

Towards Aspect-Oriented Class Diagrams

Gefei Zhang

Ludwig-Maximilians-Universität München

gefei.zhang@pst.ifi.lmu.de

Abstract

While aspect-oriented modeling has been recognized as a useful means of improving the modularity of software design, the *de facto* standard modeling language UML lacks first-class model elements representing aspects and does not provide genuine support for aspect-oriented modeling. We propose a simple extension of UML class diagrams which contains a very generic pointcut and advice language and facilitates to model with aspects. Using this approach, we achieve a better separation of concerns as well as more redundancy reduction in UML class diagrams and make them thus more readable and better maintainable.

1 Introduction

Aspect-oriented Programming [8, 7] has proven useful in practice. It helps achieve a better separation of concerns, reduces redundancy, and thus makes the *program code* less error-prone and better maintainable. On the level of *software design*, however, aspects are not yet sufficiently supported. In particular, there does not exist any aspect-oriented modeling language, which makes it difficult — if not impossible — for a designer, rather than programmer, to think, specify and document in aspects or to communicate in the world of aspects.

On the other hand, on the level of design separation of concerns and redundancy reduction would also be desirable. Not only the business logic code should be separated from the GUI code, but also the design models of business logic from those of the GUI; not only the implementation of access control should be modular, but also their design; not only does program code benefit from redundancy reduction by an aspect-oriented programming language, design models with less redundancy would be less error-prone in the same way — no wonder aspect-orientation is advantageous for both models and code, models are simply abstract code after all.

The Unified Modeling Language (UML) [9] is the *lingua franca* in object-oriented software analysis and design. It

provides several diagrams for software specification in both static and dynamic views. Class diagrams provide a static view of the classes of the objects that are used in the system and their relationships. However, due to the UML's lack of aspect-oriented language constructs, cross-cutting concerns of a software system cannot be modeled modularly in class diagrams. In the same way object-oriented programming languages are extended by aspect-oriented programming languages, we propose a simple extension of UML class diagrams, in which aspect-oriented constructs are used to achieve a more modular design.

Equipped with aspects as first class model elements, aspect-oriented class diagrams provide several benefits: reduction of redundancy, thus making the models less error-prone, a better separation of concerns, which makes the model better readable and maintainable, and support for additive rather than invasive change in software design.

The rest of this paper is organized as follows: The following Sect. 2 defines the concrete and the abstract syntax of aspect-oriented class diagrams and explains the semantics informally. In Sect. 3 we give some examples of using aspect-oriented class diagrams in software design. Related work is discussed in Sect. 4. Finally, we conclude and outline some future work.

2 Aspect-Oriented Class Diagram

We introduce the concept of aspect-orientation into UML class diagrams. An *aspect* consists of a *pointcut* and an *advice* and defines some model transformation. The transformation is done by a function *weave*, the process of the transformation is called *weaving*.

Given a base model M in the form of a UML class diagram, we define $M' := M.\text{ownedElement} \cup \{M\}$ to be the set that contains all `ownedElement` of M and M itself. The pointcut of an aspect specifies which elements of M' are to be modified. The advice describes declaratively how these elements should be modified. The algorithm of applying an aspect to a model is described in a pseudo programming language in Fig. 1. Note that weaving is only performed when the resulting model is well-formed.

```

FUNC transform(Model m, Aspect a): Model
Model newM := M
Model M' := M.ownedElement UNION {M}
FOR each m: Element IN M'
  Model t := weave(newM, m, a.advice);
  IF a.pointcut(M, m) AND isWellFormed(t)
    THEN newM := t;
RETURN newM
ENDF

```

Figure 1. Aspect and model transformation

We now define a graphical language for specifying *pointcut*, *advice*, and the formal parameter *m*. First we explain the syntax and the (informal) semantics of our generic pointcut and advice language, then we embed the new language constructs into the UML by giving an extension of the UML meta model.

2.1 Concrete Syntax

An aspect is represented as a package. It contains a pointcut compartment and an advice compartment. The notation of pointcuts and advice, as well as their (informal) semantics, is described in the following subsections.

2.1.1 Pointcut

A pointcut is also a package, the elements contained in it define which elements of a given base model *M* (or *M* itself) are selected and thus subject to weaving. It is a graphical notation of the boolean function *pointcut* used in Fig. 1.

The formal parameter *m* is marked with an icon “?”. It may be any kind of element and represents the elements to select. Each property *f* of the model element marked with “?” represents a selection criterion: model elements to select must have property *f* if it is graphically denoted normally as in usual UML class diagrams, and must not have it if it is crossed through. If several criteria are specified, the selected model elements satisfy all of them. This way, the elements owned by a pointcut form a pattern. If the given base model *M* matches the pattern, then all elements of *M* matching the model elements marked by “?” are selected.

By defining the selection criteria on the highly abstract level of model elements and properties our pointcut language is very generic. The elements to select may be any kind of model elements; not only classifiers (classes and interfaces) may be selected, but also attributes, operations, associations, etc.

Figure 2 shows some examples of our pointcut language. In Fig. 2(a), the formal parameter is a class, of which no other property is specified. This pointcut

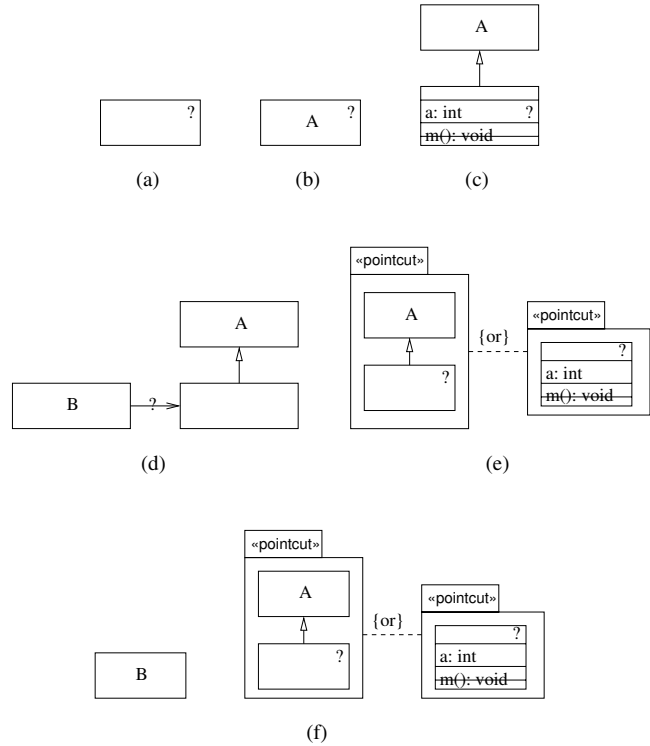


Figure 2. Pointcut examples

selects all classes in the base model without any additional constraint. The only property of the class specified in Fig. 2(b) is its name *A*, which means that all classes whose name is *A* in the base model — there exists at most one such class if the base model is consistent — are selected. In Fig. 2(c) the formal parameter is an attribute and selects the attribute *a* of type *int* in every subclass of class *A* that does not possess any operation *m(): void*. Finally, Fig. 2(d) selects all the associations from *B* to some subclass of *A*.

Pointcuts may be combined. E.g., Fig. 2(e) selects all those classes that are either derived from class *A* or have an attribute *a: int* and do not have any operation *m(): void*.

A pointcut may also contain elements that do not present any property of the formal parameter. Graphically this means that the pointcut contains some “isolated” elements that are not connected with any formal parameter. These elements must be matched by the base model. For example, if there is a class, whose name is *B*, in the base model, then Fig. 2(f) selects the same model elements as Fig. 2(e), otherwise it does not select any element.

A pointcut does not need to contain any formal parameter. In this case, we define that the model itself is selected. This is necessary when the model itself instead of some *ownedElement* should be modified.

2.1.2 Advice

The advice of an aspect defines the function *weave* in Fig. 1. If the pointcut does not specify any formal parameter, then the advice must not contain any either, in this case the advice defines modifications to the base model itself; otherwise the advice must contain exactly one formal parameter, which is of the same type as the formal parameter in the pointcut. The features of the formal parameter are interpreted similarly as in the pointcut: for every element *m* selected by the pointcut and thus bound to the formal parameter and every property *p* of the formal parameter, if *p* is depicted normally as in a usual UML class diagram, then after the weaving *m* also has property *p*, and if *p* is crossed out then after the weaving, *m* does not have property *p*.

Normally depicted properties are used to add new properties to the elements selected by the pointcut. If these properties are already available in the base model, then they simply have no impact on the modification of the base model. Properties that are crossed out are used to delete properties from the elements selected by the pointcut. If the selected elements do not show these properties in the base model, then these properties have no impact on the model modification defined by the aspect.

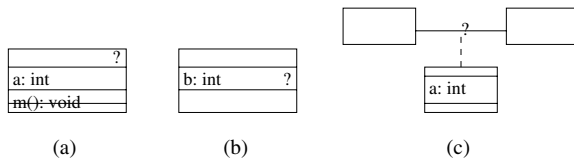


Figure 3. Advice examples

Figure 3(a) shows an advice which adds an attribute `a: int` to and deletes an operation `m(): void` from every selected class. This advice may appear in any aspect whose pointcut has a class as formal parameter. If a selected class from the base model already has the attribute `a: int` or does not have the operation `m(): void`, the corresponding property will simply be ignored at weaving time. Note that this advice implies a select criterion in the pointcut: only those classes are selected that do not have an attribute `a` which is not of type `int`. The advice shown Fig. 3(b), if used together with a pointcut that selects some attributes, overrides every selected attribute with `b: int`. Similarly, the aspect only applies to those classes that do not have an attribute `b` of any other type. Figure 3(c) defines an advice which may be used together with any pointcuts that selects associations and adds the properties “being an association class” and “having an attribute `a: int`” to all the selected associations.

2.2 Abstract Syntax

We embed the constructs defined in the last subsection into the UML by a simple extension of the UML meta model and give well-formedness rules of aspect-oriented class diagrams in the Object Constraint Language (OCL) [15].

Aspect is defined as a stereotyped package. An aspect has a pointcut and an advice, both of which are also stereotyped packages. The meta model extension is shown in Fig. 4.

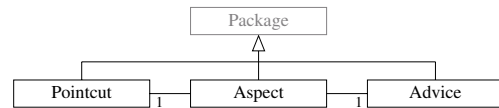


Figure 4. Meta model extension introducing aspects into class diagrams

For specifying the formal parameter *m*, we introduce a stereotype `«parameter»`, which extends the meta class Element. See Fig. 5.



Figure 5. Meta model extension: stereotype Parameter

Before giving the well-formedness rules concerning formal parameters, we first define the following functions which return the set of all formal parameters of a pointcut and an advice, resp.

```
context Pointcut
def: fm : Collection(Element) =
  ownedElement -> select(
    stereotype -> exist(
      name = 'parameter'))

context Advice
def: fm : Collection(Element) =
  ownedElement -> select(
    stereotype -> exist(
      name = 'parameter'))
```

The well-formedness rules concerning formal parameters in pointcut and advice are the following:

- If the pointcut contains formal parameters, they must be of the same type, and the advice must also contain one of this type.

```

context Aspect inv:
pointcut.fm -> size() > 0 implies
  advice.fm -> size() = 1 and
  pointcut.fm -> forall(
    oclType = advice.fm
      -> toSequence()
      -> at(1).oclType)

```

- If the pointcut does not contains any formal parameter, then the advice must not contain any, either.

```

context Aspect inv:
pointcut.fm -> isEmpty() implies
  advice.fm -> isEmpty()

```

3 Examples

We demonstrate aspect-oriented design using UML by means of some examples.

3.1 Logging

Logging is a typical cross-cutting concern that should at best be designed and implemented using aspect-oriented techniques. At design time, the feature that every class has an operation `log(): void` can be modeled by the aspect shown in Fig. 6(a). If it is required that the individual classes not be responsible for logging themselves but a central class `Logger` be used, aspect `Logger` as shown in Fig. 6(b) is used, ensure that the classes do not have an operation `log`, instead, they are associated with the class `Logger`. Note that using class diagrams we do not model the behavior of operations and/or classifiers, but only the signatures of the operations. Modeling that all classes in the system call the function `Logger.log()` at some dedicated time during execution requires techniques of aspect-oriented behavior modeling, which is subject to future research.

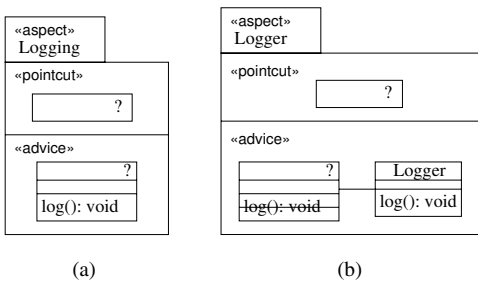


Figure 6. Example: Logging

3.2 Airline

Figure 7(a) shows the model of some software system that manages flights, planes and passengers of an airline, where a flight is run by a plane and may have many passengers. To add mobility to this system and model explicitly that the instances of `Plane` and `Passenger` can change their location, the UML profile *Mobile UML* [5] is used, where the mobile objects must have an attribute `atLoc: Location` representing the current location and their classes are stereotyped as `mobile`. This is easily modeled using the aspect `Mobility` shown in Fig. 7(b). Note how the aspect reduces redundancy and modularizes the design. Aspect `Booking` defined in Fig. 7(c) makes the association between `Flight` and `Passenger` to an association class and thus allows to model flight bookings. Using aspects to model orthogonal features like these achieves a clear separation of concerns.

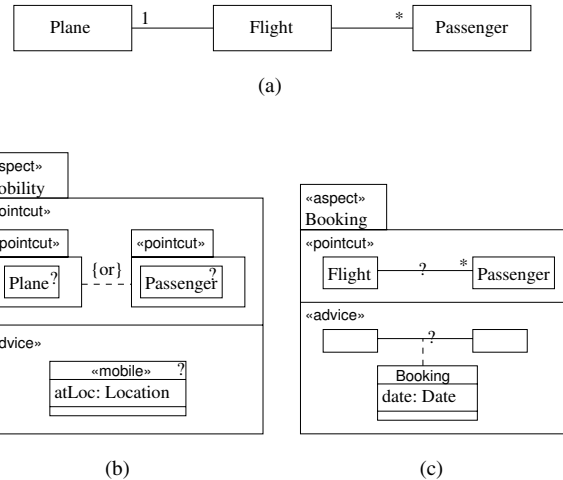
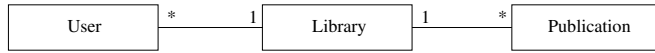


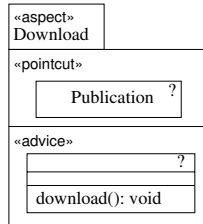
Figure 7. Example: Airline

3.3 Library

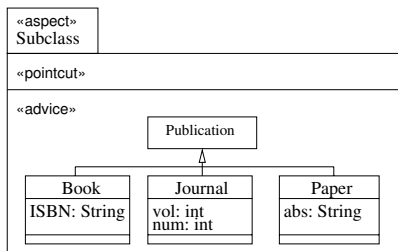
Suppose a library uses some software system for publication and user management. The base model is shown in Fig. 8(a). In order to develop an online library where the users can download the publications via the Internet, class `Publication` must be enhanced by an operation `download(): void`, which sends the content of a publication to, say, the Web server. This enhancement is modeled by the aspect `Download` as depicted in Fig. 8(b). Aspect `Subclass`, which is depicted in Fig. 8(c), differentiate three kinds of publications: books, which have an ISBN number; journals, which have a volume number, and papers, each of which has an abstract. The point-



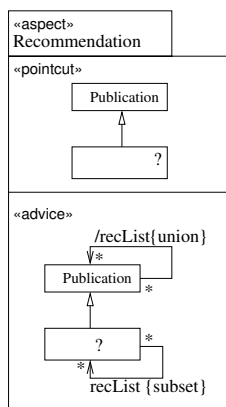
(a)



(b)



(c)



(d)

Figure 8. Example: Online Library

cut is empty, and the advice adds the three subclasses of `Publication` to the model. For an online library, to provide for each publication a recommendation list of similar or related publications may be desirable (a similar feature is the “*Customer-who-bought-this-product-also-bought...*” list of www.amazon.com). In order to add this feature to our system, aspect `Recommendation`, as shown in Fig. 8(d), defines for class `Publication` and each of its subclasses a reflexive association, which models the recommendation list of the publication. Note that this design ensures that the recommendation list of a publication consists of only publications of the same kind, which cannot be expressed modularly without using aspects.

4 Related Work

Our approach is to our knowledge the first one that makes UML class diagrams aspect-oriented. Compared with most existing work concerning aspect-oriented modeling (cf. [1, 3, 10]), our work focuses on modeling in an aspect-oriented way using UML rather than using UML to model aspect-oriented programs.

While using templates to represent pointcuts *or* advice in the UML is not a new idea, e.g. see [11], our approach is to our knowledge the first one that combines both pointcut *and* advice in a first-class model element. The graphical notation of aspects using our syntax is therefore more compact.

Stein et al. propose in [12, 13] to use templates to represent criteria of classifier selection. Our pointcut can select not only classifiers, but also other kinds of elements such as associations, attributes, operations and so on.

Theme/UML ([6, 2]) uses templates and collaborations to model cross-cutting behavior of classifiers. Aspect-oriented class diagrams, on the contrary, are used to describe the static structures of software systems modularly.

Straw et al. define in [14] a set of (textual) composition directives to describe customizable model composition. Their low level directives `create`, `add`, `remove` and `override` can all be expressed in our graphical advice language. Graphical notation of their high level directives, which describe the orders of model compositions, is still subject to future work.

A case study of using aspect-oriented class diagrams to improve the design of adaptive Web applications is given in [4], although a slightly different notation is used there.

5 Conclusions and Future Work

We have defined an aspect-oriented extension of UML class diagrams. Aspects are introduced into class diagrams as first-class elements, consisting of a pointcut and an advice. Using aspects, the static structures of software sys-

tems are modeled more modularly, and redundancy in models can be reduced. We are currently working on tool support for this language and a formal semantics.

An important subject of future work is aspect composition. When different concerns are modeled separately by different aspects, how should they be integrated with each other to build a system that is cross-cut by all these concerns? Which properties does the result of aspect integration have? These questions are to be answered by future research.

Class diagrams are used to specify the static structures of software systems. Aspect-oriented behavior modeling calls for genuine support for aspects by interaction and activity diagrams as well as state charts. Our future work includes defining the necessary language constructs to introduce aspect-orientation into these diagrams.

A further topic of future work is generation of aspect-oriented program code from aspect-oriented design models.

Acknowledgements

This research has been partially sponsored by the Deutsche Forschungsgemeinschaft (DFG) within the project MAEWA (WI 841/7-1). The author also thanks Hubert Baumeister and Alexander Knapp for fruitful discussions related to this paper.

References

- [1] O. Aldawud, T. Elrad, and A. Bader. UML Profile for Aspect-Oriented Software Development. In *3rd Int. Wsh. Aspect-Oriented Modeling (AOM)*, Boston, March 2003.
- [2] E. L. A. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. In *Proc. 26th Int. Conf. Software Engineering (ICSE'04)*, pages 158–167. IEEE, 2004.
- [3] M. Basch and A. Sanchez. Incorporating Aspects into the UML. In *3rd Int. Wsh. Aspect-Oriented Modeling (AOM)*, Boston, March 2003.
- [4] H. Baumeister, A. Knapp, N. Koch, and G. Zhang. Modelling Adaptivity with Aspects. In D. Lowe and M. Gaedke, editors, *5th Int. Conf. Web Engineering (ICWE'05)*, volume 3579 of *Lect. Notes Comp. Sci.*, pages 406–416. Springer, Berlin, 2005.
- [5] H. Baumeister, N. Koch, P. Kosiuczenko, P. Stevens, and M. Wirsing. UML for Global Computing. In C. Priami, editor, *Proc. 1st IST/FET Int. Wsh. Global Computing (GC'03)*, volume 2874 of *Lect. Notes Comp. Sci.*, pages 1–24. Springer, Berlin, 2003.
- [6] S. Clarke. Extending Standard UML with Model Composition Semantics. *Sci. Comp. Program.*, 44(1):71–100, 2002.
- [7] T. Elrad, R. Filman, and A. Bader. Aspect-Oriented Programming. *Comm. ACM*, 44(10):29–32, 2001.
- [8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proc. 11th Eur. Conf. Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lect. Notes Comp. Sci.*, pages 220–242. Springer, Berlin, 1997.
- [9] Object Management Group. Unified Modeling Language: Superstructure, version 2.0. Specification, 2005. <http://www.omg.org/cgi-bin/doc?formal/05-07-04>.
- [10] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli. A UML Notation for Aspect-Oriented Software Design. In *1st Int. Wsh Aspect-Oriented Modeling (AOM'02)*, Enschede, The Netherlands, 2002.
- [11] D. Stein, S. Hanenberg, and R. Unland. A UML-based Aspect-oriented Design Notation for AspectJ. In *Proc. 1st Int. Conf. Aspect-Oriented Software Development (AOSD'02)*, pages 106–112, 2002.
- [12] D. Stein, S. Hanenberg, and R. Unland. A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. In U. Aßmann, editor, *Proc. Int. Wsh. Model Driven Architecture: Foundations and Applications (MDAFA'04)*, pages 60–74, 2004.
- [13] D. Stein, S. Hanenberg, and R. Unland. Query Models. In T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, editors, *Proc. 7th Int. Conf. Unified Modeling Language (UML'04)*, volume 3273 of *Lect. Notes Comp. Sci.*, pages 98–112. Springer, Berlin, 2004.
- [14] G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. M. Bieman. Model Composition Directives. In T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, editors, *Proc. 7th Int. Conf. Unified Modeling Language (UML'04)*, volume 3273 of *Lect. Notes Comp. Sci.*, pages 84–97. Springer, Berlin, 2004.
- [15] J. Warmer and A. Kleppe. *The Object Constraint Language 2nd Edition: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.