Customizing Aspect-Oriented Variabilities using Generative Techniques

Uirá Kulesza[1] Carlos Lucena[1] Paulo Alencar[2] Alessandro Garcia[3]

[1]*Software Engineering Laboratory – Computer Science Department – PUC-Rio – Brazil*
{uira, lucena}@inf.puc-rio.br

[2]*Computer Systems Group – Computer Science Department – University of Waterloo – Canada*
palencar@csg.uwaterloo.ca

[3]*Computing Department – InfoLab 21 – Lancaster University – UK*
garciaa@comp.lancs.ac.uk

Abstract

With the emergence of aspect-oriented (AO) techniques, crosscutting concerns can be now explicitly modularized and exposed as additional variabilities in program families. Hence, the development of highly customizable software family architectures requires the explicit handling of crosscutting variabilities through domain engineering and application engineering steps. In this context, this paper presents a generative model that addresses the implementation and instantiation of variabilities encountered in AO software family architectures. The use of our model allows for an early specification and preparation of AO variabilities, which in turn can be explicitly customized by means of domain engineering activities. All the variabilities of the architecture are modeled using feature models. In application engineering, developers can request an instance of the AO architecture in a process of two stages: (i) the definition of a feature model instance which specifies the resolution of variabilities for the AO family architecture; and (ii) the definition of a set of crosscutting relationships between features.

1. Introduction

Aspect-oriented (AO) techniques have been proposed as an approach which aims to separate and modularize crosscutting concerns [7, 8]. Crosscutting concerns are concerns that often crosscut several modules in a software system. It encourages modular descriptions of complex software by providing support for cleanly separating the basic system functionality from its crosscutting concerns. Hence, AO techniques can be used now to exploit variabilities relative to crosscutting concerns, thereby enhancing the reusability and customizability of software family architectures.

Aspect-oriented programming (AOP) [8] proposes the *aspect* abstraction and new composition mechanisms which allow for the implementation of crosscutting variabilities. Aspects are modular units that extend the functionalities of classes in well-defined execution points, the so-called *join points*. AspectJ [1] is the most popular programming language that enriches the Java language with AO extensions. It provides constructions to specify more reusable and variable aspects, such as, abstract aspects and abstract *pointcuts*.

Recent work [2, 9-13] has focused on the use of AO techniques to enable the implementation of flexible and customizable software family architectures. In these research works, aspects are exclusively used to modularize crosscutting variable (optional or alternative) features encountered in the programming of frameworks or software product lines. However, there is a lack of support for software family architects and application developers to respectively manage and instantiate AO variabilities in different development stages. It is important to define models which allows to handle not only orthogonal variabilities enabled by classical OO mechanisms (such as, OO framework hotspots), but also the new ways of variabilities supported by AOP.

In this context, this paper presents a generative model [5] which provides explicit means to instantiate aspect-oriented family architectures. Using our model, AO variabilities are prepared to be customized through domain engineering activities. All the variabilities of the architecture are modeled using feature models. In application engineering, developers can request an instance of the AO architecture in a process of two stages: (i) the definition of a feature model instance which specifies the resolution of variabilities for the AO family architecture; and (ii) the definition of a set of crosscutting relationships between features. These crosscutting relationships are used to customize

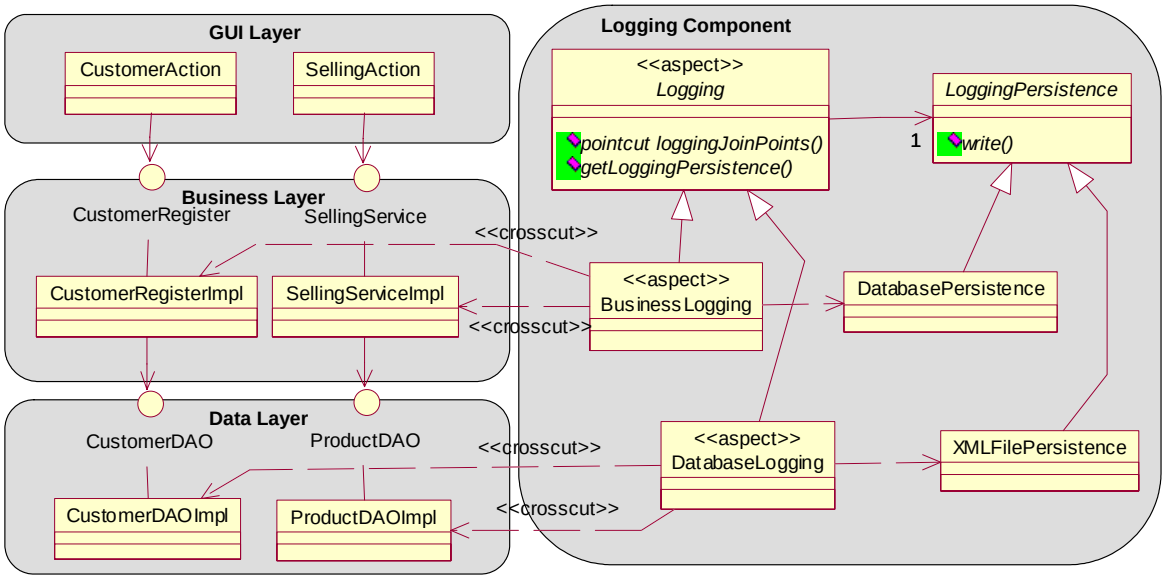

Figure 1. Logging Aspects

abstract pointcuts which define how aspects can affect system classes.

The remainder of this paper is organized as follows. Section 2 presents an overview of the variability mechanisms supported by AOP. Section 3 describes our generative model. Section 4 exemplifies our model application to the customization of logging aspects. It describes both domain implementation and application engineering activities. Section 5 presents our conclusion and points to directions for future work.

2. Variabilities in AOP

AO programming languages offer constructions to specify a set of join points. The join point model defined in such languages allows the aspect implementation to extend the class functionalities in specific points. Each aspect specifies a set of pointcuts and advices to implement these extensions. Pointcuts have a name and are collections of join points. Advices are a special method-like construction of aspects which are used to attach new crosscutting behaviors along the aspect pointcuts. These mechanisms can address the implementation of optional and alternative crosscutting features encountered in the implementation of software architectures.

As discussed in Section 1, AO languages, such as AspectJ [1], also support the definition of abstract aspects which can contain both abstract pointcuts and methods. These constructions enable to postpone the implementations of pointcuts and methods to concrete subaspects. Each subaspect can customize these elements considering a particular implementation of interest. Thus, more reusable aspects can be specified using these mechanisms.

Figure 1 presents an example of a `Logging` abstract aspect. It defines the `loggingJoinPoints()` abstract pointcut and the `getLoggingPersistence()` abstract method. The former is used by the subaspects to specify the join points in an application which will be logged. The latter allows subaspects to define a specific persistence mechanism to accomplish the logging. Figure 1 also shows two subaspects which implement in different ways the logging for a web application. The `BusinessLogging` aspect defines that the logging of the business services will be realized in the database. The `DatabaseLogging` aspect specifies the logging of the database accesses in a XML file. Figure 2 shows the AspectJ source code of the `Logging` abstract aspect and the `BusinessLogging` subaspect.

```
public abstract aspect Logging  {
    private LoggingPersistence log =
                getPersistenceLogging();

    abstract pointcut loggingJoinPoints();
    after(): loggingJoinPoints() {
       this.log.write(thisJoinPoint);
    }
    public abstract LoggingPersistence
                        getLoggingPersistence();
}
public aspect BusinessLogging extends Logging {
    pointcut loggingJoinPoints():
       execution(CustomerService+.*) ||
       execution(SellingService+.*);

    public LoggingPersistence
                    getLoggingPersistence(){
      return new DatabasePersistence();
    }
}
```

Figure 2. Source Code of Logging Aspects

The subaspects presented in Figures 1 and 2 were manually codified to customize the Logging feature for a specific web system. Different crosscutting features could also be implemented as abstract aspects and be reused in the implementation of subaspects. An automated mechanism could support the generation of these subaspects. It requires not only configuring existing functional variabilities (such as, the alternative persistence in the logging example), but also generating specific pointcuts in the subaspects. In this paper, we propose an aspect-oriented generative model which aims to generate these subaspects, including the customization of their pointcuts.

3. Approach Overview

Our approach is centered on the concepts of generative programming. Generative Programming (GP) [5] addresses the study and definition of methods and tools that enable the automatic generation of software from a given high-level specification language. It has been proposed as an approach based on domain engineering [5]. GP promotes the separation of problem and solution spaces, giving flexibility to evolve both independently. To provide this separation, GP proposes the concept of a generative domain model.

A generative domain model is composed of three basic elements: (i) problem space – which represents the concepts and features existent in a specific domain; (ii) solution space – which consists of the software architecture and components used to build members of a software family; and (iii) configuration knowledge – which defines how specific feature combinations in the problem space are mapped to a set of software components in the solution space. Two new activities need to be introduced to domain engineering methods in order to address the goals of GP:

- development of proper means to specify specific members of the software family. Domain-specific languages (DSLs) must be developed to deal with this requirement;
- modeling of the configuration knowledge in detail in order to automate it by means of a code generator.

We have defined an AO generative model following the model presented by Czarnecki and Eisenecker [5]. However, we propose the extension of that generative model to support the instantiation and customization of AO architectures. It allows configuring and generating specific crosscutting and non-crosscutting variabilities. Our generative model is composed by the following elements:

(I) *a feature model* – this model works as a configuration domain-specific language (DSL) responsible to specify and collect the features to be instantiated in the software family architecture. It is used to collect information to configure both the crosscutting and non-crosscutting variabilities. A set of crosscutting relationships between features is used to help the customization of aspects pointcuts.

(II) *an AO architecture* – it defines the main components of a software family architecture. This architecture defines a set of variabilities which need be customized to define a complete application. Crosscutting variabilities are implemented as aspects in this architecture. Each component of the architecture is specified as a set of classes, aspects and templates. The latter ones define elements that will be customized during the instantiation of the architecture. We also provide guidelines to implement these AO architectures by means of a base OO framework and a set of aspects which define optional and alternative crosscutting features existing in the OO framework. More details on these guidelines are provided in [9];

(III) *a configuration model* – it specifies the mapping between the features existing in the crosscutting feature model and the components (or their respective sub-elements, such as, class, aspect or templates) of the AO architecture. The configuration model is used to support the decision of which components must be instantiated and what customizations must be realized in those components considering a specific application.

There are several activities involved in the process of development of the elements of our generative approach. These activities are organized under the perspectives of domain implementation and application engineering. Next section details them in the context of the Logging example presented in Section 2.

4. Customizing AOP Variabilities: a Working Example

In this section, we illustrate our generative model by showing the customization of an AO architecture. We explore the customization of the Logging subaspects presented in Section 2. This is an illustrative example which allows to show how aspects can be customized using our approach. The same strategies, that we are going to show in the Logging example, can be used to customize other and different reusable aspects in more complex architectures, such as frameworks or product lines [2, 9, 11].

4.1. Domain Implementation

We first present the domain implementation activities which prepare an AO architecture to be automatically instantiated.

Activity 1*: AO Architecture Implementation.*

The first activity of the domain implementation is to implement an AO family architecture that addresses a

set of variabilities in a specific domain. The Logging aspect example (presented in Section 2) defines two variabilities: (i) *the logging pointcuts* – which represent the execution join points in the web system that will be logged; and (ii) *the logging persistence mechanism* – which defines alternative persistence ways to store the logging information. These variabilities are addressed by the aspect/class hierarchy presented in Figure 1.

The Logging subaspects are the only elements which need to be customized during the instantiation of a web system. In our approach, every element (class, aspect, interface or configuration file) which need to be customized during application engineering is implemented as a code template. Code templates allow us to represent structure and behavior of specific classes and aspects that we want to generate. Java Emitter Templates (JET), a generic template engine of the Eclipse Modeling Framework (EMF) [4], has been used to specify our templates. Thus, to specify the general structure of our Logging subaspects, we defined the `ConcreteLogging` JET template. It is used to generate specific logging subaspects. The `loggingJoinPoints` pointcut and `getLoggingPersistence()` method are customized based on information collected by the feature model. Our templates are processed by our code generator during architecture customization.

Activity 2*: Representation of the Variabilities in the Feature Model.*

After implementing all the variabilities of an AO architecture, the next activity is to represent them in a feature model. We use the feature model proposed in [6], which allows modeling mandatory, optional and alternative features, as well as their respective cardinality. The feature modeling plugin (fmp) [3] supports the modeling of feature models in Eclipse platform.

In order to support the customization of aspects in our approach, we have extended the feature model proposed in [6]. We can assign the *crosscutting* or *joinpoint* property to specific features. A *crosscutting feature* is used to represent aspects which can extend the behavior of other system features. A *joinpoint feature* is used to specify specific execution points in the system which can be extended by aspects. Crosscutting relationships between these elements can be defined in the application engineering (Section 4.2) in order to customize aspects to affect specific parts of the system.

Figure 3 shows the feature model of the Logging example using the fmp plugin. We first represent the business and data services which are provided by the web system. These features are all represented as mandatory (symbol ●), because they do not represent variabilities in the web system. They were only modeled because they represent possible features which the application engineer can desire to log their

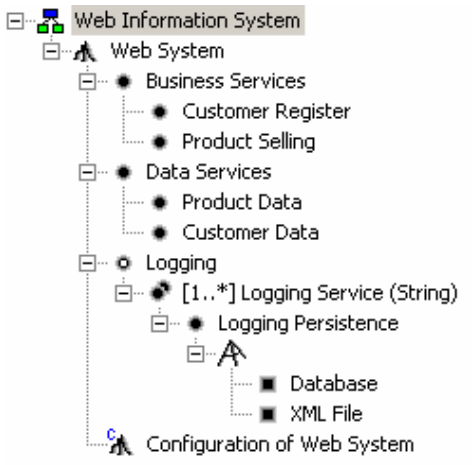

Figure 3. Logging Feature Model

execution. Since they are all candidates to be extended by crosscutting features, we call them *joinpoint features*.

Figure 3 also presents the logging variabilities. The logging feature is optional (symbol ○). It is composed by a set of logging services. Each logging service represents a possible logging subaspect to be created. The definition of a logging service feature involves the choice of one between two alternative persistence mechanisms: Database or XML File features. Each logging service needs also to be configured to extend specific services of the web system. Because of that, they are characterized as *crosscutting features*.

Activity 3*: Specification of the Configuration Model.*

The last activity of the domain implementation is the configuration model specification. The configuration model represents the configuration knowledge [5] in generative programming. It is used mainly to define how a specific configuration of features is mapped to a configuration of architecture components. All the information specified in the configuration model is used by our code generator to enable the automatic customization of AO architectures during application engineering.

Our configuration model is composed by three different elements: (I) description of dependency relationships between the architecture model's components (and sub-elements) and the features specified in the feature model; (II) definition of valid crosscutting relationships between *crosscutting* and *joinpoint* features; and (III) specification of the

mapping between *joinpoint* features and specific joinpoints in classes of the AO architecture. All these elements are being implemented as wizards of Eclipse [14] plugins.

The first element of our configuration model are the dependency relationships between architecture components (and sub-elements, such as, classes, aspects and templates) and features. They are used to define the mapping between the feature model and the architecture components. They allow to specify which components must be instantiated when specific features are selected. The following guidelines are used when defining the dependency relationships: (i) if a component (or sub-element) must be instantiated to every product of the product line, then no dependency relationships needs to be specified; (ii) if a component (or sub-element) depends on the occurrence of a specific feature, a dependency relationship must be created between them.

The dependency relationships are used by our code generator to decide which classes and aspects will be included in a product based on feature model instances defined by application engineers. In case of templates, the dependency relationships define if they will be processed and included in the final product generated. Every template element depends on specific features which provide knowledge necessary for their instantiation.

Figure 4 shows a set of dependency relationships between the Logging component and its respective feature model. It shows that the logging component will be instantiated only if the logging feature is selected by the application engineer. When the logging feature is selected, every element inside the logging component which does not have a dependency relationship with any feature will be automatically instantiated in the architecture. This is the case of the `Logging` abstract aspect and the `LoggingPersistence` class. Figure 4 also presents that the `ConcreteLogging` aspect template has a dependency relationship with the logging service feature. It means that a new different concrete logging aspect will be created for each logging service feature specified. Finally, `DatabaseLogging` and `XMLFileLogging` classes will be instantiated only if the database and XML file features, respectively, were requested by the application engineer.

The second element defined in our configuration model is the potential relationships between crosscutting and joinpoint features. This information is used by our code generator to check if the application engineers have specified valid crosscutting relationships. It allows to restrict the set of valid crosscutting relationships. Figure 4 shows the set of valid crosscutting relationships for the Logging example. It shows that every logging service feature can extend the following features: register and selling services, and product and customer data services features.

The third and last element of our configuration model is the mapping between the joinpoint features and the concrete joinpoints in classes of the AO architecture. This information is used by our code generator to customize pointcuts during the generation of aspects. The mapping involves the identification of which parts of classes (e.g.: constructor execution and method call) correspond to specific joinpoint features. In our particular implementation, the mapping refers to specific and valid AspectJ joinpoints. Figure 4 shows the joinpoint mapping for the Logging example. The business and data services joinpoint features of the web system are mapped to specific joinpoints existing in the implementation of its components.

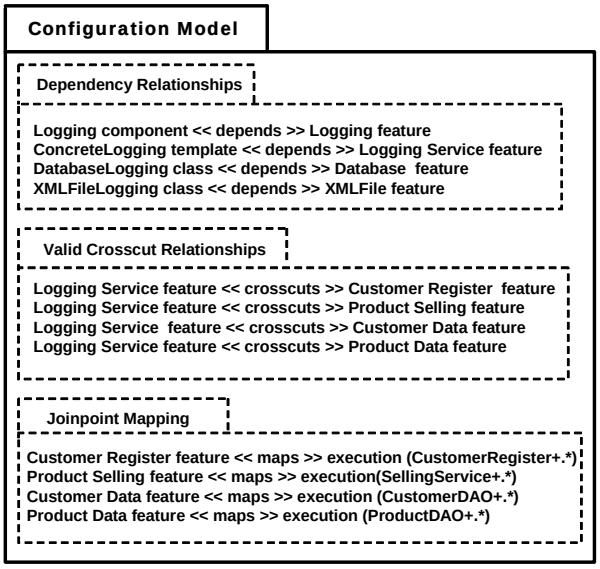

Figure 4. Logging Configuration Model

4.2. Application Engineering

In application engineering, developers request an instance of the AO architecture by specifying all desired variabilities. This request is composed of two activities: (i) choice of variabilities in a feature model instance; and (ii) choice of valid crosscutting relationships between features. This latter step is used to enable the customization of aspect pointcuts. A tool uses the information collected by these steps and the configuration model to generate an instance of the AO architecture.

Figure 5 shows a feature model instance of the Logging example. The application engineer is requesting two different logging services. The first one

is used to log information about the registration and selling business services. The other one logs information about the database access of the selling and data services. Each of them uses a different way to persist the logging information. The crosscutting relationships between features are specified separately from the feature model instance. For the Logging example, four crosscutting relationships must be defined: (i) the "Business Services" logging service crosscutting feature is related with the register and selling business services joinpoint features; and (ii) the "Data Services" logging service feature is related with the product and customer data services joinpoint features.

Using the feature model instance, the configuration model and the AO architecture of the Logging example, our code generator creates two Logging subaspects which affect the business and data services of the architecture. The complete algorithm of our code generator is described in [10].

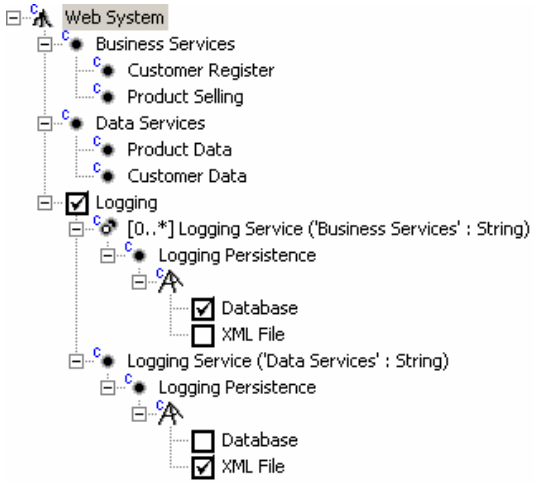

Figure 5. Logging Feature Model Instance

5. Conclusions and Future Work

This paper presented an aspect-oriented generative model which addresses the instantiation of variabilities encountered in AO architectures. We also described a set of domain implementation and application engineering activities which are adopted to prepare AO architectures to be automatically instantiated. To the best of our knowledge, the only research work which explores the instantiation of AO architectures is the Framed Aspects approach [13]. It proposes the integration between Frame and AOP technologies. The main difference between our and the Framed Aspects approach, is that they define many of the decision steps about the instantiation process in the template code of frames by means of meta-tags. In our approach, the decisions related to the architecture customization process are described separately by our configuration model. It makes easier to adapt or evolve the decisions related to the architecture customization. We also use feature model instances to gather all information necessary for the resolution of AO variabilities.

We are currently implementing a tool, as an Eclipse plug-in [14], which supports all the models presented in the paper. Also, new case studies involving software families from different domains are being realized to validate our approach. We are also exploring the instantiation of aspect libraries using our approach in these case studies. Finally, we are also refining a set of guidelines to modularize the implementation of framework variabilities using aspects [9].

Acknowledgments. The authors have been partially supported by CNPq, FAPERJ and European Network of Excellence on AOSD (AOSD-Europe).

References

[1] AspectJ Team. The AspectJ Programming Guide. http://eclipse.org/aspectj/.

[2] V. Alves, et al. "Extracting and Evolving Mobile Games Product Lines". In 9th International Software Product Line Conference (SPLC'05), September 2005.

[3] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004.

[4] F. Budinsky, et al. Eclipse Modeling Framework. Addison-Wesley, 2004.

[5] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[6] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In Proceedings of the Third Software Product-Line Conference, September 2004.

[7] R. Filman, T. Elrad, S. Clarke, M. Aksit. Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005.

[8] G. Kiczales, et al. "Aspect-Oriented Programming". Proc. of`ECOOP'97, LNCS 1241, Springer, Finland, June 1997.

[9] U. Kulesza, et al. "Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming", Proceedings of ICSR´2006, Turin, Italy, June 2006.

[10] U. Kulesza, et al. "Instantiating and Customizing Product Line Architectures using Aspects and Crosscutting Feature Models". Proceedings of the Workshop on Early Aspects, OOPSLA'2005, October 2005, San Diego.

[11] U. Kulesza, et al. "A Generative Approach for Multi-Agent System Development". In "Software Engineering for MAS III". Springer, LNCS 3390, pp. 52-69, 2004.

[12] M. Mezini, K. Ostermann: "Variability management with feature-oriented programming and aspects". Proceedings of FSE'2004, SIGSOFT, pp. 127-136, 2004.

[13] N. Loughran, A. Rashid. "Framed Aspects: Supporting Variability and Configurability for AOP". Proceedings of ICSR'2004, pp. 127-140, 2004.

[14] S. Shavor, et al. The Java Developer's Guide to Eclipse. Addison-Wesley, 2003.