

A Concept-Based Approach for Early Aspect Modelling

Dennis Wagelaar

Vrije Universiteit Brussel

Pleinlaan 2

B-1050 Brussels

dennis.wagelaar@vub.ac.be

ABSTRACT

Current aspect-oriented languages express aspects in many different ways. This diversity contributes to several problems when trying to model aspects in an early stage of the software lifecycle. This paper discusses a software modelling approach, called CoCompose, which supports aspect-oriented mechanisms without committing to one specific mechanism. A model made with CoCompose can be translated into several implementation languages using an automated process.

1. INTRODUCTION

Aspects or crosscutting concerns are expressed in many different ways by current aspect-oriented languages. A few examples are the join point mechanism used by AspectJ [1], the hyperspaces mechanism [2] and the composition filters mechanism [3]. One specific aspect-oriented mechanism may be more efficient to use than another depending on the application context.

This diversity in representing aspects contributes to several problems when trying to model aspects in an early stage of the software lifecycle. Some of these problems are discussed below.

- In current object-oriented software modelling approaches, the implementation semantics (e.g. method, class, aspect¹) of each software element must be determined when introducing the element, even though it may be hard to choose the exact semantics at that moment.

With the introduction of aspect-oriented constructs, the problem increases. In an early stage of the software lifecycle, it may not be possible to determine what the best mechanism is to implement an aspect. It may not even be possible to determine whether an element will be crosscutting (i.e. an aspect) or not.

For example, UML [4] and extensions to UML [5][6], represent software elements using a set of language elements with fixed implementation semantics (e.g. method, class). It is not possible to model software elements without determining their implementation semantics. When using a deterministic transformation from a UML design to an implementation, the possible implementations of a UML language construct will be narrowed down to a limited set of implementation constructs.

- If there are gaps in the process of automated code generation from software models, these may require manual transformations between software lifecycle stages. If, in addition, the software structures in different lifecycle stages differ, one local update in one stage may require numerous

updates in the other. The original local update is not well traceable in the other stage either.

For example, there are early stage aspect-oriented software modelling approaches, such as Cosmos [7], which do not commit to implementation constructs initially. Still, the transformation of software models described using such early stage approaches must be done manually.

This paper will focus on software modelling without unnecessarily committing to implementation constructs, including aspect implementation constructs. The following section will discuss an approach that attempts to deal with these problems. Section 3 will illustrate the tool support that is implemented for this approach. In section 4, a comparison is made with other modelling approaches. Section 5 discusses the conclusions on the proposed approach and points at directions for future work.

2. COCOMPOSE: A CONCEPT-BASED MODELLING APPROACH

CoCompose [8] is a software modelling approach that supports aspects or crosscutting concerns. The CoCompose modelling language introduces *concepts* as a central modelling element to overcome the problem of early commitment to implementation constructs. *Composites* are introduced as a reusable abstraction of design concepts, such as inheritance, superimposition or a specific design pattern. Instances of composites can be used to connect the concepts into a coherent model. These two language elements will be discussed in detail below.

2.1 Concepts

Concepts are the central language elements of the CoCompose modelling language. The generic notion of concepts prevents software developers to prematurely choose a specific implementation construct for the element to introduce. When translating into an implementation language, such as AspectJ, concepts can be mapped to implementation constructs, such as aspects, classes, instances, parameters, attributes, methods or pointcut descriptors.

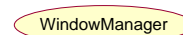


Figure 1. A concept.

Concepts can contain embedded implementations in order to avoid making each detail of the software system explicit as a concept (e.g. modelling expressions and statements as concepts).

¹ Meant as an AspectJ “aspect” construct.

These implementations are expressed in an implementation language to which the developer wants to map.

Concept implementations can have constraints to describe the extent of their applicability. The most important constraint is for which implementation construct the implementation holds. The concept depicted in Figure 1, for example, could have an implementation in Java as a class. This means that the implementation is only applicable when WindowManager will be implemented in Java as a class.

Other constraints can be imposed by the contents of the implementation. Consider the following implementation body of the above concept as a Java class:

```
public WindowManager(<System> syst) {
    this.system = syst;
}
```

A reference to another concept, System, is used within this implementation, denoted by the '<' and '>' braces. This implementation requires the System concept to be implemented as a Java class or interface. Embedded implementations and their constraints are stored with the concept in XML format.

2.2 Composites

Composites are reusable abstractions of design concepts, such as an association relation or an observer design pattern. They allow software developers to introduce new design concepts for use within concrete designs. Instances of composites can be applied to concepts as aspects to describe the relationships between them. Composites can also be used to model specific aspect-oriented mechanisms (e.g. aspect weaving, hyperslice composition, superimposition, composition filter).

Composites can contain *composite roles* and *published concepts* in their interface. Composite roles describe the role a concept plays in the relationship defined by the composite. Published concepts represent concepts exported by the composite.

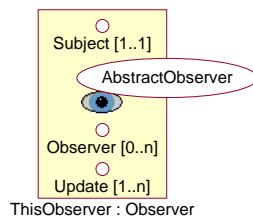


Figure 2. A composite.

The Observer composite depicted in Figure 2 represents the interface of an observer design pattern. It has three roles: Subject, Observer and Update. Concepts can be “wired” to a role to indicate that they fill that role. It also exports one concept: AbstractObserver. This concept, which is introduced by the composite, can be used outside the composite now.

Figure 3 depicts an example that is modelled in the CoCompose language. It contains an instance of the Observer composite applied as an aspect to four concepts. The Slider concept plays the role of a Subject. It can thus be observed by ColorPanel and Label, which play the Observer role. Update represents the

update operation, which should be executed on each Observer when the state of the Slider changes.

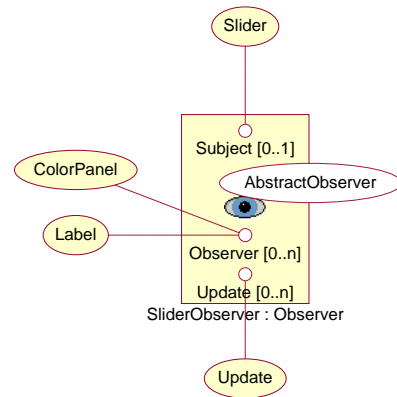


Figure 3. An example modelled in CoCompose.

The definition of a composite can contain embedded descriptions of the structure of the design concept represented by the composite. This structure, called *solution pattern*, again consists of concepts and composites. The solution pattern is partitioned in a default part and one role part for each role. When applying a solution pattern, the default part is applied once for the whole model and the role parts are applied once for each concept filling the corresponding role.

The solution pattern depicted in Figure 4 is a solution pattern for the Observer composite. The three roles, Subject, Observer and Update, are represented by the small coloured circles, called *solution roles*. The colours of the solution roles distinguish between the default part and the several role parts in the solution pattern. For example, the Notify concept has the same colour as the Update solution role. This means that Notify belongs to the Update role part. All the elements in the standard colour belong to the default part (i.e. HasObservers, Observers, Attach, etc.).

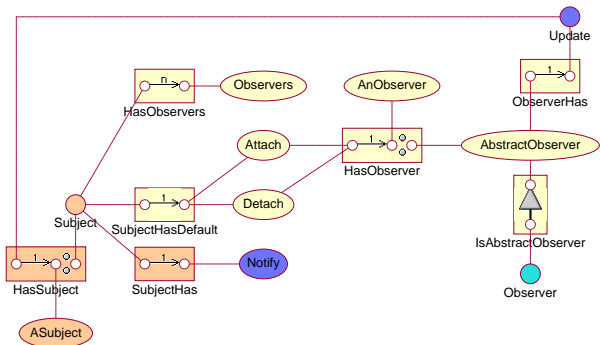


Figure 4. An Observer composite solution pattern.

Each composite’s definition can also contain *implementation generators*, which can implement the composite in a specific implementation language. An implementation generator also declares several role parts. These describe the implementation form constraints on the concepts filling the roles. An implementation generator will usually work for a limited combination of implementation forms. The role parts with their

implementation form constraints are expressed in XML format. The actual implementation generator functionality is expressed in Java.

The `IsAbstractObserver` depicted in Figure 4 is an instance of an Inheritance composite. Its implementation generator for Java, for example, should work for object classes but not for methods. Furthermore, when the child role is filled by a concept implemented as an interface, the parent role cannot be filled by a class. These implementation form constraints are described in the role parts.

By being able to define several solution patterns and implementation generators for one composite, the composite isn't tied down to a single implementation. It represents a high-level element that can be implemented in several ways.

2.3 Transformation and code generation

In order to close gaps in automated code generation, an automated process has been developed for translating CoCompose models into specific implementation languages. This process is based on Design Algebra [9]. It consists of three steps, which will be briefly explained below. A more detailed explanation can be found in [8].

The first step is concerned with flattening the model: eliminating composites through the application of their solution patterns. Since each composite can have several optional solution patterns, this will result in a set of models. These models represent each feasible combination of solution patterns to be applied. Each of these flattened models should only have composites left with implementation generators for the chosen implementation language.

Figure 5 shows the example model after flattening. Note that only one model results in case there is only one solution pattern. If other solution patterns were used, e.g. optimised for aspects, hyperslices or composition filters, one flattened model would result for each solution pattern.

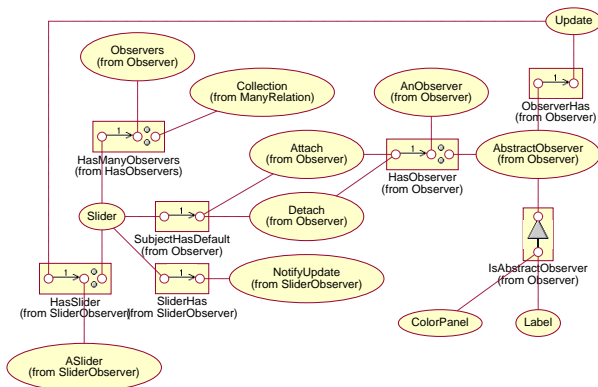


Figure 5. The example after flattening.

The next step determines the implementation form (e.g. class, method) of each concept. The possible forms for implementing a concept are determined by (1) the form constraints of implementation generators and (2) the available embedded implementations of that concept and (3) the form constraints on the chosen embedded implementations for other concepts.

Concepts will often have multiple possible implementation forms, while only one is needed. An implementation form priority heuristic is applied to eliminate all but the "best" combination of implementation forms according to the heuristic.

The last step generates the code for the transformed model. First, a skeleton implementation is generated, based on the chosen implementation forms of all concepts. Any embedded implementations of the concepts are inserted. Then, the implementation generators for each composite are applied.

3. TOOL SUPPORT

CoCompose has prototype tool support written in Java. A screenshot of the tool is depicted in Figure 6.

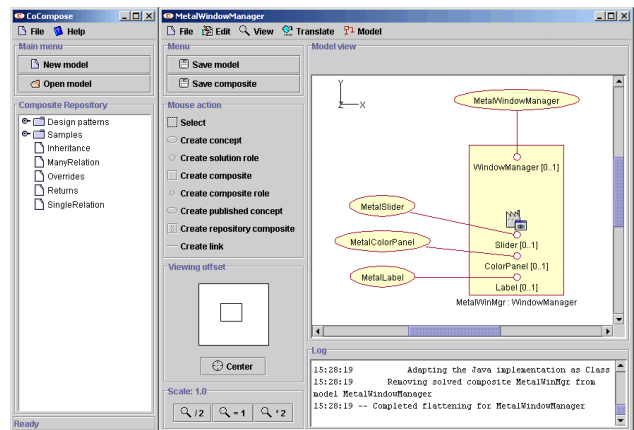


Figure 6. The CoCompose tool.

The tool supports modelling in the visual language and automated translation into an implementation language. It can also manage a repository of composites for reuse in multiple diagrams.

Currently, the tool can generate implementations in two programming languages: Java and ConcernJ [9], an extension to Java with support for composition filters. Support for implementation languages is organised in plugins, which allows for easy addition of support for other implementation languages.

4. RELATED WORK

This section discusses a selection of modelling approaches, which are related to CoCompose and compares them. Related transformation approaches are not included to keep the discussion focused on what to transform first.

4.1 Cosmos

Cosmos is a general-purpose concern-space modelling schema. It includes three types of elements: concerns, relationships and predicates. Concerns are categorised as logical or physical concerns, which again are divided in subtypes of logical or physical concerns. The relationships are also divided into categories and subcategories. Predicates are an exception in that they are left open for the developer to define and categorise.

When modelling a software system in Cosmos, a developer identifies and categorises concerns and relationships and defines predicates. An important difference between Cosmos and CoCompose is that CoCompose does not further categorise its

concepts. CoCompose does allow concepts to be grouped and hidden within solution patterns of composites and composites to be categorised by type. These categorisations are however left open for the developer to determine. CoCompose uses Design Algebra to determine implementation constructs, whereas Cosmos uses categorisation. Another important difference is that Cosmos is meant to supplement existing design approaches. As such, it doesn't define a direct mapping to an implementation. CoCompose defines its own transformation to an implementation.

4.2 Theme/UML

Theme/UML [5] is an aspect-oriented extension to the UML modelling language. It uses composition patterns and subjects to describe aspects. Both of these mechanisms are based on the UML package construct. Subjects can be combined with each other through binding of elements from either subject as mutual elements. Composition patterns can be used by filling the parameters with actual elements. Thus, the composition pattern is "applied" to concrete elements.

Clarke and Walker also illustrate how Theme/UML models can be implemented. AspectJ is used as an example implementation language. They show how the design structure can be preserved in the implementation by separating the aspect and its composition specification (the pointcut).

The composition pattern construct used in Theme/UML is very similar to CoCompose composites with a solution pattern. Composites can also be parameterised by concrete elements. Composites can also use published concepts to make elements available for binding them with other elements. The binding is done through wiring elements to each other. An important difference lies in the fact that Theme/UML uses UML constructs as parameters for binding. CoCompose uses the generic notion of concepts to specify roles and published concepts within composites. Another significant difference is that Theme/UML links a single UML structure to each subject or composition pattern. CoCompose allows for the definition of alternative solution patterns; the exact solution pattern structures are encapsulated in the composite.

4.3 Feature modelling

In the context of generative programming [11], feature models are used to model a family of systems instead of a single software system. This is done through specifying features and how features can be combined. Features can be mandatory or optional, depending on the presence of other features.

An analogy of features can be made to CoCompose composite roles. Whether or not a role is filled determines whether a certain feature of the composite is generated. The alternative ways to implement a composite represent a set of features of which one must be chosen.

5. CONCLUSIONS & DISCUSSION

The modelling language proposed in this paper allows for modelling of software elements – including aspects – without committing to implementation semantics. Many other early-stage modelling approaches choose to narrow down the possible implementation semantics of an element on introduction of that

element. In a later stage, the chosen implementation semantics may no longer suffice and refactoring [12] must be done.

While modelling in less abstract language constructs may be more intuitive, the lack of abstraction diminishes reuse. Intuitiveness can be preserved in CoCompose by the ability to define solution patterns on several abstraction levels. Higher-level solution patterns can be abstract and reusable for many implementations. Lower-level solution patterns can use composites that represent implementation constructs (e.g. inheritance, pointcut or composition filter).

Even though CoCompose uses highly abstract language elements, an implementation can be generated automatically. This improves maintainability, since updates need only be done locally. Note that the transformation process in its current form generates implementation alternatives of the system as a whole. This means it is hard to scale up the transformation process to large software systems.

My position is that (1) by allowing for such abstract language elements as concepts and (2) abstracting from implementation structures using alternative solution patterns, one can do factoring instead of re-factoring. The original model of the software system will be more stable, since many design and implementation decisions are postponed until the model is factored (i.e. transformed) into an implementation.

6. FUTURE WORK

The abstraction mechanism comprised by composites and alternative solution patterns has similarities to software components. In component-based development, the component interface is also separated from its implementation. Research can be done on whether the composite construct forms a good basis for describing a component in an early stage: "blue prints" of components. Also, the transformation process may benefit from incrementally generating alternative implementations and, as such, become more scalable.

Currently, generating an implementation from a CoCompose model still requires manual choices when alternative solution patterns exist. It is expected that assigning quality attributes to solution patterns should make automatic choices between alternatives possible. Fuzzy logic in combination with heuristics [13] can be used for automating these choices.

A relationship with feature modelling has already been discussed briefly. Research can be done on how to integrate feature modelling and the ideas of CoCompose.

CoCompose does not express sequence. There exist approaches for expressing sequence and interaction, which are also based on roles as placeholders in generic patterns [14][15]. Research can be done on how to integrate the ideas of these approaches and the ideas of CoCompose.

7. ACKNOWLEDGEMENTS

For earlier contributions on CoCompose, I would like to thank Lodewijk Bergmans. Furthermore, I would like to thank Viviane Jonckers, Wim Vanderperren and Bart Wydaeghe for reviewing this paper and their discussions on the subject.

8. REFERENCES

- [1] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W., An Overview of AspectJ, Proceedings of ECOOP 2001, LNCS 2072, Springer-Verlag, Berlin, 2001.
- [2] Ossher, H., Tarr, P., Multi-Dimensional Separation of Concerns and the Hyperspace Approach, in Aksit, M. (ed.), Software Architectures and Component Technology, Kluwer Academic Publishers, Dordrecht, 2002.
- [3] Bergmans, L., Akşit, M., Tekinerdogan, B., Aspect Composition using Composition Filters, in Aksit, M. (ed.), Software Architectures and Component Technology, Kluwer Academic Publishers, Dordrecht, 2002.
- [4] OMG, Unified Modeling Language Specification, Version 1.4, September 2001.
- [5] Clarke, S., Walker, R., Towards a Standard Design Language for AOSD, in Kiczales, G. (ed.), Proceedings of the 1st International Conference on Aspect-Oriented Software Development – AOSD 2002, ACM Press, New York, 2002.
- [6] Stein, D., Hanenberg, S., Unland, R., A UML-based Aspect-Oriented Design Notation for AspectJ, in Kiczales, G. (ed.), Proceedings of the 1st International Conference on Aspect-Oriented Software Development – AOSD 2002, ACM Press, New York, 2002.
- [7] Sutton Jr., S., Rouvellou, I., Modeling of Software Concerns in Cosmos, in Kiczales, G. (ed.), Proceedings of the 1st International Conference on Aspect-Oriented Software Development – AOSD 2002, ACM Press, New York, 2002.
- [8] Wagelaar, D., A Concept-Based Approach to Aspect-Oriented Software Design, MSc. Thesis, University of Twente, Enschede, The Netherlands, August 2002.
- [9] Salinas, P., Adding Systemic Crosscutting and Superimposition to Composition Filters, EMOOSE MSc. thesis, Vrije Universiteit Brussel, 2001.
- [10] Tekinerdoğan, B., Akşit, M., Synthesis Based Software Architecture Design, in Akşit, M. (ed.), Software Architectures and Component Technology, Kluwer Academic Publishers, Dordrecht, 2002.
- [11] Czarnecki, K., Eisenecker, U., Generative Programming: Methods, Tools, and Applications, Addison Wesley, Reading, Massachusetts, 2000.
- [12] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., Refactoring: Improving the Design of Existing Code, Addison Wesley, Reading, Massachusetts, 1999.
- [13] Akşit, M., Marcelloni, F., Deferring Elimination of Design Alternatives in Object-Oriented Methods, Concurrency and Computation: Practice and Experience, Vol. 13, pp. 1247-1279, Wiley, December 2001.
- [14] Wydaeghe, B., Vanderperren, W., Visual Component Composition Using Composition Patterns, in Proceedings of Tools 2001, Santa Barbara, July 2001.
- [15] Vanderperren, W., Applying aspect-oriented programming ideas in a component-based context: Composition Adapters, in Proceedings of GCSE 2001, Erfurt, September 2001.