have been followed; either module replacement or data controlled variation.

All these approaches still require developer intervention at some point, whether at a relatively high level, such as replacing one service with another, or at a lower level, such as modification, recompilation and redeployment of source code.

Our approach leverages the data controlled approach, but is applied to the BPEL orchestration level of the architecture. The variations have been externalised and parameterised by an interface to remove developer intervention. Abstraction has been applied at the BPEL level; we have formalised our implementation of these variations in three design patterns.

4 IMPLEMENTING PRODUCT VARIATIONS WITHIN BPEL PROCESSES

The rest of the paper presents our experience with explicitly managing variability within an SOA application implementation. In our case the variations that we identified and addressed can be summarised in the following three categories:

- Terminology variations: These include the domain concept name. For example, some institutions refer to an organisation structure unit as a school where others refer to it as a department. Other variations refer to the format of the documentation that exists within the process.

- Activity ordering variations: The variation may refer to the order of the activity execution, in some cases the activity may need some specialisation, or is even not required.
- Business rule variations; for example a course validation panel structure varies from one institution to another. Some institutions will require at least two external panel members whereas for others one is sufficient.

5 IMPLEMENTING THE VARIATIONS WITH BPEL DESIGN PATTERNS

In this section we are going to look at each type of variation that we had to deal with and discuss the implementation that we adopted.

Each pattern has been implemented with the following:

- BPEL WSDL definitions which allow the variation parameters to be passed to the process at runtime;
- A JSP page which passes an XML document created from either user-specified values, or a selection of pre-written XML documents, to process, thereby executing the process with the variations specified in the XML document, and;
- A BPEL process which accepts the XML document as an input parameter, and applies each of the three design patterns in turn.

More information and sample code for these patterns can be found at http://covarm.tvu.ac.uk/BPELVariations.html

Software design patterns offer flexible solutions to common software development problems (Gamma et al. 1995). Each pattern is comprised of a number of parts, including purpose/intent, applicability, structure, and sample implementations.

A design pattern is a general repeatable solution to a commonly occurring problem in domain; it is a description or template for a solution to a problem. Our format for describing a pattern is:

- Pattern name
- Intent
- Motivation
- Applicability
- Implementations
- Sample code
- Related patterns

5.1 Terminology Variation Pattern

Pattern Name and Classification: Terminology Variation Pattern.

Intent: To provide a flexible way to implement terminology customisation.

- **Motivation:** Where two or more organisations use the same process, there are likely to be variations in the terminology used. For example, one organisation may refer to a 'course', another to a 'programme', and yet another to a 'syllabus', yet the process will need to:
- deal with all these references as if they are the same thing;
- return information using the correct terminology;
- where appropriate, use the specified terminology to retrieve the correct information

Implementation: This can be achieved by the process using a meaningfully named variable that can hold the terminology variant. For each attribute in the domain that can be predicted to change, i.e. is likely to be a terminology variant, a BPEL variable should be used. These variables are initialised at process runtime.

Applicability: This pattern can be applied wherever there may be variations in terminology for a 'shared meaning' term within a BPEL process.

Sample code: In the course validation example, such a variable could be called 'LEARNING_PRODUCT_STRING' and may refer to 'course' in one organization , 'programme' in another organization , or 'syllabus' in yet another . Considering this, if we wished to display information about an educational institution, the BPEL code that concatenates the information string would look like this:

```
   concat(bpws:getVariableData('INSTITUTION_
STRING'), ' is a ',
bpws:getVariableData('INSTITUTION_TYPE_STRIN
G'), '. Its organisational unit is called a
',
bpws:getVariableData('ORGANISATIONAL_UNIT_ST
RING'), ', and its learning product is
called a ',
bpws:getVariableData('LEARNING_PRODUCT_STRIN
G'), '. ')
```

The same process, using different terminology variants would then produce a different, and relevant, string for each institution:

```
   "TVU is a University. Its organisational
unit is called a Faculty, and its learning
product is called a Module."
   "CCC is a College. Its organisational
unit is called a Department, and its
learning product is called a Course."
```

It is therefore also possible to create terminology-specific input to calls to external services, for example as a search string to retrieve a course/programme/module name from a document. The process simply needs to assign the value of the variable to the service's input variable:

```
    <assign
name="setLearningProductNameString">
        <copy>
          <from
variable="LEARNING_PRODUCT_NAME_STRING"/>
          <to
variable="getLearningProductNameInput"/>
        </copy>
      </assign>
```

This has the effect that even where different institutions have documents where the learning product name may be tagged as `<programme-name/>`, `<course-title/>`, or `<module-name/>`, the process can still retrieve the correct field within the document with no customisation necessary.

Consider an example where a generic 'document' data type is being used. The document contains 'sections' and each 'section' has a 'section name'. If we wanted to retrieve data from a particular section, such as the course/programme/module name, the process would be able to apply this pattern to select the appropriately named section from the document, regardless of the terminology used by the invoking institution.

The process would simply need to say "retrieve the value stored in the section called 'LEARNING_PRODUCT_NAME_STRING'", and, because the correct value for that string has been supplied at runtime, it will be possible to retrieve the correct data without any process customization at all.

Related patterns: Adaptor pattern, bridge pattern

5.2 Activity Order Variations Pattern

Pattern Name and Classification: Activity Order Variations.

Motivation: During our analysis of the different institutions' processes, we found that while there were certain 'core' tasks, the order that these tasks occurred in varied, as well as, in some cases, certain tasks not being invoked at all by some institutions.

Implementation: For the following example we assume that there are 3 tasks, Task A, Task B, and Task C. These tasks may be performed in any order.

This pattern involves a while loop which will loop once for each task that can occur. Each task is executed when the condition in the switch case is met for that task. The condition is simply whether the while loop's index is equal to the task order specified by each institution. Each institution can specify the task order by passing integer values (in this case 1-3) for each task to occur in, 1 being the earliest, and 3 being the latest. -1 signifies a task that does not occur.

In our demonstration example, as each task is executed, it simply appends '`…Doing task X`' to a string. If a task is not done '`…1 Task Not Done`' is appended.

The task order is specified in variables which are initialised at the start of the process' execution. In our demonstration, the variables are simply `Task_A_Order`, `Task_B_Order`, and `Task_C_Order`.

Altering the values of these variables provides variations in task order with no change necessary to the process:

```
"...doing Task B ...doing Task A ...1
Task Not Done"
 "...doing Task C ...doing Task A ...
doing Task B"
```

This pattern demonstrates that it is possible to implement variations in task order with no process customisation. It is even possible for an institution to avoid completing any of these tasks if it is required, by setting the order of each to -1.

Applicability: This pattern can be applied to any set of activities within a process that may run in a different order, as required by users of the process. The granularity of the pattern can be altered to apply to sub-processes (themselves entire individual processes).

Of course, it is possible to wrap instances of this pattern within other instances; all that is required at the BPEL design / development stage is to move each scope / task into a switch case. If we had wanted to vary the order in which our 'terminology', 'processOrder' and 'businessRules' sections of our process had run in, we could have moved each scope into its own switch case.

5.3 Business Rules Variation Pattern

Pattern Name and Classification: Business Rules Variation

Intent: Isolate business policies from the rest of the process implementation

Motivation: Our analysis of different institutions showed that business rules and policies can vary both within the same process, and in relation to specific activities within the process.

Implementation: We suggest that business rules variation can be supported by the use of a defined rule schema within the process WSDL. The process is written to manipulate the data type, rather than a specific instance of it, therefore allowing for variations in element attributes.

Let's look at an example from our domain; a course validation process which includes a sub process for managing a validation event. Here, a course is formally reviewed by a panel. The rule governing the panel composition is different from one institution to another. For most institutions a panel is composed of a chair, and a number of internal and external panel members. However, the number of external attendees can vary depending on the type of event, and also may vary per institution.

Applicability: This pattern can be applied to any business rule or policy which can be expressed in schema form; provided this is possible the pattern will provide a higher level of abstraction.

Sample code: We considered the variations posed by different institutions' policies regarding panel composition for validation event meetings. These variations concerned the required roles, and minimum and maximum panel members for each of these roles. In addition, role names vary across institutions; the head of a meeting might be called a 'chair', a 'chairperson', or 'meeting head'. Using only terminology variations would not help here, because it is not possible to know in advance how many different roles may be required for a particular institutions panel.

In this case we defined a data type that could express a panel's constitution:

```
   <element name="panelRole">
      <complexType>
        <sequence>
          <element name="roleName"
type="string"/>
          <element name="min" type="int"/>
          <element name="max" type="int"/>
        </sequence>
      </complexType>
    </element>
    <element name="panelComposition">
      <complexType>
        <sequence>
          <element ref="client:panelRole"
maxOccurs="unbounded"/>
        </sequence>
      </complexType>
    </element>
```

Put simply, a 'panelConstitution' contains any number of 'panelRoles', which each contain the name of the role, and the minimum and maximum members for that role.

The process is written to manipulate the data type, rather than a specific instance of it, therefore allowing for institutional variations in role names, numbers, and so on.

To implement this pattern, we simply constructed a string description of the panel constitution as passed to our process. A while loop loops once for each panelRole specified in the panelConstitution, and appends the data from that to the description. The variations produced completely different panel constitutions with no customisation of the process necessary:

```
   "The panel is composed of: A minimum of 1
and a maximum of 1 Chair, A minimum of 3 and
a maximum of 6 Internal, A minimum of 2 and
a maximum of 4 External"
   "The panel is composed of: A minimum of 1
and a maximum of 2 Chairperson, A minimum of
2 and a maximum of 3 Department Head, A
minimum of 2 and a maximum of 4 Department
Administrator, A minimum of 1 and a maximum
of 2 Department Lecturer"
```

While this is a relatively simple example, it should be clear that this approach can be applied to more complex operations, such as guiding a user to select appropriate panel members based on each role and minimum / maximum requirements, with variations appropriate to any institution, without requiring any customisation of the process at all.

6 REFLECTIONS

In this paper we have described how the need for developer interaction to facilitate application customisation can be a hindrance to SOA ideals. Process and service customisation and replacement all require intervention from developers with specific and specialised skills. Such application customisation can also require code re-compilation and redeployment. This can lead to delays in implementing change, and also to version management issues, where there can be any number of variant processes and services in operation.

To address this, we have implemented three design patterns to be applied at the BPEL process level of an SOA application. The patterns adhere to the concepts of abstraction, flexibility, and responsiveness to change with the minimum of developer intervention.

The Terminology Variation pattern provides a mechanism whereby a process can deal with variations in terminology where a shared meaning is implicit. The pattern is effective provided the meaning of the shared term is effectively communicated to any invoking partners. The ease of use of this pattern may also be a drawback, in that the possibility of customising each and every term in a process could lead to a huge number of variables which would need to be passed and configured at runtime.

The Activity Order Variations pattern provides a way for invoking institutions to customise the order tasks are invoked within a process. The pattern can be applied almost anywhere within a process, and even as a wrapper to itself. Areas where this pattern is not applicable at present are where certain tasks may run in parallel, however; the pattern only makes it possible to run one task per loop. Given though that this pattern makes it possible to execute varying sub-processes by specification at run time, it is a powerful pattern for implementing variations to a process without developer intervention. Problems may arise as the number of activities increases; it would be helpful if this pattern could eventually be integrated into a BPEL IDE in a similar manner to the FlowN construct, which allows for the parallel execution of a number of tasks specified at runtime.

The Business Rules Variations pattern allows variations in policy to be implemented by one process, thus increasing reuse and flexibility; as well as different organisations being able to use the same process, should one organisation change its rules, it will be possible to do so without requiring modification of the process. The rule definitions need to be created carefully, however, in order to take the level of abstraction to a high enough level to make it usable by different parties, while still retaining enough relevance to provide value.

These patterns are not independent of each other either; indeed it could be argued that the Business Rule Variations pattern employs the Terminology Variations pattern, such as in our panel constitution example, where ‘role-name’ is the generic term which is specialised by the invoking institution at runtime.

7 CONCLUSIONS

BPEL and SOA provide ways in which product variation can be implemented far more efficiently than the alternative of a ground-up solution. There are, however, still bottlenecks and impediments to the vision of truly flexible applications. It is our belief that the patterns presented in this paper could point in the direction of solutions which could remove some of these barriers.

We hope that these patterns can be examined, and improved, to provide future developers with the tools to build genuinely flexible and customisable applications.

ACKNOWLEDGEMENTS

This work has been funded in part by the Joint Information Systems Committee (JISC). http://www.jisc.ac.uk

REFERENCES

Barn, B., Dexter H., Oussena, S. and Sparks, D. 2006, SOA-MDK: Towards a Method Development Kit for Service Oriented System Development, Proceedings of the 15TH International Conference on Information Systems Development: Methods and Tools, Theory and Practice, Budapest, Hungary

Barn, B., Dexter, H., Oussena, S. Petch, J., 2006, An Approach to Creating Reference Models for SOA from Multiple Processes. In: IADIS Conference on Applied Computing, Spain.

Bergstein, H., 2004, JavaServer Faces, O’Reilly.

Csarnecki, K., Eisenecker, U., 2000, Generative Programming: Methods, Tools and Applications, Addison-Wesley.

Emmerich, W., 2000, Software Engineering and Middleware: A Roadmap. In: The Future of Software Engineering, ACM Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995, Design patterns: elements of reusable object-oriented software, Addison-Wesley.

Greenfield, J., and Short, K., 2004, Software Factories: Assembling applications with patterns, Frameworks, Models and Tools, John Wiley and Sons.

Johnson, R., 2005. J2EE Development Frameworks, Computer, Vol. 38.

Johnson, R. 2000, Documenting Framework using Patterns, ACM SIGPLAN notices, Vol. 27, Number 10.

Juric, M. et al, 2006, Business Process Execution Language for Web Services 2nd Edition

Kang, K.C., Lee, J., Danohoe, P., 2002, Feature-oriented Product Line Engineering, IEEE Software, Vol. 19, Number 4

Newcomer, E., Lomow, G., 2005, Understanding Web Services with SOA, Addison Wesley Professional

Olivier B., Roberts T., and Blinco K. "The e-Framework for Education and Research: An Overview". DEST (Australia), JISC-CETIS (UK), www.e-framework.org, accessed December 2006.

Parnas, D., 1976, On the Design and Development Families. IEEE Transaction on Software Engineering, March 1976.

Van Zyl, J.A., 2002, Product Line Architecture and the Separation of Concerns, Second Software Product Line Conference – SPLC 2, San Diego, Kluver Publication.