Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 163 (2007) 19–28
www.elsevier.com/locate/entcs

On the Dominance of Decompositions in Models and their Aspect-Oriented Implementations

Tommi Mikkonen[1]

Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland

Abstract

Aspect-oriented approaches have commonly advocated separation of concerns. Some approaches have applied this separation in a symmetric fashion, like Hyper/J, whereas some others have relied on asymmetric separation, like AspectJ. The difference in the approaches is that the different concerns play a symmetric role in the former, whereas the latter explicitly includes a conventional implementation on top of which other concerns are woven onto as aspects. The question then arises, how are the concerns of the conventional implementation special in the latter, and will the opportunity to use symmetric separation lead to a fundamentally different decomposition. In this paper, we discuss the dominance in decompositions in specifications and corresponding aspect-oriented implementations. As examples, we use the specification method DisCo which allows modeling of concerns in a fashion that separates the different concerns to specification branches, and aspect-oriented implementations using Hyper/J and AspectJ that can be composed for DisCo specifications. As the final outcome, we propose that any aspect-oriented approach addressing the system at the level of program code necessarily has some concerns that are more dominant than some others due to the control flow of programs.

Keywords: Dominant decomposition, symmetric and asymmetric aspect-orientation

1 Introduction

Tyranny of dominant decompositions is the term introduced to address problems related to the inability to address all concerns of a software system using the same facilities. Reported in [17], this has lead to the introduction of a paradigm where all the concerns can be treated in a similar fashion, with a practical programming-level implementation in Hyper/J [20]. Then, all concerns can be treated symmetrically, which enables the creation of systems so that both conventional modularity and cross-cutting properties are enabled using hypermodules.

[1] Email: tommi.mikkonen@tut.fi

doi:10.1016/j.entcs.2006.10.013

In contrast to symmetric approaches to manage cross-cutting properties, asymmetric approaches have been introduced. For instance, AspectJ [18] introduces facilities for augmenting a baseline implementation with additions referred to as aspects. Then, one can advance such that an existing system, given as a conventional program, is taken as the starting point, and new behaviors are woven into the system with aspects. Furthermore, provided with a convention where aspects are used for certain issues, the developers can anticipate the injection of aspects, and overlook such parts of the system in the baseline implementation.

In general, the relation between symmetric and asymmetric approaches to aspect-orientation [6,7,4] has been an interesting topic of research (e.g. [9]). In this paper, we discuss the dominance of decompositions in terms of a specification, where one type of separation of concerns is provided, and sketch aspect-oriented implementations for it using Hyper/J and AspectJ. The purpose is to address differences and commonalities of the techniques, and to compare their different properties to each other as well as the underlying specification. Towards the end of the paper, we connect our results to the framework provided by model-driven architecture, MDA [8,21] by concluding that implementation-level techniques essentially require a dominant decomposition for executability and meaningful binding to control flow. The experiences are based on a case study, where the behavior of a mobile switch has been re-engineered [10].

The rest of this paper is structured as follows. Section 2 introduces the specification and modeling method we use as the starting point. Furthermore, we discuss the structure of specifications that has been commonly used for separating concerns. Sections 3 and 4 sketch implementations for DisCo specifications using Hyper/J and AspectJ, where symmetric and asymmetric decompositions of concerns are offered. In addition, we discuss the dominance of decompositions in described systems. Then, Section 5 finally concludes the paper.

2 Specification

As the method of specification in this paper, we use DisCo [12,19], a formal method that is based on Temporal Logic of Actions [13], and allows the development of specifications and models in a fashion where different concerns are incrementally introduced.

2.1 DisCo Principles

DisCo specifications are composed in terms of layers that contain classes and actions. Classes are containers of data, and actions can be understood as multi-object methods that can alter values of variables. Actions are executed in an interleaved fashion without any interference from the rest of the system and their execution is bound to be finished once it has been initiated, which makes actions atomic units of executions. The language used for composing specifications is textual. However, animation facilities have been provided to ease the analysis of specifications [16,3]. Furthermore, the relation of DisCo specifications and their denotation using UML

has been studied in [15].

Layers can build on top of other layers, which is referred to as refinement in DisCo terminology. A restriction is made that actions can only alter values of variables given in the same layer, which guarantees that safety properties will be preserved by construction. To satisfy this restriction, refinements can introduce new variables and operations on them as well as augmentations to actions, provided that the new action logically implies its ancestor. In fact, one can consider that the concept of ancestors plays a key role in the DisCo approach. All classes and actions can be considered as the refinements of their ancestors in earlier layers, with the layer defining an empty system as the origin of the hierarchy. This gives an explicit structure for any DisCo specification, with the opportunity to define concerns in individual layers.

When two or more layers are merged, actions may be merged into more complex ones or be kept apart from one another. This corresponds to weaving in the aspect-oriented setting, to which we will return in Sections 3 and 4, where aspect-oriented techniques are considered.

2.2 Concern-Based Modeling of Abstractions with DisCo

The use of layers allows modeling of systems using several levels of abstraction. For instance, when modeling a telephony exchange, it is possible to model the system using abstract concepts, such as call control, connections, and legs, which are individual connections from the exchange to a caller or callee, and charging, as well as in terms of processes used for implementing the abstract concepts. This allows each layer to focus on a certain concern the modeler wants to address separately.

As long as the relation between abstractions is simple, for instance each abstract concept can be associated with a certain low-level concept, this scheme is relatively simple. For instance, a low-level concept can be a process or interprocess communication. However, when an abstraction is defined that requires cooperation of several low-level concepts, more complex implementations result. In this paper, we will refer to these two types of abstractions as primitive and non-primitive, respectively. Another way to consider such abstractions is to refer to them as local and cross-cutting.

The refinement of a non-primitive abstraction to a directly implementable form is yet another concern. Therefore, this issue is commonly addressed using a DisCo layer that defines the relation between an abstract concept and its more concrete implementation. Techniques have been introduced for such transformations, including the option to use archived design steps that resemble design patterns [11].

2.3 Specification Architectures in DisCo

Due to the definition of the methodology, specifications given in DisCo contain two different types of architectures. One is composed in terms of classes and actions, and the other is composed with layers. The architecture composed with classes and actions resemble those composed with conventional techniques, but the architecture

composed with layers is less conventional. Therefore, this topic will be addressed in more detail in the following.

Layered specifications in DisCo allow layers where individual concerns are addressed. Layers are truly symmetric in the sense that the refinement relationship between layers preserves (safety) properties of all component layers, and the order in which layers are given can vary. Furthermore, refinements can only make more restrictions, which resembles the constraint-oriented design style introduced in connection with LOTOS [5]. Another difference to commonly used aspect-oriented approaches is that layers are complete in the sense that they only describe behaviors in terms of the variables included in them. Then, when layers are composed, a new universe is created where the rules of behavior of all component layers are satisfied.

Such layers allow an approach where the control of the system is given in one layer, which is then composed with other layers. When considering the architecture created with layers, a control layer can play two different roles.

- The use of primitive (or local) abstractions only allows an approach where fundamental properties are defined in the beginning, leading to a bottom-up approach.
- The use of non-primitive (or cross-cutting) abstractions requires an implementing layer, resulting in a top-down construction of the system. This layer can be given as a separate specification branch to promote separation of concerns.

These two approaches are illustrated using a simplified version of a telephony exchange as an example (Figure 1). There are three main functions in the exchange. Layer `Legs` introduces abstract concept legs, i.e., connections between the caller/callee and the exchange; layer `Processes` implements the abstract concept using available techniques, with different actions for incoming and outgoing calls and routing; and layer `Charging` introduces the capability to charge for established legs. As it is the relation of these functions that are important, we will only discuss them in an abstract fashion. Moreover, operations given in them have been included in the figure, as they are meaningful for implementations we will describe later on.

2.4 Implementing DisCo specifications with conventional techniques

When using conventional techniques, a one shot implementation aims at attacking the complete composition of all the layers, where the resulting architecture1 obeys the structure defined in layers, but overlooks the layer structure. In Figure 1, the bottom-up approach is straightforward to implement, but the top-down approach requires the creation of a composite specification shown in Figure 1. In other words, architecturally significant parts given inside the layers form the dominant decomposition, and the layered structure as a whole is overlooked [14].

More recently, we have also considered aspect-oriented techniques [1]. This enables us to preserve the structure created with layers. Moreover, we have considered applying the same design methodology in the design of aspect-oriented systems. In the following sections, we shift the focus on composing aspect-oriented implementations in a fashion that preserves the layered structure of the specification.

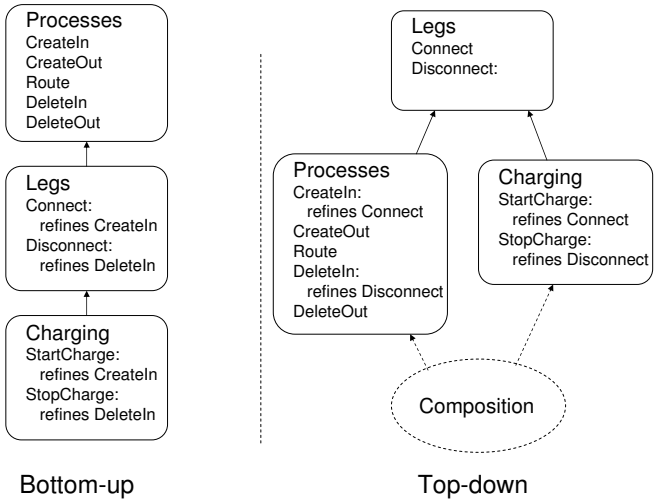

Fig. 1. Specification architecture of DisCo specifications

3 Hyper/J Implementation

Hyper/J [20] is probably the most prominent approach to symmetric aspect-oriented programming. It can intuitively be used for a similar separation of concerns as DisCo. However, the level of abstraction in the two approaches is different, which has some consequences on how systems can be constructed in a pragmatic fashion. In the following, we outline a mechanism for implementing the above DisCo specification using Hyper/J.

3.1 Source of Symmetry

When composing a Hyper/J system, independent subsystems are defined in the beginning. They can be individual classes or collections of them, and they can be even tested in isolation from one another. The goal is to create all the necessary operations of the eventual implementation in isolation.

Once a collection of implementation classes exist, the designer uses hypermodules to define how the different parts of the system are integrated. The means provided by Hyper/J basically require that two methods are equated, and enable the definition of call order of the methods. However, more complex facilities for relating the different classes can be imagined. Because the starting point for the development is a collection of classes and definitions that combine them, it is obvious that from the technical perspective all classes play a similar role.

3.2 Aligning Hyper/J Implementation with DisCo Specification

As the starting point of implementing a DisCo specification with Hyper/J, one should compose classes out of DisCo layers, reflecting the contents of the layers

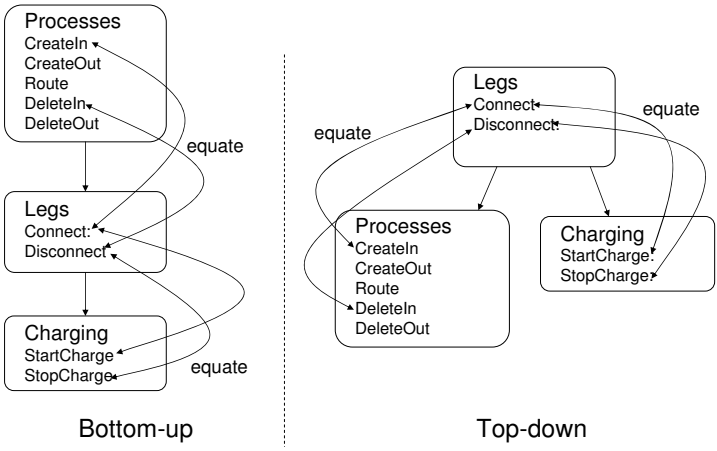

Fig. 2. Architecture of Hyper/J implementation preserving the structure

in detail. Then, once Java implementations for the contents are available, one should match the classes and equate the operations that correspond to each other in different classes. Sequential and branching architectures can be treated in a similar fashion, which makes Hyper/J a natural candidate for implementations of DisCo specifications. Moreover, equating the methods can be implemented in pieces, which allows an incremental approach to the development not unlike that of DisCo (Figure 2).

However, there is one major drawback. Before one includes the branch that defines the master control flow only declarative goals can be achieved. In other words, by combining branches that define operations that do not trigger themselves to execution, one cannot create any runnable programs. As a result, the semantics of different branches have different contribution to the development. Therefore, the branch defining the control flow can be considered as a dominant dimension in the semantic sense.

As a result of the special role of control flow, in cases where an early phase of the DisCo specification defines executions the situation is simple. The corresponding Hyper/J implementation can be given and tested in a straightforward fashion. Then, as the design advances, new features can be immediately augmented with normal routines. Furthermore, also testing by running the system is enabled. However, in cases where some later branch introduces the control, combinations where the implementation branch is missing result in declarative statements. This in particular applies to non-primitive abstractions, which often define no control as such but are conceptually important. They cannot be tested in a Hyper/J implementation independently of the rest of the system, but they can still be defined and used as intermediate systems that can be studied with reviews.

4 AspectJ Implementation

In contrast to Hyper/J, AspectJ [18] aims at the definition of systems in a fashion where a baseline implementation is given first. This baseline is given using conventional Java, and it will be extended with aspects that are woven into code. In other words, aspects can be taken as extensions of the baseline system, and they most naturally follow the control given in the baseline, although an option has been provided to override operations. In the following, we outline a mechanism for implementing the above DisCo specification using AspectJ.

4.1 Source of Asymmetry

The fundamental source of asymmetry in AspectJ is that there are two types of artifacts. One type is the conventional Java classes, and the other type is the aspects. The types also play a different role in the development process, which is addressed in the following.

When the development begins, a baseline implementation is created with conventional Java classes. This system can be tested and run normally. Then, once the baseline is satisfactory, aspects are woven to it to introduce new properties. However, unlike in Hyper/J where additional information was used to combine operations, aspects of AspectJ have information on where they should be woven. In fact, a common goal is to use aspects such that the baseline needs not to know about the use of aspects, because this would liberate the developers of the baseline to focus on its goals and requirements.

4.2 Aligning AspectJ Implementation with DisCo Specification

When composing an AspectJ implementation of a DisCo specification, the starting point is always an executable that essentially defines the master control flow. The definition can be given either in the main branch, or in some other branch, but we consider that this is the fundamental dominant decomposition of the system in the sense of its behaviors. Moreover, in order to create a runnable baseline it is an obvious necessity.

Once the baseline is completed, other features are integrated to it using aspects. In this case, non-primitive abstractions can become abstractions that implement the abstraction in one module, assuming that the control of the system is introduced in some later branch (Figure 3). This has already been addressed in detail in [1].

5 Discussion

Specification-level separation of concerns can be different from implementation-level separation of concerns. In this paper, behavioral dominant decomposition at a level of specification and its structural dominant decomposition in aspect-oriented systems were shown to have the potential to be different. The reason for the difference lies in the control flow, which can be abstracted away in a specification but

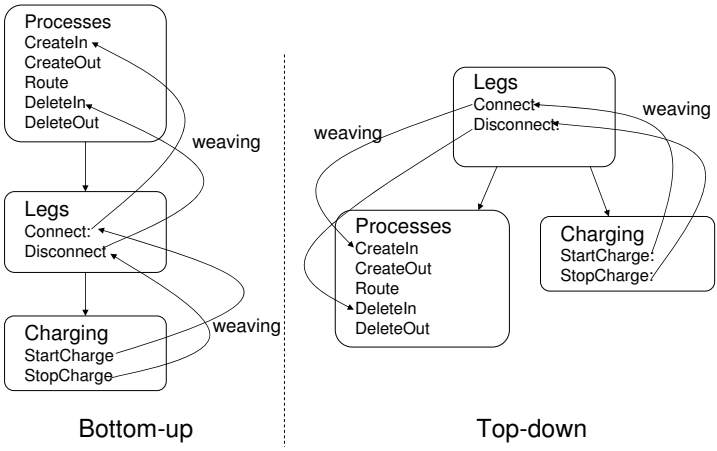

Fig. 3. Architecture of AspectJ implementation preserving the structure

forms a necessary element in an implementation. When considering the behaviors of the system, executions that must take place commonly introduce a dominant decomposition any case in implementations, as otherwise only declarative definitions can be given. As a result, structure-preserving implementations of the specifications, whose architecture had prescribed differences regarding the order in which certain concerns were introduced, are similar for both Hyper/J and AspectJ.

We believe that the reason for the difference lies in cross-cutting abstractions, which extend from one implementation-level abstraction to another. Without aspect-oriented techniques, such abstractions would result in tangled code, but with them the architecture of the composed implementation can be aligned with the specification. The rationale for the above is that non-primitive abstractions are cross-cutting, and can therefore benefit from aspect-oriented techniques for obvious reasons. However, without an executable main branch, it is difficult to locate the correct methods to equate or associated pointcuts. Yet they are an important tool when composing models, where the abstractions can be studied in a meaningful fashion. In particular, as already mentioned, they can be used for re-engineering the structure of systems into a form that easily lends itself to an aspect-oriented implementation, which has also given inspiration for the example used in this paper.

The consequences of the observation are many. Firstly, even if technically symmetric facilities are offered for composing systems in an aspect-oriented fashion, the fact that the introduction of control flow that necessarily is a cross-cutting concern implicitly creates asymmetry may be an argument on what types of systems benefit from aspect-orientation. Secondly, the fact that in some cases abstractions are not executable as such but bear declarative meaning is something that may affect unit testing of aspects, and give weight for research aiming at unit testing at the level of aspects, which seems to require a test driver or stub that models

the expected control flow. Finally, we believe that it is essential for early use of aspect-oriented techniques to overlook control flow oriented dominance, and focus on problem-oriented decompositions. Later on, implementation techniques, where control flow is intimately present, can then be used to introduce this concern, following the practices of Model-Driven Architecture, MDA [8,21]. In essence, both Hyper/J and AspectJ require considering similar issues at the level of Platform Specific Models without offering much support for Computation or Platform Independent Models of MDA. Therefore, the facilities the developer can be benefitted from can be considered restricted.

Finally, we believe that in addition to the low-level relation of different concepts discussed in this paper, the relation between specification level concepts and corresponding aspect-oriented techniques enable a more sophisticated view to early use of aspect-orientation. Although based on different origins and terminology, the possibility to compose such an alignment gives a raise to an extended discussion [2].

References

[1] Timo Aaltonen, Joni Helin, Mika Katara, Pertti Kellomäki, and Tommi Mikkonen. Coordinating objects and aspects. *Electronic Notes in Theoretical Computer Science*, 68(3), March 2003.

[2] Timo Aaltonen, Mika Katara, Reino Kurki-Suonio, and Tommi Mikkonen. On horizontal specification architectures and their aspect-oriented implementations. Accepted to Transactions on AOSD, to appear.

[3] Timo Aaltonen, Mika Katara, and Risto Pitkänen. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3–18, 2001. At URL `http://www.jucs.org`.

[4] AOSD-Europe. Survey of aspect-oriented analysis and design approaches. At URL `http://www.aosd-europe.org`, May 2005.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[6] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[7] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[8] Object Management Group. MDA guide version 1.0.1. OMG Document Number omg/2003-06-01, June 2003.

[9] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmerically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Thomas J. Watson Research Conter, December 2002.

[10] Seppo Isojärvi. DisCo and Nokia: Experiences of DisCo with modeling real-time system in multiprocessor environment. Formal Methods Europe Industial Seminar 1997, Otaniemi, Finland, February 20, 1997.

[11] Pertti Kellomäki and Tommi Mikkonen. Design templates for collective behavior. In Elisa Bertino, editor, *Proc. ECOOP 2000, 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 277–295. Springer-Verlag, 2000.

[12] Reino Kurki-Suonio. *A Practical Theory of Reactive Systems — Incremental Modeling of Dynamic Behaviors*. Springer, 2005.

[13] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[14] Tommi Mikkonen. A development cycle for dependable reactive systems. In Y. Chen, editor, *Proc. IFIP International Workshop on Dependable Computing and its Applications, DCIA98*, pages 70–82. University of Witwatersrand, Johannesburg, South Africa, 1998.

[15] Risto Pitkänen and Petri Selonen. A UML profile for executable and incremental specification-level modeling. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 158–172. Springer, 2004.

[16] K. Systä. A graphical tool for specification of reactive systems. In *Proc. Euromicro'91 Workshop on Real-Time Systems*, pages 12–19. IEEE Computer Society Press, 1991.

[17] P. Tarr, H. Ossher, and Jr. S. M. Sutton. Multi-dimensional separation of concerns. In David Garlan, editor, *Proc. 21st International Conference on Software Engineering*, pages 107–119. ACM Press, 1999.

[18] AspectJ home page at URL http://aspectj.org.

[19] DisCo home page at URL http://disco.cs.tut.fi.

[20] Hyper/J home page at URL http://www.alphaworks.ibm.com/tech/hyperj.

[21] MDA home page at URL http://www.omg.org/mda/.