Designing with Inheritance and Composition

Omar Alam
School of Computer Science, McGill University,
Montreal, QC H3A 2A7, Canada
Omar.Alam@mail.mcgill.ca

Jörg Kienzle
School of Computer Science, McGill University,
Montreal, QC H3A 2A7, Canada
Joerg.Kienzle@mcgill.ca

Abstract

Inheritance and composition are two different techniques that allow a modeller to extend the properties of a class. In this paper we highlight the differences of these two closely-related concepts when used in aspect-oriented designs. In particular, we explain that when an aspect wants to extend a base class of a source model, the designer should choose to use composition if she intends the extension to *replace* the base class. If she intends to define an *alternative* to the base class with extended functionality, inheritance should be used. We demonstrate the power of the combined use of both techniques by showing an aspect-oriented design of parts of a workflow middleware product line.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design; I.6.5 [*Simulation and Modeling*]: Model Development

Keywords aspect-orientation, inheritance, composition

1. Introduction and Motivation

Inheritance, or generalization-specialization as defined by the UML [6], is a well-known concept of object-orientation (OO) that makes it possible to share structural and behavioural properties among objects. Concretely, common structure and behaviour is defined in what is called a *superclass*. Any *subclasses* that inherit from it also have the same structural properties. The behavioural properties can also be shared by subclasses, if the subclasses are behavioural subtypes of the superclass [4]. Inheritance as defined by OO does, however, not guarantee that. The discussion in this paper therefore focusses mainly on the shared structural properties.

Aspect-orientation (AO) is a new modularization paradigm that focusses on identification, separation and composition of crosscutting structural and behavioural concerns. AO techniques also make it possible to define within a module, usually called an *aspect*, structural and behavioural properties shared among objects: when the aspect is woven with the rest of the application, the structure and behaviour defined in the aspect appears in all the places where the aspect is applied. Structural weaving is usually performed by composing/merging classes (or partial classes). For instance, in the aspect-oriented programming language AspectJ [2], inter type declarations can be used to define fields and methods in an aspect that are then merged into a base class when the aspect is applied. At the modelling level, France et al.'s class composition technique [7] or UML's package merge [5] make it possible to take two classes defined in separate source models and combine them into one class in the target model, retaining the fields and methods from both source classes. From now on, we refer to the AO way of extending structural properties as *composition*.

At first glance it might seem that having two ways of sharing structural properties, inheritance and composition, is redundant. On the other hand, inheritance and composition are two *different mechanisms* that make it possible to *extend* structural elements with additional properties. To the best of our knowledge, the subtle differences between the two techniques have not been discussed in the AO literature. The contributors of UML that defined UML package merge are also vague when talking about the merge operation and how it compares to generalization:

UML infrastructure, v. 2.4.1, page 163 [5]: "It (i.e. the *Merge* operation) is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both. ... Most often it is used to provide different definitions of a given concept for different purposes, starting from a common base definition. A given base concept is extended in increments, with each increment defined in a separate merged package."

In the past years we have worked on several AO case studies in which we created structural models of significant size that used both inheritance and composition. Some case stud-

VariComp'12, March 26, 2012, Potsdam, Germany.

ies involved more than 30 aspects. In this paper we describe the semantic differences between the two techniques, and why we need both mechanisms for structural modelling. In particular, we will give guidelines that allow a designer of an aspect to decide whether to use inheritance or composition.

The remainder of the paper is structured as follows: Section 2 presents the semantic difference between structural inheritance and structural composition, and the design consequences that result from choosing one over the other. Section 3 illustrates the power of combining the two mechanisms by showing parts of the structural design of a product line of workflow execution engines, and the last section draws some conclusions.

2. Inheritance and Composition

In this paper we are going to illustrate our ideas using the Reusable Aspect Models (RAM) approach [3]. RAM allows a modeller to describe the structure and behaviour of a design concern using class diagrams, state diagrams and sequence diagrams. Since this paper focusses on structural modelling, all example models presented in the paper just show the structural view of RAM models.

2.1 Composition in RAM

In RAM, when an aspect B needs properties defined by some other aspect A, then B can compose with A (or in AO terms, A can be woven into B). This creates a dependency between the two aspects: B depends on A. In a sense, A is a "low-level" aspect, since it provides general structure and behaviour, useful on its own and also in the context of B. B is a "higher-level" aspect, since it provides more specific structure and behaviour based on A. Of course, the functionality of B can itself be used within another aspect C to provide even more specialized functionality.

RAM supports the creation of such aspect hierarchies. Every aspect has a well-defined aspect interface that comprises the public model elements, i.e., the structural and behavioural properties that the aspect exposes to the rest of the model, as well as the mandatory instantiation parameters. The mandatory instantiation parameters designate the model elements that are only partially defined, i.e., the elements that need to be composed with other model elements when an aspect is applied.

The class diagrams in the structural views of RAM aspects are composed based on France et al.'s composition technique [7]. The composition directives, or *instantiations* as they are called in RAM, tell the weaver which model elements (i.e., classes, associations, methods and parameters) are to be composed. In RAM, the instantiation directives are located in the higher level aspect. In our example, A specifies which model elements of B are to be merged with which elements of A. Using the directives, the weaver generates an independent aspect model of A that contains all model elements of B. In addition to performing the composition according to the instantiation directives, the RAM weaver also applies automated information hiding: all public elements of B are switched to intra-aspect visibility, unless A explicitly re-exposes them in its interface. As a result, none of the model elements of B are visible to the outside anymore, unless otherwise specified.

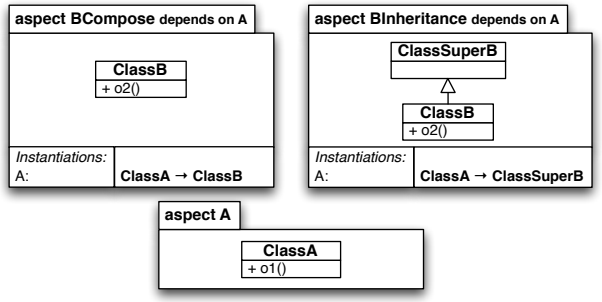

Figure 1. Composition vs. Inheritance

2.2 Deciding between Composition or Inheritance

Often, a RAM designer wants to extend the properties of a class defined in a low level aspect within the design of a higher level aspect. There are two design options that can achieve this: using composition with inheritance, or using composition only. Although at first glance both techniques achieve the same result, this design choice has considerable consequences later on, i.e., for aspects that are higher up in the hierarchy.

Fig. 1 illustrates the two choices by showing two ways of designing an aspect B that depends on a lower-level aspect A. The higher level aspects, *BCompose* (on the left) and *BInheritance* (on the right), both define a class `ClassB` that extend the properties of `ClassA` in aspect *A*. In *BCompose*, the instantiation directives directly merge `ClassA` defined in *A* with `ClassB`. Aspect *BInheritance* on the other hand defines a super class `ClassSuperB`, and merges that class with `ClassA`. `ClassB` in *BInheritance* is modelled as a subclass of `ClassSuperB`. In both cases, the resulting class `ClassB` has the properties of `ClassA`, i.e., operation `o1()`, and is extended with an additional operation `o2()`.

There is, however, a significant difference between the two designs. In *BCompose*, the entity `ClassA` does no longer exist. Its properties were absorbed into `ClassB`. By using composition only, the designer decides that in the context of B, only `ClassB`, i.e., a specific extension of `ClassA`, is of value. Users of *BCompose* should not be able to create instances of an entity that only has the properties defined by `ClassA`. Actually, once the weaver has generated an independent aspect model for *BCompose*, the fact that some of the properties of `ClassB` were once modularized within a separate entity is not visible anymore. In essence, the designer of *BCompose* declares `ClassB` to be a (more specific) *replacement* of `ClassA`. If one thinks in terms of extension,

this means that all instances of ClassA are extended to have the additional properties specified in ClassB.

The situation is different in *BInheritance*. As specified in the instantiation directives, the entity ClassA provided by *A* still exists in the context of *B* and is called ClassSuperB. Additionally, the entity ClassB is defined. ClassB is more specific, because it inherits the properties of ClassSuperB (and therefore ClassA) and defines additional properties. By specifying two classes that are related through inheritance the designer decides that in the context of *B* two kinds of objects are useful: instances of the general ClassSuperB and instances of ClassB. In essence, ClassB provides an *alternative* functionality to ClassSuperB. If one thinks in terms of extension, both ClassA (now named ClassSuperB) and its extension ClassB are available. Whenever an instance of an entity needs to be created, a user of *BInheritance* has the choice to either instantiate ClassSuperB or ClassB.

To summarize: to make the right design decision when extending a lower-level entity, the designer needs to determine if the newly designed entity provides an *alternative* to the lower-level one, or if it is a *replacement* of it. In the former case, the lower-level entity should be exposed as a super class and inheritance should be used to extend the new entity from it. In the latter case, only the new entity should be part of the new aspect, and composition should be used to add the properties of the low-level entity to it. As a result, all instances of the lower-level entity now have the additional properties of the entity in the higher-level aspect.

2.3 Consequences for Higher Levels

The immediate consequence for a higher level aspect is obviously that in the case of *BCompose* only the extended entity is available, whereas *BInheritance* offers the general entity and the extended entity.

Let us assume that only *BCompose* has been designed, and that a higher-level aspect *C* needs the functionality offered by ClassB. If C also needs the more general functionality provided by ClassA, it can simply depend on *A* and as a result declare instances of ClassA. However, in *C* ClassA and ClassB are not related in any way. The "ClassA'ness" of ClassB was lost as a result of the replacement design choice made by the designer of *BCompose*.

This situation is illustrated in aspect *C1* on the left side of Fig. 2: ClassB from aspect *BCompose* is merged with ClassC1, and ClassA from aspect *A* is merged with ClassC2. Although after the weaving the classes ClassC1 and ClassC2 both have the variable var1 and the operations o1() and o3(), they can not be treated in a uniform way. It is not possible, for instance, to use one reference type to refer to both ClassC1 and ClassC2 instances, or to use polymorphism.

The situation is different for aspect *C2* on the right side of Fig. 2 that depends on aspect *BInheritance*. In aspect *C2*, ClassC and ClassB both have the properties var1 and o1()

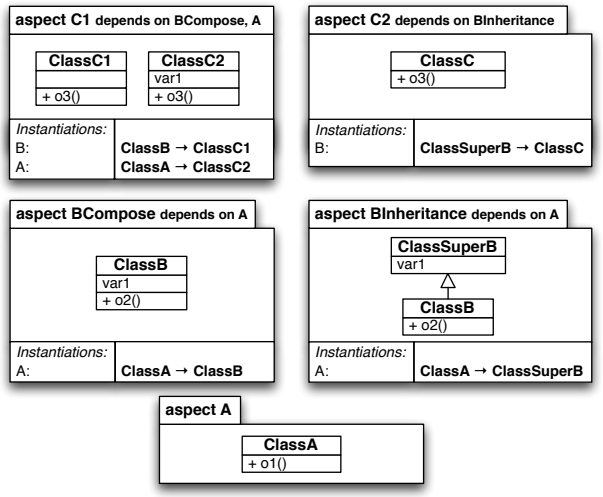

Figure 2. Design Choice Consequences

and o3(), and can both be referenced by a ClassC reference, since ClassB is a subclass of ClassC.

3. Real-World Example: Workflows

A workflow is a depiction of a set of operations that need to be completed in a certain order to fulfill a certain goal or task. For example, workflows have been used in software development to describe how a system under development is supposed to interact with its environment. A well-known example modelling formalism that can be used to describe workflows in general is UML Activity Diagrams [6]. Another example is the User Requirements Notation (URN) [1], a visual language standardized by the International Telecommunications Union intended for modelling interaction scenarios between a system under development and its environment. In order to be able to execute workflows defined in URN, we have worked on the definition of a URN workflow execution environment. We have elaborated an aspect-oriented design model of a workflow middleware that provides the user with the functionality to define URN workflows, to instantiate them and finally execute them. In this section we show the structural views of some interesting aspect models of this workflow middleware, namely the aspects *WorkFlow*, *ParallelExecution*, *OutPath*, *ConditionalExecution*, *InPath*, *Synchronization* and *Stub*, and point out for each aspect why they were designed using inheritance and composition or composition only.

The top left aspect in Figure 3 shows the *WorkFlow* aspect which defines the minimal model elements found in a work flow. It states that a work flow is composed of nodes, which can be sequence or control flow nodes, one of which is the end node. SequenceNode and ControlFlowNode are *alternatives* of WorkFlowNode, clearly shown by the inheritance relationship, and similarly EndNode is an *alternative* of ControlFlowNode. Both SequenceNode and ControlFlowNode do not substitute the need of WorkFlow-

Node. The WorkFlowNode has two abstract methods, depositToken() and addNextNode(WorkFlowNode n), which are implemented differently by the two subclasses. This allows other parts of the system to treat workflow nodes in a uniform way. For example, a potential work flow execution engine can deposit a token into any kind of node whether it is a SequenceNode or a ControlFlowNode.

Some nodes in a workflow are not just connected to one following node. Figure 3 shows a high-level aspect called *OutPath* that depends on the *WorkFlow* aspect. It defines the structure needed for control flow nodes with more than one named outgoing path in the class |CFNWithOutPath. Here, |CFNWithOutPath is an *alternative* of ControlFlowNode because the designer does not want all control flow nodes to have outpaths. Similarly, the designer decides that OutPathNode is an extenstion of SequenceNode and hence does not replace it, but offers OutPathNode as an *alternative* to SequenceNode. Since CFNWithOutPath is not a complete workflow node, the designer specifies that this alternative must be completed in a higher level aspect, indicated by the additional "|" [3]. Finally, the instantiation directives ensure that ControlFlowNode and SequenceNode are composed with ControlFlowNode and SequenceNode in the *WorkFlow* aspect.

The *ParallelExecution* aspect shown in Figure 3 defines a contol flow node that allows a workflow to continue execution of several following nodes in parallel. It is an example that shows how the designer can make use of the *OutPath* aspect to define a concrete control flow node that has outpaths. To define a parallel execution control flow node, the designer declares a ParallelExecutionNode and composes it with the partial |CFNWithOutPath class of *OutPath*. As a result, ParallelExecutionNode replaces |CFNWithOutPath. The fact that ParallelExecutionNode is an alternative ControlFlowNode is already defined in *OutPath*.

The *ConditionalExecution* aspect shows how the designer can make use of the *OutPath* aspect again to define another alternative control flow node that represents conditional execution. The instantiation directives specify that |CFNWithOutPath is composed with ConditionalExecutionNode and OutPathNode is composed with OutPathNodeWithCondition. ConditionalExecutionNode is not an alternative to |CFNWithOutPath, but rather it is a replacement. On the other hand, OutPathWithCondition is an *alternative* of OutPathNode. As a result it is now possible for out path nodes that follow a conditional node to be associated with a Condition object.

Figure 3 shows the aspect *InPath*, that, similarly to the *OutPath* aspect, extends control flow nodes to be able to have more than one incoming path. Its structural design is similar to *OutPath*. Again, |CFNWithInPath is designed as an *alternative* of *ControlFlowNode*, and made partial to force higher level aspects to complete it further. What is new here is that *InPath* needs to extend the functionality of *all* work flow nodes in order to allow them to be connected to control nodes with inpaths. This is done by defining the class InPathConnectibleWorkFlowNode with the method addNextNode, and by composing it with the WorkFlowNode class of the lower level WorkFlow aspect. InPathConnectibleWorkFlowNode therefore *replaces* WorkFlowNode in the aspect *WorkFlow*, because all work flow nodes, once control flows with inpaths are used, need to be connectible to inpaths.

The *Synchronization* aspect uses *InPath* to define control flow nodes that synchronize execution by composing |CFNWithInPath with SynchronizationNode.

A stub is a URN workflow element that allows workflows to be nested. A stub has several in and out ports, which are bound to start and end nodes in another workflow. When the flow of control enters the stub in the outer workflow, the flow of control continues in the inner workflow according to the binding.

The aspect *Stub*, also shown in Figure 3, depends on two lower level aspects, *InPath* and *OutPath*. Both |CFNWithInPath and |CFNWithOutPath are composed with the class StubNode. The StubNode *replaces both* CFNWithInPath and CFNWithOutPath and composes them together into one class. Notice that by composing these two subclasses of ControlFlowNode, StubNode also inherits from ControlFlowNode. As a result, aspects in a higher level that use StubNode will have access to the properties of control flow node, and properties provided by |CFNWithInPath and |CFNWithOutPath.

In some way the aspect-oriented design presented in this section represents a product line of workflow middlewares, because the user can simply ask the weaver to generate a specific workflow middleware by selecting the desired workflow aspects (i.e. the desired features).

In the bottom left half of Figure 3 (A), the final woven design model that the weaver generates when *all* aspects are selected is shown. Because we used alternatives in the aspects that designed the individual control flow nodes, a particular work flow can instantiate the subset of the nodes it needs. For example, one work flow can be composed of SequenceNode, EndNode, ConditionalExecutionNode and OutPathNode only, whereas another work flow can use a different subset, e.g. SequenceNode, EndNode, SynchronizationNode and InPathNode. However, if *all* workflows that the design needs to support do not use one of the nodes defined in an aspect, a new design model can be generated by selecting the desired subset of aspects. Figure 3 (B) illustrates a design in which only *Workflow* and *ConditionalExecution* (and indirectly also *OutPath*) were applied.

4. Conclusion

Inheritance and composition are two different techniques that allow a modeller to extend the properties of a class.

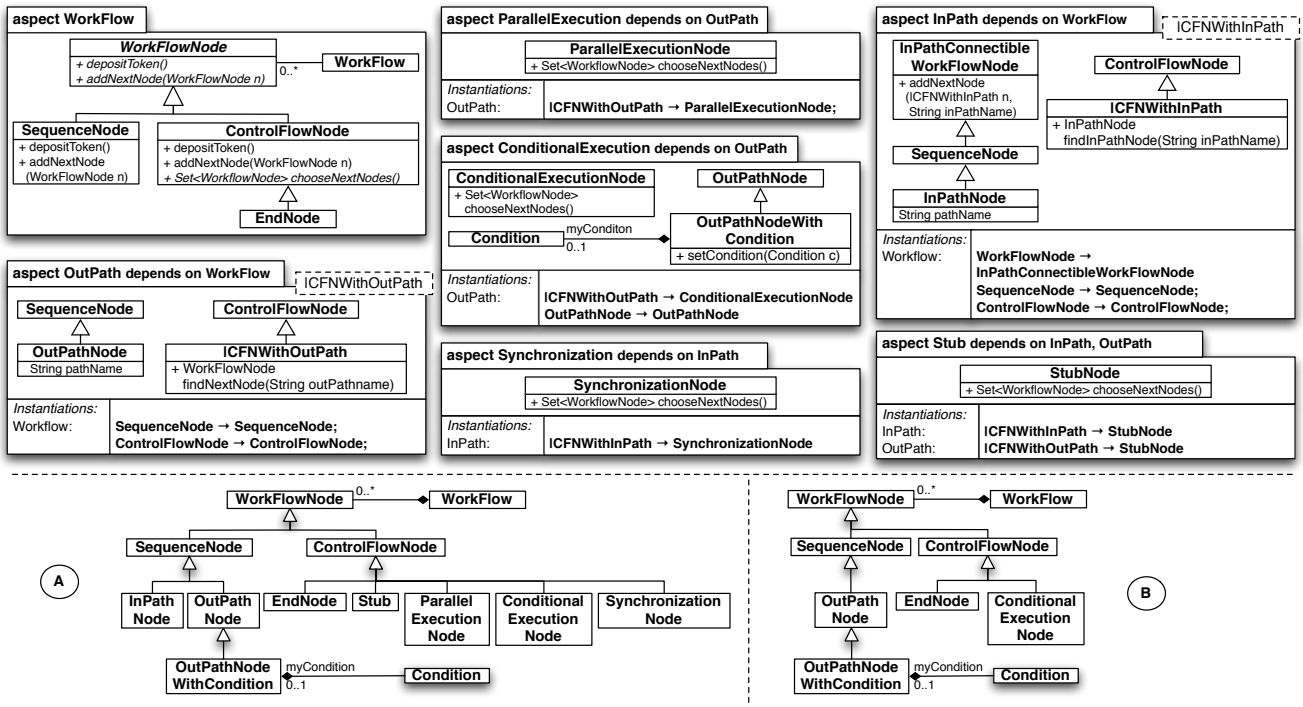

Figure 3. The Workflow Aspects and Two Generated Workflow Middleware Design Models

To the best of our knowledge, the aspect-oriented modelling community has so far not established well-defined guidelines on when to use inheritance, which is an object-oriented concept, and when to use composition, which is an aspect-oriented concept.

In this paper we highlighted the differences of these two closely-related concepts when used in aspect-oriented designs. In particular, we explained that when an aspect wants to extend a base class of a source model, the designer should choose to use composition if she intends the extension to *replace* the base class. As a result, *all instances of the class are always created with the extension*. If she intends to define an *alternative* to the base class with extended functionality, inheritance should be used. That way, the designer that applies the aspect can choose which class to instantiate, i.e., the base class or the extended one, depending on the context.

We have demonstrated the power of the combined use of both techniques by showing an aspect-oriented design of parts of a workflow middleware product line. A base workflow aspect that only allows to define sequential workflows was optionally extended with aspects that provide more elaborate control flow structures, i.e., conditional execution, parallel execution, synchronization, and nested workflows. To generate a design model supporting a specific set of workflow features, the weaver simply composes the aspect models defining the desired workflow elements.

References

[1] INTERNATIONAL TELECOMMUNICATION UNION (ITU-T). Recommendation Z.151 (11/08): User Requirements Notation (URN) - Language Definition, approved November 2008.

[2] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP'2001)* (2001), no. 2072 in Lecture Notes in Computer Science, Springer, pp. 327 – 357.

[3] KIENZLE, J., AL ABED, W., AND KLEIN, J. Aspect-Oriented Multi-View Modeling. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development - AOSD 2009, March 1 - 6, 2009* (March 2009), ACM Press, pp. 87 – 98.

[4] LISKOV, B., AND WING, J. M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 6 (Nov. 1994), 1811–1841.

[5] OBJECT MANAGEMENT GROUP. *Unified Modeling Language: Infrastructure (v2.4.1)*, December 2011.

[6] OBJECT MANAGEMENT GROUP. *Unified Modeling Language: Superstructure (v 2.4.1)*, December 2011.

[7] REDDY, Y. R., GHOSH, S., FRANCE, R. B., STRAW, G., BIEMAN, J. M., MCEACHEN, N., SONG, E., AND GEORG, G. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development I* (2006), 75–105.