Symmetric Aspect-Orientation: Some Practical Consequences

Jaroslav Bálik Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 3, 84216 Bratislava 4, Slovakia
jaroslav.balik1@gmail.com, vranic@fiit.stuba.sk

Abstract

To some extent, contemporary software development has incorporated the AspectJ style of aspect-oriented programming. This style is denoted as asymmetric since it explicitly distinguishes between aspects and the base. Although academic symmetric aspect-oriented approaches, in which there is no such distinction, gained no direct acceptance in industry, several approaches used in practice exhibit symmetric aspect-oriented features. As shown in this paper, this ranges from peer use cases and features as analysis and design concepts to particular programming language mechanisms such as traits (Scala), open classes (Ruby), or prototypes (JavaScript). Even inter-type declarations and advices as known from AspectJ can be used to emulate symmetric aspect-oriented programming. The examples given in this paper indicate the basic possibilities for this. However, detailed studies of the corresponding academic and industry approaches should be carried.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design

General Terms Design, Languages

1. Introduction

High hopes have been put into aspect-orientation as a way towards improving software modularity. As industry adoption doesn't quite look like a fulfillment of these hopes, academia responds by reinitiating the search for approaches to software development that will provide a better modularity. This is legitimate, but it might be that a key to the solution of this problem lies in analyzing what out of existing aspect-oriented approaches seems to be acceptable or even appealing to industry and why. In this, the approaches to software development that are only a step from being aspect-oriented should be of a special interest.

In so-called asymmetric aspect-oriented approaches there is a distinction between the base and aspects that affect this base. In contrast to this, in symmetric aspect-oriented approaches, applications are composed out of partial views—or aspects—without explicitly denoting any of them as a base. While asymmetric approaches found their way to industry, symmetric approaches seem to remain the realm of academic research. Surprisingly, several contemporary approaches to software development commonly not classified as aspect-oriented exhibit some features of symmetric aspect-oriented approaches.

This paper attempts to point to some of prominent symmetric aspect-oriented features applied in practice or those that could be easily applied as a straightforward and seamless extrapolation of the industry state of the art. Section 2 clarifies the issue of symmetry in aspect-oriented approaches. Section 3 explains how use cases actually represent symmetric aspect-oriented decomposition. Section 4 reflects on feature modeling and version control. Section 5 describes some features in programming languages used in industry close to symmetric aspect-oriented programming.

2. Symmetry of Aspect-Oriented Approaches

Symmetry is an important property of aspect-oriented approaches. Simply stated, asymmetric aspect-oriented approaches distinguish between so-called basic elements and aspects that affect them or other aspects. This is characteristic for PARC AOP [12] and AspectJ as its language representative. AspectJ and number of other languages and frameworks influenced by it are used in industry.

Symmetric aspect-oriented approaches treat all elements equally with elements representing partial views of classes. The elements are composed according to composition rules, which are usually introduced separately. Hyper/J is a symmetric aspect-oriented language [14]. It ended at a prototype level and as such has never been used in industry applications. CaesarJ strived to support symmetric aspect-oriented

programming, but it has had no industrial application either; a recent overview of industrial AOSD projects reports only one controlled experiment [15].

Beside element symmetry, a complex view of symmetry includes relationship and join point symmetry [9].[1] If an element itself defines with what other elements it is composed—as an aspect in AspectJ does by its pointcuts—such an approach exhibits a relationship asymmetry. Relationship symmetry is achieved if the relationships can be placed in any of the element they involve, or outside of them. Join point symmetry as conceived by Harrison et al. [9] is defined only in the sense of static aspect-oriented composition (that can be performed on lexical basis), so it is of limited applicability to contemporary aspect-oriented approaches.

In the PARC AOP approach, the main decomposition is object-oriented. Crosscutting concerns are encapsulated in elements called aspects. The structure of aspects is different than the structure of the base decomposition elements, which constitutes an element asymmetry. Aspects can affect the base decomposition, but the opposite direction of influence is impossible, so there is also a relationship asymmetry, too.

Subject-oriented programming in Hyper/J implements concerns by the means of partial, subjective classes organized into so-called hyperslices and hypermodules. Subjective classes have the same structure for all concerns, therefore subject-oriented programming is symmetric from the element perspective. Every concern can affect other concerns, so from the relationship perspective, it is also symmetric.

While PARC AOP appears to be fully asymmetric, and subject-oriented programming fully symmetric, other approaches may exhibit mixed symmetry [9]. For example, composition filters [1, 7] are asymmetric with respect to elements: the main concern is implemented in classes and crosscutting concerns are implemented in input and output filters. With respect to relationships, composition filters are symmetric: the relationships are placed outside of the elements and can relate filters to each other, too (though only to define the order of their application).

3. Peer Use Cases

A use case describes a coherent functionality that provides some result of value to a user. As the term says, it is a case of a system use [2]. As such, use cases can be seen as modules of specification. This kind of modules is well accepted by both developers and users, but with common modularization techniques gets lost in design and, consequently, code itself.

Extension use cases—use cases that extend by one or more of their flows other use cases—are clearly an asymmetric aspect-oriented mechanism, as pointed out by Jacobson and Ng [11]. Extensions can be directly implemented by advices which keeps extension use cases modular in code.

Other use cases with no extend or include relationships between them—so-called peer use cases—can be preserved in code, too, again by using aspect-orientated programming [11]. Each use case contributes with what is necessary for achieving its functionality in its own module. These modules are then composed to get a whole application. This is a case for symmetric aspect-oriented programming.

Consider an example in Fig. 1. The *Enroll into Study Year* and *End Study* are peer use cases. A closer inspection of their behavior expressed by sequence diagrams displayed in Fig. 2 reveals the structural elements behind them.

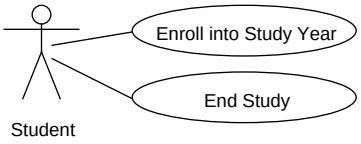

Figure 1. Peer use cases.

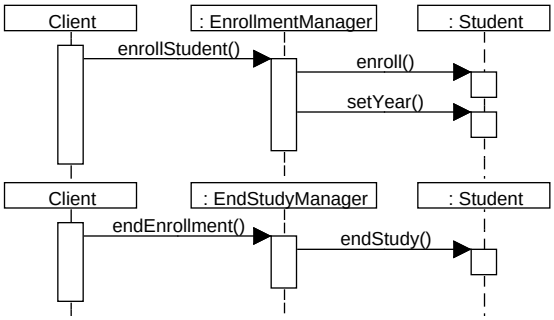

Figure 2. Sequence diagrams of peer use cases.

Under the influence of object-oriented programming, we tend to view these elements as objects. The behavior they are involved in can be expressed by collaborations with the corresponding classes associated to them as shown in Fig. 3. These collaborations are actually use case realizations.

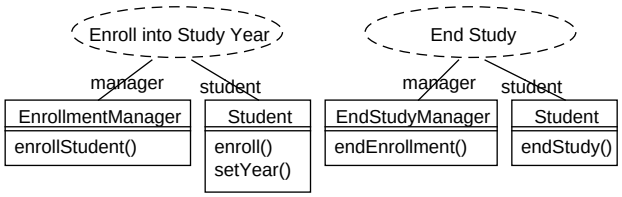

Figure 3. Collaborations with partial classes.

It is immediately clear that each collaboration has its own version of the *Student* class. Although less obvious, but with appropriately generalized association role names a little bit clearer, the manager classes appear to be actually one and only *Manager* class. However, with aspects we may keep these views—or aspects—separate of each other, with the pieces of each view kept together.

This is something developers actually do in use case driven software development, a common approach today. However, they weave models manually ending up with woven models and woven code based on these models. Given

[1] Some points on symmetry presented here are a part of our earlier work [3].

the right tools, developers could easily accept the idea of maintaining separate views of models and code.

Another point is worth mentioning here. Theme [4] is a comprehensive approach to aspect-oriented analysis and design that supports both asymmetric and symmetric modeling keeping concerns—called *themes* there—modularized all the way from specification to implementation. There are no reported industry applications of the Theme approach as such, but it has been demonstrated that Theme/Doc is very close to use case modeling [20]. This is as if one of the best known academic approaches to aspect-oriented analysis and design has been already widely used in practice.

4. Feature Modeling

Although academic feature modeling notations are not used in practice [10], feature decomposition is commonly used in contemporary software development. Features are often easily identified at the time of requirement specification and they can serve as a backbone during entire software evolution, which is typical in software product lines.

Features are often given as separate tasks to programmers and—if proper commit messages are used—they can be tracked in version control systems. With enhanced support for merging and branching already available in distributed version control systems like Mercurial or Git, it becomes easier to maintain features in different feature branches [8].

Branch merging corresponds to symmetric aspect-oriented composition. The elements of the main branch are of the same category as the elements of feature branches, which constitutes an element symmetry. Each branch can affect the elements of any other branch if they are declaratively complete, i.e. they declare everything to which they refer to [19], which constitutes a relationship symmetry.

A feature branch is forked from the main branch when developing a new feature. After completing the feature implementation, its branch is merged back with the main branch. The problem is that feature branches are temporary, so the next step in making distributed version systems more aspect-oriented is to make branches persistent.

5. Aspect-Oriented Implementation in Established Programming Languages

While designed-to-be symmetric aspect-oriented programming languages can't be said to have had success in industry, some features in languages used in industry are getting very close to the idea of symmetric aspect-oriented programming.

5.1 Traits

The traditional inheritance model is criticized for mixing the stable data interface with unstable behavioral interface, which leads to a poor class design, such as deep inheritance hierarchies, unnecessary methods in parent classes, and bad encapsulation [17]. This problem can be overcome with traits. A trait is a a group of pure methods that serves as a building block for classes and is a primitive unit of code reuse [18]. Scala provides a trait construct under this very name, but the idea of traits can be traced even back to C++'s templates whose member functions are composed with those of a concrete class at compile time, so that the object exhibits both the class and template behavior at run time [17].

Assume we have to implement a student from different perspectives for the purposes of a university information system to be developed in Scala. The basic class is empty:

```
class Student() { }
```

The basic student data are introduced by a trait:

```
trait BasicStudent extends Student{
    var _name = ""
    var _surname = ""
    def setName(str:String) = { _name = str }
    def setSurname(str:String) = { _surname = str }
    def getName = _name
    def getSurname =_surname
}
```

Some students study only part-time. Assume such students are due to pay a tuition fee. This can be implemented by another trait:

```
trait PartTimeStudent extends Student {
    var _tuitionFee = 0
    def payTuitionFee(amount:Int) =
        { _tuitionFee = amount + _tuitionFee }
    def tuitionFee = _tuitionFee
}
```

The class and trait are composed using the **with** clause to get an extended behavior for John Doe, who is a part-time student:

```
object App {
    def main(args : Array[String]) {
        var student = new Student() with PartTimeStudent
            with BasicStudent
        ...
    }
}
```

5.2 Open Classes

There are several programming languages derived from Smalltalk that support so-called open classes. A popular representative of this group is Ruby. With Ruby's open classes, it is possible to add and modify the members of the same class at multiple places of source code.

This feature can be used to implement subjective classes. Such subjective classes can be conveniently stored in separate files. The composition would then be determined by the order in which they are included in the application.

An example from the previous section is here implemented in Ruby. The basic student data are expressed by a class provided in the Student.rb file:

```
class Student
    def initialize(name, surname)
      @name = name
      @surname = surname
    end
    def name; @name; end
    def surname; @surname; end
end
```

The additional functionality needed for part-time students is expressed by another Student class provided in a separate file named PartTimeStudent.rb:

```
class Student
    def payTuitionFee(val)
        if @tuitionFee == nil
            @tuitionFee = val
        else
          @tuitionFee = @tuitionFee + val
        end
    end
    def tuitionFee @tuitionFee end
end
```

The composition of partial classes is then realized by importing the corresponding files with the require clause:

```
require "./Student.rb"
require "./PartTimeStudent.rb"
...
```

5.3 Prototype-Based Programming

Prototype-based programming is a kind of object-oriented programming that relies on objects with a complete absence of classes. Instead of class inheritance, the prototype object cloning is used.

A widely used prototype-based language is JavaScript, a member of the ECMAScript family. The implementation of subjective views is quite similar to the one with open classes, but changes are realized on a single object.

In our part-time student example, the basic student prototype could be represented by the student object:

```
var student = {
    "_name": "",
    "_surname":"",
    setName":function(name) { this._name = name },
    "getName":function() { return this._name },
    "setSurname":function(surname) { this._surname = surname },
    "getSurname":function() { return this._surname }
};
```

The partTimeStudent object is a clone of student:

```
var Factory = function(){};
Factory.prototype = student;
var partTimeStudent = new Factory();
```

Subsequently, methods and attributes necessary for the role of a part-time student are added to it:

```
partTimeStudent['_tuitionFee'] = 0;
partTimeStudent['payTuitionFee'] =
    function(val) { this._tuitionFee = this._tuitionFee + val };
partTimeStudent['getTuitionFee'] =
    function() { return this._tuitionFee };
```

The example presented here is implemented in pure JavaScript. The JavaScript framework called Prototype[2] gets even closer to subject-oriented programming with its extend clause that enables to merge subjects by copying the source object members to the target object.

5.4 Emulation in Asymmetric Approaches

Symmetric aspect-oriented programming can be emulated to some extent in asymmetric approaches. The key is to keep the base as thin as possible and to build everything with aspects. Inter-type declarations can be used to establish the structure, including initial method bodies. The behavior is then implemented by advices.

6. Related Work

There has been a continuous effort to convince industry of usefulness of aspect-oriented programming [22]. This paper focuses on identifying unrecognized mainstream uses of symmetric aspect-oriented approach.

Partial class models result from peer use cases. WEAVR [5, 6] is a practical approach to partial UML model weaving, but it's asymmetric. It's based on graphical pointcut specification recalling Join Point Designation Diagrams.[3]

The Theme approach embraces the way to implement themes using AspectJ, AspectWerkz, and Concern Manipulation Environment,[4] so they persist in code. This is very similar to preserving use cases as proposed by Jacobson and Ng and to symmetric aspect-oriented programming emulation presented in Sect. 5.4.

If they are not corrective, changes can be viewed and modeled as additional application features [21]. Often they affect multiple places in the application. Sometimes they need to be taken out and even reapplied to another development line of the same application. Even more important, for their further maintenance, it is desirable to have changes modular. If the application feature model is missing, partial feature models can be used [13]. In change implementation, asymmetric aspect-orientated programming is used.

The Data–Context–Interaction (DCI) [16, 17] paradigm's role based design is very close to symmetric aspect-oriented approach. DCI relies on traits for implementing roles that can be used to emulate symmetric aspect-oriented programming (see Sect. 5.1).

[2] `http://www.prototypejs.org/`

[3] `http://www.dawis.wiwi.uni-due.de/en/research/foci/aosd/jpdds/`

[4] Concern Manipulation Environment (`http://www.research.ibm.com/cme/`) was an IBM project that included Hyper/J features. Unfortunately, it isn't publicly available.

7. Conclusion and Further Work

Although academic symmetric aspect-oriented approaches, in which there is no distinction between aspects and the base, gained no direct acceptance in industry, several approaches used in practice exhibit symmetric aspect-oriented features. As shown in this paper, this ranges from peer use cases and features as analysis and design concepts to particular programming language mechanisms such as traits (Scala), open classes (Ruby), or prototypes (JavaScript). Even inter-type declarations and advices as known from AspectJ can be used to emulate symmetric aspect-oriented programming.

The examples given in this paper just indicate the basic possibilities for this. The awareness of what can be done with symmetric aspect-oriented approaches in industry today directly or with only a small effort to adopt them should be raised and promoted by in-depth analysis and comparison of the corresponding academic and industry approaches.

Acknowledgments

This publication is the partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF. The work was partially supported by the Scientific Grant Agency of Slovak Republic (VEGA), grants No. VG 1/1221/12 and VG 1/0675/11.

References

[1] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In *Proc. of the Aspect-Oriented Programming Workshop at ECOOP'98*, Brussels, Belgium, 1998.

[2] J. Arlow and I. Neustadt. *UML 2 and the Unified Process*. Addison-Wesley, 2005.

[3] J. Bálik and V. Vranić. Sustaining composability of aspect-oriented design patterns in their symmetric implementation. In *2nd International Workshop on Empirical Evaluation of Software Composition Techniques, ESCOT 2011, at ECOOP 2011*, Lancaster, UK, July 2011.

[4] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[5] T. Cottenier, A. van den Berg, and T. Elrad. The motorola WEAVR: Model weaving in a large industrial context. In *Proc. of 6th International Conference on Aspect-Oriented Software Development*, Vancouver, British Columbia, Canada, Mar. 2007. ACM.

[6] T. Cottenier, A. van den Berg, and T. Elrad. Motorola WEAVR: Aspect orientation and model-driven engineering. *Journal of Object Technology*, 6(7):51–88, Aug. 2007. `http://www.jot.fm/issues/issue_2007_08/article3`.

[7] A. R. de, M. Hendriks, W. Havinga, P. Durr, and L. Bergmans. Compose*: a language- and platform-independent aspect compiler for composition filters. In *Proc of 1st International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008*, Paphos, Cyprus, July 2008.

[8] V. Driessen. A successful Git branching model, Jan. 2010. `http://nvie.com/posts/a-successful-git-branching-model/`.

[9] W. H. Harrison, H. L. Ossher, and P. L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research, Dec. 2002.

[10] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans. A preliminary review on the application of feature diagrams in practice. In *Proc. of International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2010*, Linz, Austria, Jan. 2010.

[11] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of 11th European Conference on Object-Oriented Programming, ECOOP'97*, LNCS 1241, Jyväskylä, Finland, June 1997. Springer.

[13] R. Menkyna and V. Vranić. Aspect-oriented change realization based on multi-paradigm design with feature modeling. In *Proc. of 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009*, LNCS 7054, Krakow, Poland, Oct. 2009. Springer.

[14] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology*. Kluwer, 2002.

[15] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Südholt, and W. Joosen. Aspect-oriented software development in practice: Tales from AOSD-Europe. *Computer*, 43(2):19–26, Feb. 2010.

[16] T. M. H. Reenskaug. The common sense of object oriented programming. `http://folk.uio.no/trygver/2008/commonsense.pdf`, 2008.

[17] T. M. H. Reenskaug and J. O. Coplien. The DCI architecture: A new vision of object-oriented programming. `http://www.artima.com/articles/dci_vision.html`, Mar. 2009.

[18] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *Proc. of 17th European Conference on Object-Oriented Programming, ECOOP 2003*, LNCS 2743, Darmstadt, Germany, July 2003. Springer.

[19] P. Tarr and H. Ossher. *Hyper/J User and Instalation manual*. IBM Research, 2000.

[20] V. Vranić and P. Michalco. Are themes and use cases the same? *Information Sciences and Technologies, Bulletin of the ACM Slovakia*, 2(1):66–71, 2009. Special Section on Early Aspects at AOSD 2010.

[21] V. Vranić, R. Menkyna, M. Bebjak, and P. Dolog. Aspect-oriented change realizations and their interaction. *e-Informatica Software Engineering Journal*, 3(1):43–58, 2009.

[22] D. Wiese, R. Meunier, and U. Hohenstein. How to convince industry of AOP. In *Proc. of 6th International Conference on Aspect-Oriented Software Development*, Vancouver, British Columbia, Canada, Mar. 2007. ACM.