

Aspects in UML Models from a Code Generation Perspective

Georg Beier
Westfälische Hochschule Zwickau (FH)
University of Applied Sciences
P.O.B. 20 10 37
D-08012 Zwickau
+49 375 536 1370
georg.beier@fh-zwickau.de

Markus Kern
syscovery netsoft gmbh
Weinheimer Str. 68
D-68309 Mannheim
+49 621 717 6840
markus.kern@syscovery.com

ABSTRACT

In this paper, we look at aspects from the perspective of code generation. From our experiences with template based code generators, we propose to attribute aspects to UML packages in order to improve code generation.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: Object-oriented design methods, Computer-aided software engineering

General Terms

Documentation, Design, Languages

Keywords

Aspect Oriented Modeling, Code Generation, Aspect Weaver

1. INTRODUCTION

Aspects describe crosscutting concerns that blur code structure. Separating concerns in object models is quite an old concept. Shlaer and Mellor introduced the notion of a domain as “a separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to rules and policies characteristic of the domain” [4]. These domains provide services that are used by others, their clients. Unfortunately, many of these services can not simply be called, but they affect the structure of their clients. E.g. using a persistence service for application objects will usually introduce additional attributes, methods and possibly base classes. Frankel et al. applied the domain concept to UML packages and described several mappings of services into client code [1, 2]. In the light of the concept of aspects, these are typical crosscutting concerns. Building on the above concepts, we propose

- to represent aspects as features of packages,
- to mark the application of aspects using tagged values, and
- a procedure to apply aspects for code generation.

2. EXPERIENCES WITH IMPLEMENTING ASPECTS IN CODE GENERATOR SCRIPTS

Until now, we have implemented aspects using a template based code generator [3]. Each aspect was coded as one or more templates that are included into higher level templates at appropriate positions, e.g. into the class declaration. This procedure proved to work in projects of size up to 6 person years. Aspect templates could be developed independently but needed some integration effort.

Appendix A shows a snippet of template code that generates support functions for drag & drop in C++. The complexity of the template code primarily depends on the complexity of the aspect. As can be seen from this code example, template code often involves a lot of navigation through the meta model.

2.1 Advantages of this Procedure

- Template development is straight forward. Typically, a prototype implementation of an aspect is turned into one or more templates.

2.2 Disadvantages of this Procedure

- Aspects exist in template code only, with no direct linkage to UML models. This obscures the structure of aspect implementations and their relationship with other model elements.
- Aspect modeling is target language dependent.
- Aspect templates are hard to reuse.

3. MODELING ASPECTS IN PACKAGES

Aspects can be attributed to packages that completely model one or more services, a “domain” in the sense of [4]. We will call these packages server packages. Elements of other packages, the clients, use certain services. They are possibly affected by the aspects of those services.

This is illustrated by the class model shown in Figure 1. A simple authentication and authorization package provides two services:

1. A client can be authenticated by using the login method. This is a simple service with little structural implications on client code. The signature of the method is all that matters.

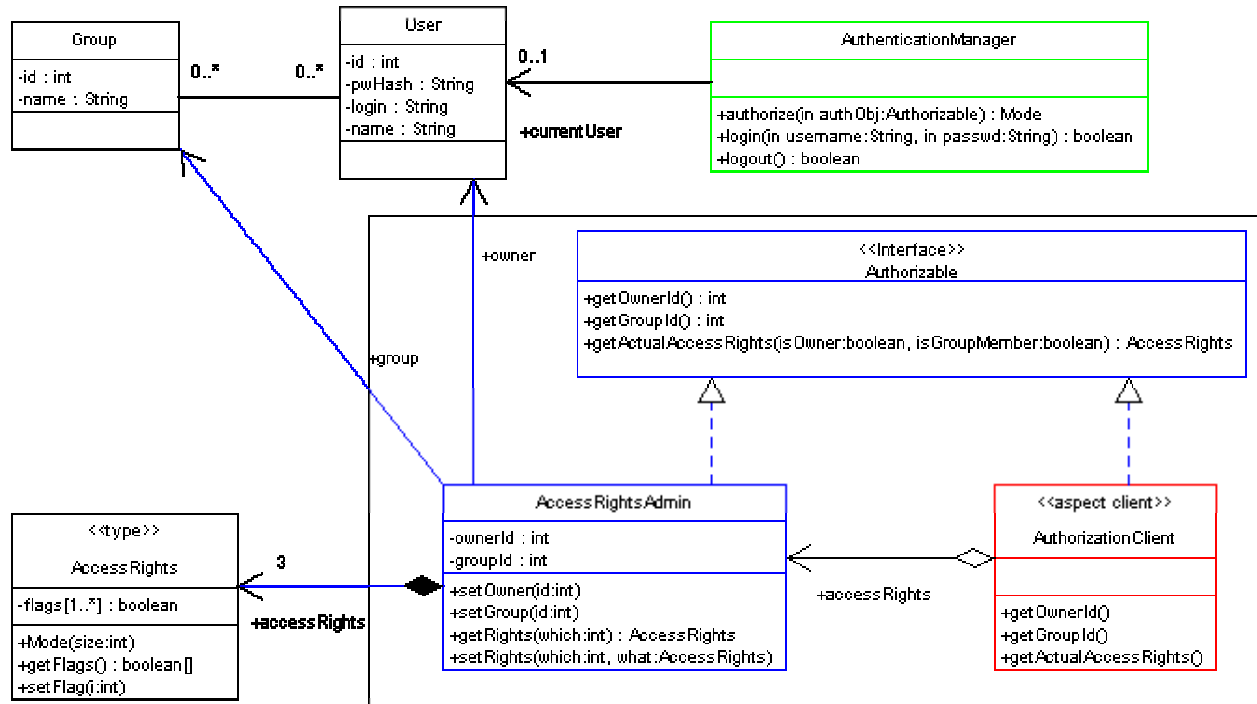


Figure 1: Class model showing aspect “authorization”

- An object can be authorized for some actions, e.g. read, write, delete. To efficiently use the provided authorization service, an object on the client side should implement the pattern shown in the box. The service not only consists of a method, but also an implementation structure. Code generation should generate the *implements* relation and the *accessRights aggregation* for every class that uses the authorization service. Additionally, methods should be generated that delegate the methods defined in interface *Authorizable* to the *AccessRightsAdmin* object. So this aspect has strong implications on the client code.

From our experiences, aspects can affect all model elements that are relevant in code generation, i.e. classes, attributes, methods, associations, state models etc. Their effect may show on two different levels of detail:

- Modifications on the UML model of the client: New model elements like attributes, methods, classes, and associations may be created, eventually replacing existing elements. We call this the structural part of an aspect because it changes the structure of the UML model.
- Modifications to the code generation inside methods: Code generator templates may be supplemented or replaced. We call this the coding part of an aspect. It is below the scope of UML.

For that reason, two different notations are needed. Effects affecting the structure of the UML model should be modeled graphically, using stereotypes to distinguish substitutes for client elements from model elements in the service package. Code generator templates can then be attached to the substitutes to keep all definitions for an aspect in a single place.

Model elements on the client side that use a specific service and thus are affected by ist aspect can easily be marked using tagged values.

4. ADVANCED CODE GENERATION FROM ASPECTS

An improved aspect mapping procedure will first modify the UML model before code generation. This will greatly reduce the complexity of code generator templates. A prerequisite for this procedure is a UML tool with a repository that is writable from code generator scripts or templates. This seems to become available in new tool developments.

In larger systems, many model elements will be affected by several aspects, e.g. “authorization” and “persistence”. In general, the order of applying these is meaningful. From our experiences, it is sufficient to determine the order on a per package base.

Thus the process of aspect modeling for code generation has the following steps:

- Build UML models for the structural parts of all aspects.
- Build code generator templates for non coding parts of all aspects.
- Mark all model elements in client packages that are affected by aspects.
- Translate the models from (1.) into code that modifies the model. After gaining some experience with modeling aspects in UML, this might be automated.

5. Determine the order of applying multiple aspects.
6. Run model modification and subsequent code generation from the modified model.
7. Verify, test, debug and iterate.

5. CONCLUSIONS

Separation of concerns in UML models using an appropriate package structure is well established [1, 2, 4]. Most of these packages introduce aspects that will crosscut code. Explicitly modeling structural parts of aspects in UML and coding parts as code generator templates improves quality and completeness of code generation. The code generator weaves the aspects.

6. REFERENCES

- [1] Frankel, Mike, and Winant, Becky. A Taxonomy for Domain Partitioning & Reuse. Object Magazine, 3, 1998
- [2] Frankel, Mike. Bridging Techniques for Maximum Domain Model Reuse. White Paper, Esprit Systems Consulting, Inc., 1997
- [3] Huber, Stefan, and Seidl, Heinz G. Architecture Component Development -generating more code from your UML models. White Paper, Aonix, Inc. San Diego, CA, 2000, www.aonix.com/content/downloads/stp/ACD1.pdf
- [4] Shlaer, Sally, and Mellor, Stephen J. Object Lifecycles. Modeling the World in States. Yourdon Press, Englewood Cliffs, NJ, 1992

Appendix A. Template Code Snippet for Aspect “Model Inferred Drag&Drop”, C++ Target

This code is written for the Aonix ACD code generator [3]. For illustration, lines that will appear in generated code are printed **bold** while navigation in the model is printed in *italics*.

```

////////////////////////////////////
/*      generate test and accept functions for drag and drop of operations
      see implementation for semantic details.
*/
template genDragDropDecls(MClass)
    //--helper functions for drag&drop-----
    virtual bool canAcceptDragDrop(int, int, bool = false);
    virtual bool acceptDragDrop(int, int, bool = false);
    virtual bool unplugDragDrop();
    virtual bool aggregationCycleDragDrop(int, int);
end template

////////////////////////////////////
/*      generate test and accept functions for drag and drop of operations
      a drag object can be accepted (dropped) under the following default conditions:
      -   dragged object isKindOf a class that is linked by a aggregation association
      -   dragged object is accepted by the owner of this object (i.e. an aggregate
          containing this object)
*/
template genDragDropFunctions(MClass)

//--test if drag&drop object could be accepted-----
bool [MClass.name]::canAcceptDragDrop(int cid, int oid, bool clone)
{
    [udOut("UDDDT", "", "Test drag and drop")]
    if( ! clone && myMetaClass()->isA(cid) && getKey() == oid) return false; // cannot drop on self
    if( ! clone && aggregationCycleDragDrop(cid, oid)) return false; // avoid cyclic aggregations
    [loop(Instances->MClass([MClass.id] " " getBaseClassList([MClass])) as Super)]
        [loop(Super->Role as FromRole->MAssociation->MAssociationEnd as ToRole->MClass as Partner
            Where [FromRole.id] != [ToRole.id])]
            [if([FromRole.aggregation] == "Aggregation")]
                if(DbMetaClass<[Partner.name]>::metaInstance()->isKindOf(cid)) return true;
            [else]
                [if([ToRole.aggregation] == "Aggregation")]
                    [insert("include",[Partner.name])]
                {
                    Ptr<[Partner.name]> owner = to_[ToRole.name].navigate(this).first();
                    if(owner.valid() && owner->canAcceptDragDrop(cid, oid)) return true;
                }
            [end if]
        [end if]
    [end loop]
    return false;
}

```

... 80 more lines for three more function bodies