Aspect-oriented Code Generation

**Marcelo Victora Hecht, Eduardo Kessler Piveta,
Marcelo Soares Pimenta, R. Tom Price**

Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Campus do Vale - Bloco IV
Bairro Agronomia - Porto Alegre - RS -Brasil
CEP 91501-970 Caixa Postal: 15064

{mvhecht, epiveta, mpimenta, tomprice}@inf.ufrgs.br

Abstract. *The maturing of aspect-oriented software modeling approaches provides support for the automatic generation of aspect-oriented code. In this paper we describe several means for automatic code generation from Theme/UML models, and discuss some difficulties involved in this process.*

Resumo. *O amadurecimento das abordagens de modelagem de software orientado a aspectos fornece subsídios para a geração automática de código a partir de modelos UML. Neste artigo são descritos diversos mecanismos para a geração de código a partir de modelos Theme/UML, e são discutidas algumas dificuldades envolvidas neste processo.*

1. Introduction

High-level programming languages and Code generation are among the oldest and most widespread techniques to accelerate the transformation process between the design of software and its implementation in executable code. Programming languages facilitate the expression of the solution for a given problem, reducing the amount of code needed and increasing the proximity between the software and the artifacts whose behavior it tries to represent. Code generation, by its turn, reduces the amount of code that has to be written, eliminating the necessity of manual labor, in addition to reduce the chance of accidental programming errors and simplifying the propagation of changes in design.

In the context of programming languages, one of the more recent advances is *aspect-oriented programming* [Kiczales, *et al.* 1997]. It allows an explicit separation of concerns that affect several parts of a software system, like security, persistence and tracing. In traditional analysis and design methods, those concerns end up scattered throughout a system and tangled with its functional requirements – this is called *crosscutting*. Aspect-oriented software development allows those concerns to be modeled and implemented independently from each other and from the main concerns of the system, improving several software quality factors such as extensibility, maintainability and reusability.

In automatic code generation, extensive use has been made of standard system modeling languages as UML [OMG UML 2004], allowing software specifications that are detailed, understandable and with less ambiguities. Currently, a major topic of research within code generation is MDA (Model-Driven Architecture) [OMG MDA 2001], a normalization of the code generation concept using UML models, as well as other elements associated to them, such as MOF [OMG MOF 2006], the metamodel on which UML itself is defined, and XMI [OMG XMI 2005], the standard for interchange of UML data using XML documents. The goal of MDA is to reduce as much as possible the cognitive distance between a system's design and its final implementation.

The use of aspect-oriented programming generated a demand for software design techniques that support its characteristics. One of the proposals for this is the Theme/UML approach [Clarke 2001, Clarke and Baniassad 2004] using *Composition Patterns.* It has been continuously developed since 1998 [Rashid, *et al.* 2005], aiming at managing the separation of concerns earlier in the development cycle. Initially focused on *subject-oriented programming* [Clarke, *et al.* 1999], it evolved to become the most general proposal currently available, and at the same time compatible with several aspect-oriented programming languages [Clarke and Walker 2001, 2002].

The existence of this type of model, and of its computer-understandable versions (through XML-based representations), opens up the path to several possibilities, such as quality metrics analysis, *bad smell* detection, refactoring, aspect extraction from traditional object-oriented programs, and automatic aspect-oriented code generation.

The goal of this paper is to describe a way to generate code in the AspectJ language [Kiczales, *et al.* 2001], the most extensively used aspect-oriented programming language today, from UML models extended according to Theme/UML [Clarke 2001]. To this purpose, we developed a XML representation of Theme/UML models, to serve a the input for a code generator programmed using XSLT (Extensible Stylesheet Language Transformations) [XSLT 2.0 2005].

The paper has been divided as follows: in Section 2, we revise the requirements to build a code generator, as well as the state of the art in this area, including MDA; in Section 3, we briefly investigate possible intersections between code generation and aspect-oriented programming; in section 4, we present a case study of the implementation of a aspect-oriented code generator based on Theme/UML; Section 5 cites related work in the area of code generation and aspect-oriented programming; and Section 6 contains our conclusions.

2. Code Generation

Code generation is the process of transforming high level artifacts – closer to the problem domain – in lower-level ones – nearer to the solution's architecture [Krueger 1992]. Although the term is used in various contexts, in this paper we will assume that it refers to the automatic conversion of a specification, in the form of high level models (like UML), into code (high level or binary), circumventing costly and error-prone cycles of manual programming.

2.1. Requirements for code generation

A code generator has some prerequisites. A model of the desired system has to be provided and

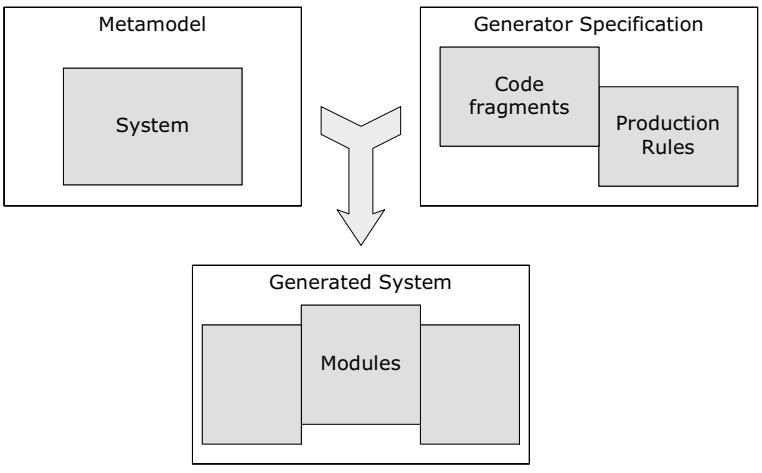

Figure 1. Typical structure of a code generator.

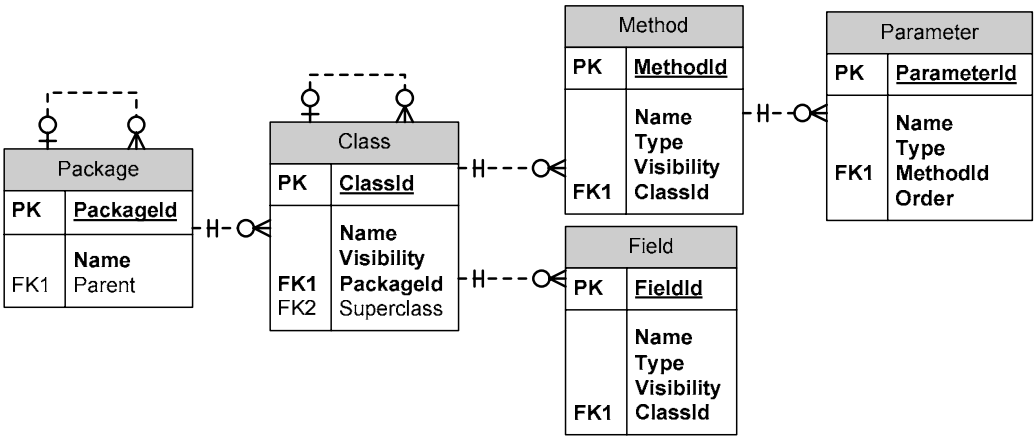

Figure 2. Metamodel for a simplified Java class model.

described within a metamodel that is sufficiently detailed and free of ambiguities to provide the structure to be generated. Also necessary is a specification of the elements of the target domain that will be composed to shape the final product, and rules, based on the contents of the metamodel, specifying how this composition is accomplished (see Figure 1).

The metamodel is a data model able to store other system models in a way that a computer can understand. It has to be specific to the generator's purpose: a metamodel used by an interface generator stores information about forms, fields, and user messages, while a metamodel for object-oriented software contains specifications of packages, classes, properties and methods. In Figure 2 we describe a metamodel regarding a simplified Java class model.

A code generator specification can be divided in two main pieces: the *code fragments* and the *production rules*. Code fragments are usually pieces of programs, in the generator's target language, that will be assembled to generate the source code that will be the output of the generator. They may have sections that are replaced with information contained in the metamodel, such as element names. Production rules define how the structure described in the metamodel will be converted to the structure of the composed code fragments in the output files. It usually contains alternative, iterative and recursive elements.

Simpler code generators, usually developed for use by a small team to solve a specific problem, such as creating object-relational mapping classes, generally don't make a clear distinction between the two parts. Code fragments are strings embedded in statements of an all-purpose programming language that shape the production rules. More mature generators divide the two parts more clearly, and frequently provide a form o programming devised specifically for the definition of production rules.

Many modeling tools commercially available have integrated code generators ([Rational Software 2005, ArgoUML 2005, Poseidon 2005]). They usually generate code according to a fixed structure, or allow limited modification. There are also several tools available that are designed specifically to build code generators ([codegen 2005, Velocity 2005, autogen 2005, CodeSmith 2005]). Practically all of them combine a script language for the programming of the production rules, and code fragments either embedded in the script or in separate files. One alternative, with the increasing popularity of the XML metalanguage for data interchange, is its use both to declare code fragments and to store the metamodel, and the use of XSLT [XSLT 2.0 2005], a programming language specialized in the transformation of XML documents, to define production rules [Dodds 2005].

2.2. Difficulties in Code Generation

The adoption of code generators can be hard, despite the gains this technology can convey. The first difficulty is economical: for something to be reused, it first must be developed, and sometimes this expense can't be justified [Biddle, *et al.* 2003]. Other common demands for to any kind of software, like the necessity of a detailed documentation, and concern for changes that can cause collateral effects with interacting systems, become especially important with code generators [Krueger 1992]. To maximize the gain offered by the generator, it is important to educate as many users as possible about its capabilities. And a badly planned modification in a code generator might impact the entire code base that depends on it.

Another problem comes up when the standard for the generated artifacts need to be modified, because changes made to those after the generation may be lost. Code generators can be divided as *passive* – creating each artifact only once – or *active* – designed to generate the same artifact many times according to changes in the model or in the generator [CGN 2005]. Passive generators create the initial code for the software, but can't be used for maintenance of already modified artifacts, and thus don't have the benefit of propagating changes in the model or in the generated standard to an existing system. Active generators allow the same code module to be generated any number of times, by the use of features like protected code sections that are copied from one version to the next.

Finally, often users of a generator face the need to use newer versions of a generator, in order to use new features, but to simultaneously keep in place older versions, to keep the compatibility with existing artifacts. Therefore, code generators are yet another dimension where maintainability problems can take place.

In section 3 we suggest ways by which AOP can help with some of those issues.

2.3. Model Driven Architecture

The advent of UML as a *de facto* standard for object-oriented software specification and modeling made it also the metamodel of choice to feed code generators. Because of that, the OMG developed the MDA (*Model Driven Architecture*) [OMG MDA 2005, Miller and Mukerji 2003] standard, with the intention of offering an vendor- and technology-independent platform for application development. According to OMG, development using MDA focuses in the functionality and behavior of a system, undistorted by the idiosyncrasies of the technological platform where it will be used. MDA attempts to separate implementation details from the business requisites, and consequently it is not necessary to repeat the whole process of modeling application requisites when there is a technology change.

MDA is based on several OMG specifications: the UML language is used to specify the system; the MOF (MetaObject Facility [OMG MDA 2005]) is used to store the UML model; and XMI (XML Metadata Interchange [OMG XMI 2005]) is used for the communication between the metamodel and the code generator tools.

3. Applying aspect-oriented concepts to code generation

Application generators are one of the most powerful reuse techniques available, able to output thousands of lines of code from a relatively small input set. On the other hand, they are also one of the least adaptable techniques, each one being designed to a specific type of output. High-level programming languages are also a form of reuse, in the sense that they allow more functions to be included in a system with less lines of code [Biggerstaff 1999].

It is appealing then to investigate how aspect-oriented programming can be combined to automatic code generators in order to combine the benefits of both technologies. This intersection may happen in two ways: aspect-oriented programming can be used in the

development of code generators, and a code generator can have aspect-oriented code as its target. While our focus in this paper is on the latter approach, in this section we will briefly investigate the former.

Initially, the advantages that AOP brings to the development of code generators are the same that it adds to any system: simplified development, ease in maintenance, increased understandability, etc.

Some ideas from aspect-orientation may be especially useful for the creation of code generators. It can, for example, be used to parameterize a generator, so that it can generate different outputs, according to the aspects selected to be weaved with it. This device can be used even to maintain several versions of the generator active.

Aspects may also be used to simplify the problem of evolving code generators, if the modification of the generated artifacts is implemented through advices and intertype declarations. That would allow the base artifacts to be generated again at will without breaking the modifications (unless join points in the base artifact were modified). For that to be possible, the generator itself could be unaware of AOP, as long as the programming language of the generated artifacts contemplates concern separation, or has some extension for that purpose. As an example, classes created by a Java code generator can be modified by aspects using AspectJ.

Biggerstaff [Biggerstaff 1998] draws a parallel between code generation and aspect-oriented programming, stating that code generation is a form of weaving code fragments. He also suggests that aspect-oriented techniques may increase the power of component libraries, granting the ability to change its components and multiplying their capabilities.

4. Implementing an automatic generator for Aspect-oriented code

In this section, we describe our efforts to implement an automatic code generator capable of transforming an aspect-oriented model into aspect-oriented code. We do not propose, at this time, to contemplate the contents of methods and advices, although, with sufficiently detailed interaction diagrams, it might be possible.

4.1. Challenges in generating aspect-oriented code

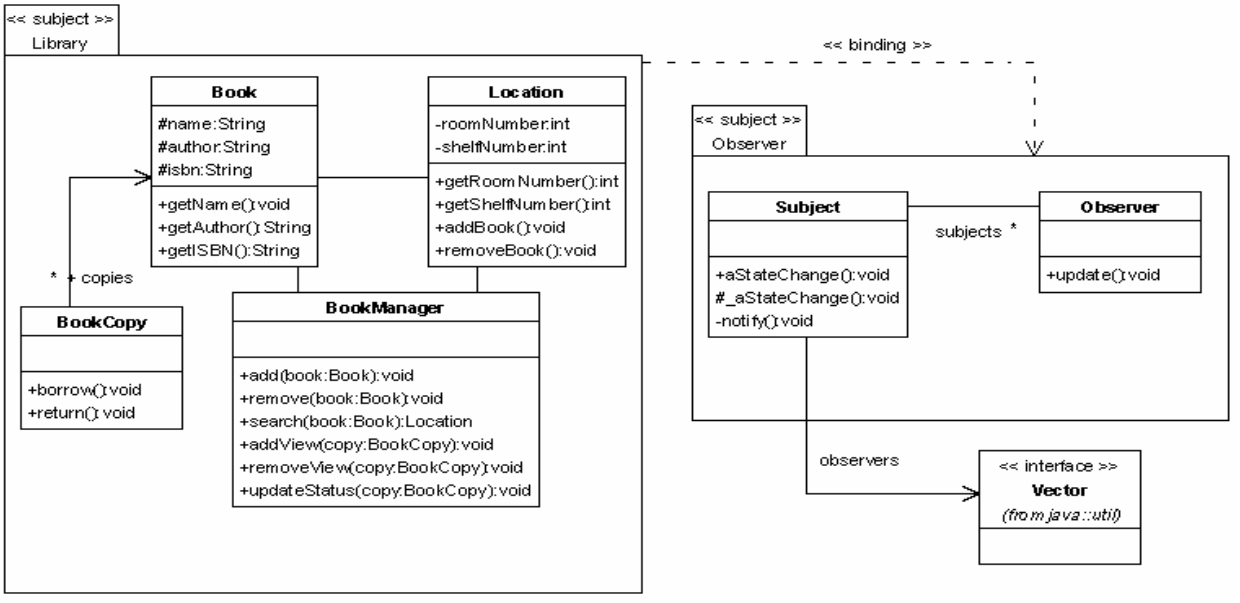

Figure 3. UML model of the example system (without Theme/UML extensions)

```
package library;

import java.util.Vector;

public abstract aspect Observer {
  interface ISubject {}

  interface IObserver {
    public void update();
  }

  Vector ISubject.observers;

  private void
    ISubject.notify_() {}

  abstract pointcut
    aStateChange(ISubject s);
  after(ISubject s):
    aStateChange(s) {
    s.notify_();
  }
}
```

```
package library;

public aspect LibraryObserver
    extends Observer {
  declare parents: BookCopy
    implements ISubject;
  declare parents: BookManager
    implements IObserver;

  public void BookManager.update() {
    this.updateStatus();
  }

  pointcut aStateChange(ISubject copy):
    target(copy)
    && args(..)
    && (execution
       (* BookCopy.borrow(..))
     || execution
       (* BookCopy.returnIt(..)));
}
```

Figure 4. Listing of AspectJ code for the *Observer* composition pattern and *LibraryObserver* binding (manually indented for better understanding).

A problem in generating aspect-oriented code is the lack of a standard modeling form that explicitly contemplates separation of concerns. Most of the proposed methods ([Groher and Baumgarth 2004, Chavez and Lucena 2002, Stein, *et al.* 2002, Basch and Sanchez 2003], [Pawlak, *et al.* 2002]) provide, as evidence of its validity, the correspondence between the models they define and an implementation in AspectJ. However, none has the required level of detail for automatic code generation, relying on human interpretation of the models to create code.

The paper that presents the most detailed implementation is [Clarke and Walker 2002], where the authors try to demonstrate that the Theme/UML approach [Clarke and Baniassad 2004], based on the *composition patterns* approach, is compatible with several implementations of aspect-oriented programming, including AspectJ. Even so, their approach is very complex, and the mapping between it and AspectJ is not trivial.

We developed a library maintenance system similar to the one used as an example in [Clarke and Walker 2001, 2002]. The base system controls the location of books, as well as loans and returns. Over that, the Observer design pattern [Gamma, *et al.* 1994] is applied as a crosscutting concern, for the book manager class to be aware when a copy is lent or returned (Figure 3). On aspect weaving, the *aStateChange* advice of *Subject* is to be associated with the join points corresponding to the *borrow()* and *return()* methods from class *BookCopy*, while the *update()* method from *Observer* is to be introduced to class *BookManager*. The use of aspects to implement design patterns is explored in more depth in [Garcia, *et al.* 2005] and [Hannemann and Kiczales 2002].

The target code for the system is presented in Figure 4. For reasons of reusability and evolvability described in [Clarke and Walker 2002], each CP in the model becomes an abstract aspect with abstract pointcuts, and the binding between the CP and the base packages turns into an aspect that makes those pointcuts concrete. This process is detailed in section 4.4. The code depicted is the output of the code generator; the only hand-made modification made was in the formatting and indentation for presentation purposes.

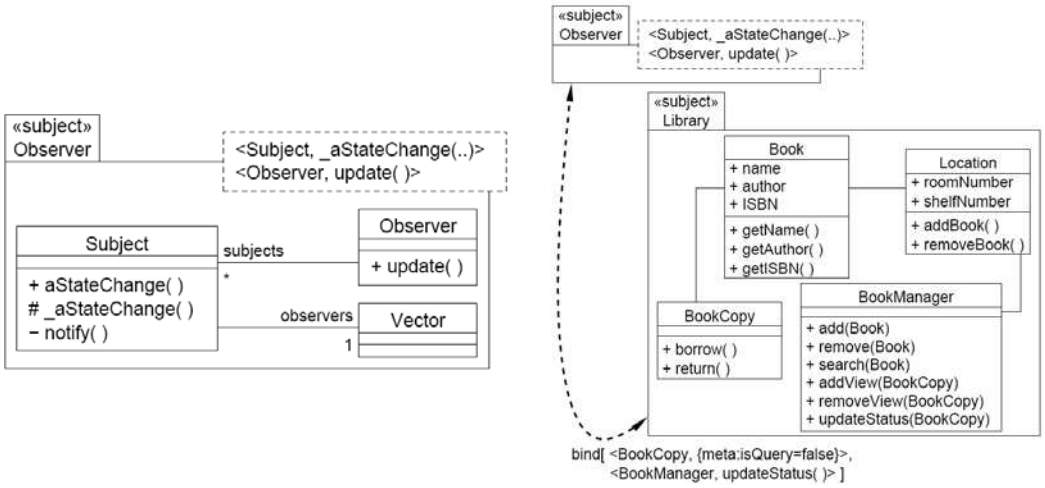

Figure 5.Class diagram using *Composition Patterns*

4.2. The Theme/UML approach [Clarke and Baniassad 2004]

This approach is divided in two parts: Theme/Doc, for requisite engineering, and Theme/UML, for system design, both considering separation of concerns in every level. Our interest here is in the second one.

The construction added by Theme to UML is the *composition pattern* (in short, CP), a definition of how to integrate artifacts (classes or operations) from two different packages. They work in a manner similar to UML *templates*, that allow elements in a model not to be fully defined, but to have *parameters* so that specific parts are replaced later by concrete elements of the model.

As used in Theme/UML, template parameter substitution can be used both to indicate dynamic crosscutting, where advices are connected to join points in the system through pointcuts, and to produce static crosscutting, where the very structure of the base system is modified [Clarke and Walker 2002], as can be seen in Figure 5. Advices are always represented by a pair of operations. The one with the name prefixed by an underscore represents the execution of the base method. Representing those operations on an interaction diagram, it is possible to define *before*, *after* and *around* advices. In the example displayed in Figure 6, the *aStateChange()* operation (the advice) executes the *_aStateChange()* operation firstly, and as a result represents an advice of type *after*.

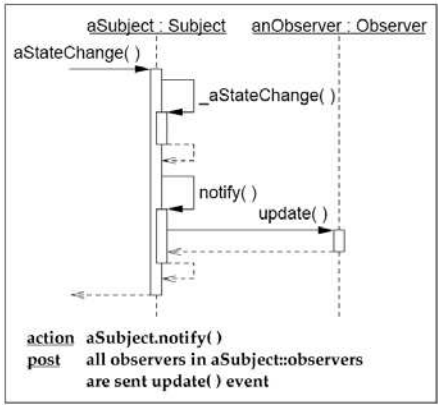

Figure 6. Sequence diagram depicting the interaction between a *composition pattern* and a modified class

```
<UML:Package xmi.id="c28" name="Observer" visibility="public"
isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref="d6a"/>
  </UML:ModelElement.stereotype>
  <UML:Namespace.ownedElement>
    <Theme:CompositionPatternParameters xmi.id="03a9">
      <UML:Operation xmi.id="8e09" xmi.idref="d4a"/>
      <UML:Operation xmi.id="8e1c" xmi.idref="d2a"/>
    </Theme:CompositionPatternParameters>
```

Figure 7. Section of a XMI document defining parameters for a package representing an Composition Pattern

4.3. Metamodel

It was necessary to define a computer-understandable format to store the class model so that it could be analyzed by the code generator system. Taking in consideration the predominance of the UML and XML metalanguages, respectively on the areas of system modeling and information exchange, it was natural to choose XMI, which combines both technologies, as the input of the generator. But there is no definition of the precise format a UML model should be stored, and no common schema for XMI documents. Instead, OMG defines a set of rules that vendors have to follow to define their own schemas. In this work, we started from the XMI generated by the Poseidon tool [Poseidon 2005], and added the necessary information to specify composition patterns – template parameters associated to packages, and parameter binding to relationships between packages.

Basically, two structures had to be added to the original XMI file. An element, labeled `CompositionPatternParameters`, is used inside a package (Figure 7) to indicate elements that can be bound to actual classes during composition to a base design – in this case, the operations *Subject._aStateChange()*, that represents an aspect, and *Observer.update()*, that is going to be

```
<UML:Dependency xmi.id="c26" isSpecification="false">
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref="c24"/>
  </UML:ModelElement.stereotype>
  <UML:Dependency.client>
    <UML:Package xmi.idref="d6c"/>
  </UML:Dependency.client>
  <UML:Dependency.supplier>
    <UML:Package xmi.idref="c28"/>
  </UML:Dependency.supplier>
  <Theme:Bindings>
    <Theme:Bind>
      <UML:Class xmi.idref="e10" />
      <Theme:BoundParameter xmi.idref="8e09" />
      <UML:Constraint xmi.id="d88" name="" isSpecification="false">
        <UML:Constraint.body>
          <UML:BooleanExpression xmi.id="d8a" body="meta:isQuery=false"/>
        </UML:Constraint.body>
      </UML:Constraint>
    </Theme:Bind>
    <Theme:Bind>
      <Theme:BoundElement xmi.type="uml:Operation" xmi.idref="d74" />
      <Theme:BoundParameter xmi.idref="8e1c" />
    </Theme:Bind>
  </Theme:Bindings>
</UML:Dependency>
```

Figure 8. Section of a XMI document extending a dependency between packages with parameter binding.

Input: A composition pattern *CP*.
Output: An abstract aspect *A* corresponding to *CP*.

- Declare abstract aspect *A*.
- For each pattern class $PClass_i$ in *CP*:
  - Declare interface I_i in *A*.
  - For each template operation with no supplementary behaviour, $\langle op \rangle_{i,j}$, on $PClass_i$, declare corresponding (abstract) method $m_{i,j}$ on I_i.
  - For each template operation with supplementary behaviour[2], $\langle \_op \rangle_{i,k}$, on $PClass_i$:
    - Declare corresponding abstract pointcut $pc_{i,k}$ with formal parameters consisting of one to capture target object (of type I_i) on which $\langle \_op \rangle_{i,k}$ is called plus one corresponding to each of the specified formal parameters on $\langle \_op \rangle_{i,k}$.
    - Declare advice on $pc_{i,k}$ according to the supplementary behaviour.
  - For each non-template operation $op_{i,l}$ on $PClass_i$, introduce method $m_{i,l}$ (which implements any behavioural specifications for $op_{i,l}$) on I_i.
  - For each non-template association from $PClass_i$ to some non-pattern class, introduce field on I_i.
- For each non-pattern class in *CP*, implement it directly if not already available.

Figure 9. Algorithm for mapping certain constructs in an uncomposed CP to AspectJ [Clarke and Walker 2002].

introduced on a base class. Another element, called `Bindings` (Figure 8), is associated to a dependency relationship between the base package and the Composition Pattern, and indicates how the composition is to proceed.

4.4. Generator and Production Rules

The generator itself was implemented using XSLT (Extensible Stylesheet Language Transformations), version 2.0 [XSLT 2.0 2005]. This language was selected because it is a standard language, with extensive tool support, and its focus on transforming XML documents

```
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:fn="http://www.w3.org/2005/02/xpath-functions"
   xmlns:xdt="http://www.w3.org/2005/02/xpath-datatypes"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:UML="org.omg.xmi.namespace.UML"
   xmlns:Theme="org.my.Theme">
 <xsl:import href="Theme2AspectJAspect.xslt"/>
 <xsl:import href="Theme2AspectJBinding.xslt"/>
 <xsl:import href="Theme2AspectJClass.xslt"/>

 <xsl:output method="text"/>
 <xsl:output method="text" name="textFormat"/>

 <xsl:template match="/">
   <xsl:apply-templates select="/XMI/XMI.content/UML:Model
/UML:Namespace.ownedElement//UML:Package/UML:Namespace.ownedElement
/Theme:CompositionPatternParameters/../.." mode="T2AJAspect"/>
   <xsl:apply-templates
select="/XMI/XMI.content/UML:Model/UML:Namespace.ownedElement
//UML:Dependency/Theme:Bindings/Theme:Bind" mode="T2AJBinding"/>
   <xsl:apply-templates
select="/XMI/XMI.content/UML:Model/UML:Namespace.ownedElement//UML:Package
[count(UML:Namespace.ownedElement/Theme:CompositionPatternParameters)=0]"
mode="T2AJClass"/>
 </xsl:template>
</xsl:stylesheet>
```

Figure 10. Section of XSLT transformation selecting the types of elements to be generated.

makes it very suitable for code generation based on XMI models [Marchal 2004].

It is important to emphasize, however, that the use of general-purpose XSLT processors for code generation is best suited for the creation of *passive* generators (section 2.2), because those processors have no support for integrating generated code with existing artifacts. If it is required that changes made to generated code are kept on within new generation cycles, other alternatives should be considered. Besides, the paradigms used on XSLT programming, based on tree traversals, are somewhat different from what programmers of imperative languages are used to. With few exceptions, XSLT processors also lack features like step-by-step execution and variable state evaluation, which would facilitate debugging in case the generator output does not correspond to the desired one.

The algorithm used to generate the files is similar to the one presented in [Clarke and Walker 2002] and reproduced in Figure 9.

As seen in the algorithm, each Composition Pattern becomes an *abstract aspect*, and the binding of the aspect to a class becomes a concrete aspect that realizes the abstract pointcuts defined in its parent. This promotes the reusability of the aspects.

Due to space constraints, it is impossible to reproduce the full listing for the XSLT files used in generation. We chose to represent here the main generator file, which distributes the appropriate elements among the three other files, which in turn are used to generate Aspects, Classes, and Bindings (Figure 10). Since in Theme Composition Patterns are just like any other UML Class Diagram package, except for the fact that they have parameters that can be bound to elements in other packages, the deciding factor for generating classes or aspects is the presence

```
<xsl:template match="UML:Package" mode="T2AJAspect">
   <xsl:variable name="uri" select="@name"/>
   <xsl:result-document href="{$uri}.aj" format="textFormat"><xsl:call-
template name="lowercase">
       <xsl:with-param name="name" select="@name"/>
     </xsl:call-template>;
<xsl:value-of select="@visibility"/> aspect <xsl:call-template
name="capitalize">
       <xsl:with-param name="name" select="@name"/>
     </xsl:call-template> {
     <xsl:apply-templates select="UML:Namespace.ownedElement/UML:Class"
mode="Interface"/>
}<!---->
   </xsl:result-document>
 </xsl:template>

 <xsl:template match="UML:Class" mode="T2AJAspect Interface">
 interface I<xsl:value-of select="@name"/> {
   <xsl:apply-templates select="UML:Classifier.feature/UML:Operation"/>
 }
 </xsl:template>

 <xsl:template match="UML:Operation" mode="T2AJAspect Interface">
   <xsl:if test="substring(@name, 1, 1) != '_' and
not(exists(../UML:Operation[@name = concat('_',
current()/@name)]))"><xsl:text>
   </xsl:text><xsl:call-template name="methodSig"/>;<!---->
   </xsl:if>

 </xsl:template>
```

Figure 11. Section of XSLT transformation converting packages from a XMI document into AspectJ aspects.

of an element called `CompositionPatternParameters` within the package.

We also reproduce a section of XSLT that generates abstract aspects from the definition of Composition Pattern packages (Figure 11). In this section, the packages received from the main file are processed according to the rules specified by the algorithm (Figure 9), generating an abstract aspect with interfaces for each class contained in the Composition Pattern and introducing methods for every operation that does not designate an advice.

5. Related works

In [Amaya, *et al.* 2005], an approach to deal with aspect oriented modeling in MDA is presented. In this approach, typical crosscutting concerns are considered as different perspectives of the system modeled using UML. This separation will keep from CIM to PSM. They also propose to model requirements using use cases and design them using composition patterns. In our approach, we do not deal with requirements, but the design is done using Theme/UML artifacts instead of subject-oriented design in UML. Theme/UML was designed with experiences in modeling subject-oriented (SO) systems, improving the former subject-oriented modeling techniques.

The Libra approach [Chaves 2004] describes potential benefits of mixing Model Driven Software Development and Aspect-Oriented Software Development. The authors propose an approach for combining them based on a new dynamic join point model for UML action semantics. Our approach could be adapted to work with the action semantics described by the authors. The main benefit of the Libra approach is the possibility to use of behavioral specification to describe aspect oriented software.

[Kulesza, *et al.* 2005] presents a generative approach for the development of multi-agent systems (MAS). The approach explores the MAS domain to enable the code generation of heterogeneous agent architectures. Aspect-oriented techniques are used to allow the modeling of crosscutting agent features. Although they provide a domain specific language (DSL) and an aspect-oriented architecture for agent systems, the relation to our work is associated to the code generation techniques. It maps abstractions of Agent-DSL to the depicted architecture. This is useful for agent-oriented systems, but not abstract enough to deal with other types of systems without adaptation. Our approach uses XMI to generate AO code, regardless of the system domain and reference architecture.

[Kulesza and Lucena 2004] provide a preliminary version of a method to develop aspect-oriented generative approaches, including a description of required adaptations of the domain engineering method to accommodate the use of aspect-oriented technologies. They describe adaptation to domain analysis, domain design and domain implementation. Our work focuses on the MDA view of code generation. Instead of using domain modeling, we focus on software modeling using UML models.

[González, *et al.* 2005] discuss principles to make an aspect-oriented analysis in software development through the identification and analysis of the dependencies model between system properties (in the context of MDA), but it does not include any attempt to implement those principles.

Finally, CAM/DAOP [Pinto, *et al.* 2005] seeks to combine component-based software engineering and aspect-oriented concepts with the use of an XML-based Architecture Description Language (ADL), called DAOP-ADL, which contemplates aspect composition. There is some similarity between the information conveyed by this ADL and by our XMI-based metamodel: they both attempt to represent aspect composition in a computer-understandable format. However, their approach is based in a proprietary platform for dynamic aspect weaving, while we base our efforts in existing modeling and programming languages.

6. Conclusion and Future Work

Aspect-oriented design and development and code generation are two technologies with a great potential to increase development productivity and software quality.

However, as we have shown, a few obstacles exist in the merging of the two technologies. The degree of productivity and quality gain from a code generator is directly dependent of the degree of quality of the metamodel it is based on, and of the preciseness of its production rules. This means that to create a generator for aspect-oriented code, we need to have a method of expressing separation of concerns in a software model in a way that is compatible with an aspect-oriented programming language.

We chose Theme/UML as the basis for our metamodel because it is a flexible and general modeling language. This is important because our interest is not only in the generation of aspect-oriented code, but also in use this model for code refactoring [Fowler 1999], detection of *bad smells* [Piveta, *et al.* 2005], and extraction of aspects from existing object-oriented systems [Monteiro and Fernandes 2005] (Figure 12). The basic structures of AspectJ – aspects, advices, pointcuts, join points, and introductions – can all be represented in Theme/UML. One of the tasks that our group is currently working on is to determine how each feature of AspectJ can be modeled in Theme/UML, and vice-versa.

We also chose to design the production rules so as to convert from Theme/UML directly to AspectJ code. The alternative approach would be to use a software model specific to the structure of AspectJ code, and to define a transformation in two stages: from Theme/UML to the AspectJ/UML model, and from this to AspectJ code. This would have the advantage of simplifying greatly the production rules at each step, making the code clearer and more maintainable. But the extra steps involved might be in itself a maintainability problem, especially if changes in AspectJ demand a modification of the intermediate model.

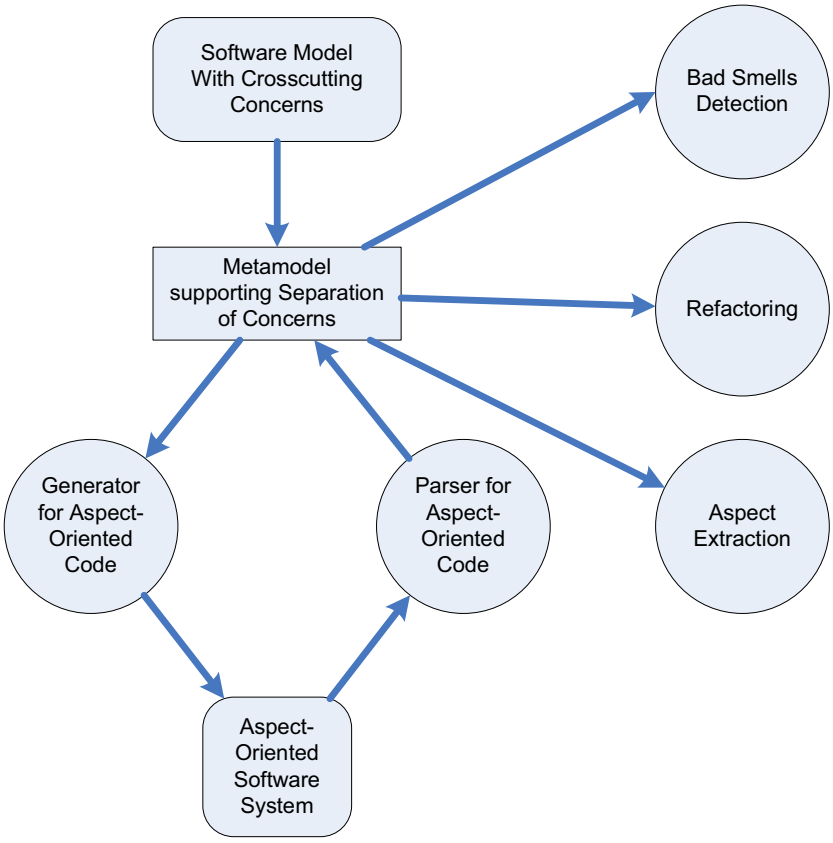

Figure 12. Activities based on a metamodel supporting separation of concerns

References

Amaya, P., Gonzalez, C. and Murillo, J. M. (2005), "Towards a Subject-Oriented Model-Driven Framework," in *Aspect-Based and Model-Based Separation of Concerns in Software Systems*.

ArgoUML (2005), http://argouml.tigris.org/, accessed on 2005-10.

Autogen (2005), http://www.gnu.org/software/autogen/, accessed on 2005-09.

Basch, M. and Sanchez, A. (2003), "Incorporating Aspects into the UML," in *Third International Workshop on Aspect Oriented Modeling*.

Biddle, R., Martin, A. and Noble, J. (2003), "No Name: Just notes on software reuse," in *ACM SIGPLAN Notices*, vol. 38, pp. 76-96.

Biggerstaff, T. J. (1998), "A perspective of generative reuse," in *Annals of Software Engineering*, vol. 5, pp. 169-226.

Biggerstaff, T. J. (1999), "Reuse technologies and their niches," in *Proceedings of the 21st international conference on Software engineering*. Los Angeles, California, United States: IEEE Computer Society Press.

Chaves, R. (2004), "Aspects and MDA: Creating aspect-based executable models," Master Thesis. Florianópolis.

Chavez, C. and Lucena, C. (2002), "A Metamodel for Aspect-Oriented Modeling," in *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*.

Clarke, S. (2001), "Composition of Object-Oriented Software Design Models," PhD thesis, Dublin City University, Dublin, Ireland.

Clarke, S. and Baniassad, E. (2004), "Theme: An Approach for Aspect-Oriented Analysis and Design," in *Proceedings of the International Conference on Software Engineering 2004*.

Clarke, S., Harrison, W., Ossher, H. and Tarr, P. (1999), "Subject-oriented design: towards improved alignment of requirements, design, and code," in *ACM SIGPLAN Notices*, vol. 34, pp. 325-339.

Clarke, S. and Walker, R. J. (2001), "Separating Crosscutting Concerns across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J," in *Technical Report TCD-CS-2001-15 and UBC-CS-TR-2001-05*, Trinity College, Dublin and University of British Columbia.

Clarke, S. and Walker, R. J. (2002), "Towards a standard design language for AOSD," in *Proceedings of the 1st international conference on Aspect-oriented software development*. Enschede, The Netherlands: ACM Press.

CGN (2005), Code Generation Network, http://www.codegeneration.net/, accessed on 2005-09.

Dodds, L. (2005), "Code generation using XSLT", https://www6.software.ibm.com/developerworks/education/x-codexslt/x-codexslt-3-1.html, accessed on 2005-12.

Codegen (2005), http://forge.novell.com/modules/xfmod/project/?codegen, accessed on 2005-09.

CodeSmith (2005), http://www.ericjsmith.net/codesmith/, accessed on 2005-09.

Fowler, M. (1999), "Refactoring: improving the design of existing code," Addison-Wesley Longman.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994), "Design Patterns," Addison-Wesley.

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C. and Staa, A. v. (2005), "Modularizing Design Patterns with Aspects: A Quantitative Study," in *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*.

González, C., Murillo, J. M. and Amaya, P. A. (2005), "Aspect-Oriented Analysis: A MDA Based Approach," in *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*.

Groher, I. and Baumgarth, T. (2004), "Aspect-Orientation from Design to Code," in *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*.

Hannemann, J. and Kiczales, G. (2002), "Design pattern implementation in Java and AspectJ," in *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. (2001), "An overview of AspectJ," in *Proceedings of the 15th Europeen Conference on Object-Oriented Programming (ECOOP 2001)*.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. (1997), "Aspect-Oriented Programming," in *Proceedings of the 11th Europeen Conference on Object-Oriented Programming (ECOOP-97)*.

Krueger, C. W. (1992), "Software Reuse," in *ACM Computing Surveys*, vol. 24, p. 131-183.

Kulesza, U., Garcia, A. F., Lucena, C. J. P. d. and Alencar, P. S. C. (2005), "A Generative Approach for Multi-agent System Development," in *SELMAS 2004 - Software Engineering for Multi-Agent Systems III*.

Kulesza, U. and Lucena, A. G. C. (2004), "Towards a Method for the Development of Aspect-Oriented Generative Approaches," in *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004*.

Marchal, B. (2004), "UML, XMI, and code generation", http://www-128.ibm.com/developerworks/xml/library/x-wxxm23/.

Miller, J. and Mukerji, J. (2003), "MDA Guide Version 1.0.1." http://www.omg.org/docs/omg/03-06-01.pdf.

OMG MDA (2005), Model Driven Architecture, http://www.omg.org/mda/, accessed on 2005-11.

OMG MOF (2006), MetaObject Facility Specification, version 1.4, http://www.omg.org/mof/, accessed on 2006-02.

Monteiro, M. and Fernandes, J. (2005), "Towards a Catalog of Aspect-Oriented Refactorings," in *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*.

Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L. and Martelli, L. (2002), "A UML Notation for Aspect-Oriented Software Design," in *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*.

Pinto, M., Fuentes, L. and Troya, J. M. (2005), "A Component and Aspect Dynamic Platform", in *The Computer Journal* 48(4):401-420.

Piveta, E. K., Hecht, M., Pimenta, M. S. and Price, R. T. (2005), "Bad Smells em Sistemas Orientados a Aspectos," in *Simpósio Brasileiro de Engenharia de Software (SBES)*, Uberlandia - Brazil.

Poseidon (2005), http://gentleware.com/, accessed on 10/2005.

Rashid, A., Chitchyan, R., Sawyer, P., Garcia, A., Alarcon, M. P., Bakker, J., Tekinerdogan, B., Clarke, S. and Jackson, A. (2005), "Survey of Aspect-Oriented Analysis and Design Approaches," in *AOSD-Europe 2005-05*.

Rational Software (2005), http://www-306.ibm.com/software/rational/, accessed on 2005-11.

Stein, D., Hanenberg, S. and Unland, R. (2002), "An UML-based Aspect-Oriented Design Notation," in Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002).

OMG UML (2004), Unified Modeling Language Specification, version 2.0, http://www.omg.org/technology/documents/formal/uml.htm, accessed on 2005-12-15.

Velocity (2005), http://jakarta.apache.org/velocity/, accessed on 2005-09.

OMG XMI (2005), XML Metadata Interchange Mapping Specification, version 2.1, http://www.omg.org/technology/documents/formal/xmi.htm, accessed on 2006-01.

XSLT 2.0 (2005), Extensible Stylesheet Language Transformations version 2.0, http://www.w3.org/TR/xslt20/, accessed on 2005-12.