Aspect-Oriented Design with the UML*

Wai-Ming Ho, François Pennaneac'h, Jean-Marc Jézéquel, Noël Plouzeau

[1] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, FRANCE
[2] waimingh, pennanea, jezequel, plouzeau@irisa.fr

1 Introduction

Separation of concerns [5] is a basic engineering principle that can bring many benefits: additive, rather than invasive, change; improved comprehension and reduction of complexity; adaptability, customizability, and reuse. Having evolved from best practices in object-oriented software engineering, the Unified Modeling Language (UML) offers such a separation of concerns with the decomposition of a software system along four main dimensions: functional (use case diagrams), static (class and package diagrams), dynamic (sequence, collaboration, activity and state diagrams) and physical (component and deployment diagrams). Furthermore the designer can add many non-functional informations to a model by labeling model elements, e.g. with design pattern occurrences [2], stereotypes or tag values.

It is appealing to think of many concerns as being independent or "orthogonal", but this is rarely the case in practice. It is essential to be able to support interacting concerns, while still achieving useful separation. It can then be an overwhelming task for the engineer to reconcile the various aspects of a model into a working implementation. To overcome this problem, we propose the concept of *aspect-oriented design* (AOD for short) which provides some methodological support for a workable separation of concerns in building UML models (Section 2). In Section 3, we briefly describe the UML All pUrpose Transformer (UMLAUT) [1] framework and how its use can help the designer to program the "weaving" of the aspects at the level of the UML metamodel.

2 AOD with the UML

The aim of this section is to extend the ideas expressed in aspect-oriented programming (AOP) [6] to the software design level.

In [1], it is shown how the application of subject-oriented programming concepts to the complete software development cycle can help to maintain consistency of requirements, design and code. Paper [1] also points out that some tool support is needed to express the various subjective views of the system being designed. We conjecture that the same approach can be applied in the case of

* Work partially funded by France Telecom R&D in the METAFOR project

[1] UMLAUT is freely downloadable from http://www.irisa.fr/pampa/UMLAUT/download.htm

AOD, as subject-oriented programming addresses complementary issues with respect to AOP [4,6]. We propose to use UML as a design language. In addition to being an open standard [3], general purpose object-oriented modeling language, UML supports the concept of multiple views. This allows the software designer to express various requirements, design and implementation decisions using each view independently. The views themselves are related to one another through the underlying metamodel of UML, ensuring the coherence of the software model. The extensibility of UML enables customizations for a specific modeling environment.

2.1 Extending Modeling Dimensions of UML

The UML itself includes specific notations to help modeling along several dimensions [7]. Four major dimensions of modeling are supported: functional (use case diagrams), static, dynamic and physical. While such builtin dimensions are a good starting point, they often need to be extended to take many design concerns into account. The UML provides us with notation hooks such as stereotypes, tag values and design pattern occurrences. These hooks can be used to add new dimensions.

Stereotypes can often be used to subtype a given model element type. Automatic tools can then identify this element among the other model elements of the same type. For instance, we may mark in Figure 1 objects of class *History* are persistent. At the same time, a different stereotype designates the operations of class *ServiceProvider* that should participate in the application of the *Command* design pattern [2].

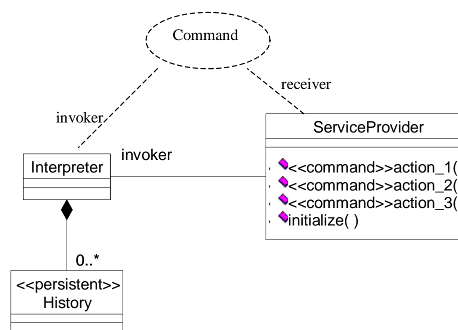

Fig. 1. UML class diagram with stereotype

Design pattern occurrences may be used to adorn a class with additional information on its role within the design model. Using this information, a weaver software can be constructed to select out classes that participate in this design pattern occurrence, and use the information annotated on the operations to

create the appropriate command classes, one for each *ServiceProvider action_i* methods.

Tag values are key-value pairs. Such pairs provide a weaver with additional information to guide the weaving process. Going back to our previous example, tag values could be put on *ServiceProvider* to specify a choice among existing implementations of the *Command* design pattern at weaving time.

These extension and annotation mechanisms gives us flexibility to model all the necessary aspects into our AOD model. The final implementation decision consists in telling the weaver which group of aspects to compose, and how they should be composed according to these non-functional informations.

3 UMLAUT: A Weaver for the UML

The role of the weaver in AOP is to integrate the different dimensions of a design model. A weaver automates the task of blending together the different decisions. In order to perform this integration, a weaver is programmed with a given set of rules that guide it in resolving conflicting demands of cross-cutting concerns. A weaver is thus composed of two components: a driver program that reads various design decisions, and a rule applicator that decides how these decisions are bound in the final product. The UMLAUT tool takes on this two steps approach. The core of the tool serves as the weaver 'driver' and an open transformation framework provides the mechanism for specifying different integration rules.

UMLAUT is a tool dedicated to the manipulation of UML models. Since UML is itself described by a metamodel in UML, manipulating the metamodel is the same as manipulating any model. Hence we deal with the weaving of AOD designs by handling the model at the metamodel level. In our UMLAUT tool, a weaving process is implemented as a model transformation process: each weaving step is a transformation step applied to a UML model.

3.1 General Architecture

UMLAUT's architecture is a three layers architecture (Figure 2). The input front end provides the interface for reading UML models described in various formats (XMI, Rational Rose^{TM} MDL, Eiffel source, Java source). UMLAUT also provides a graphical user interface for editing UML models interactively. The middle core engine is made up of the UML metamodel repository and the extendible transformation engine. Finally, the output back end contains various code generators (Eiffel, XMI). The design concept of UMLAUT is a basic core (the middle layer) that communicates with its surroundings via *hot spots* (*i.e.* interfaces). Functional modules can be plugged in order to specialize the behavior and to meet specific requirements.

3.2 The Core Engine

The core engine is made up of the UML metamodel [3] implementation and a transformation framework. The metamodel is implemented as a set of collabo-

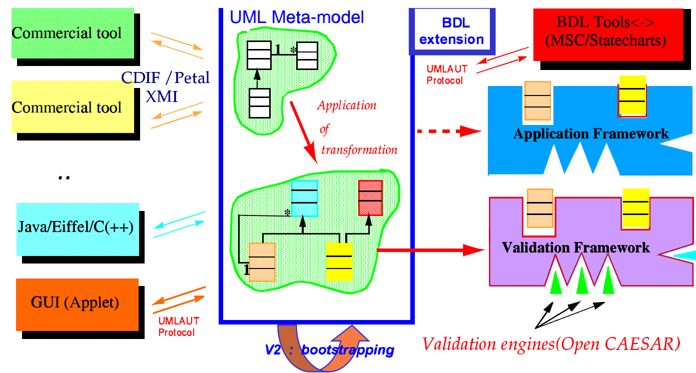

Fig. 2. UMLAUT Architecture

rative Eiffel classes. The resulting implementation is a direct mapping of UML metaclasses to Eiffel classes. At runtime, a UML model is canonically represented as an abstract syntax tree of Eiffel class objects. This direct mapping of UML metamodel specification with an object implementation gives us true representation of any UML model without the risk of information loss along the translation.

3.3 The Extendible Transformation Framework

UMLAUT's transformation engine is implemented as an object-oriented framework, allowing two levels of utilisation:

Application The framework is used as a weaving tool. A weave operation is a composition of transformation operators, chosen from a library in UMLAUT. In general, these operators involves metaprogramming queries like *get_class_name*, modifiers like *add_class_method* or code generation operators such as *generate_attribute_setter*.

Aspect New implementation strategies may be added to the existing library by aspect developers who have in-depth knowledge of performance issues. For example, the *Command* design pattern in figure 1 may have multiple implementations and they are added to the library using the framework abstractions.

The open framework gives users control over how an implementation is realized, while preserving a high level of abstraction. At a different level, it allows aspect experts to develop different strategies for optimal implementation. It encourages a separation of concern between application implementation and aspect implementation.

4 Conclusion

Recent approaches such as adaptive programming, aspect-oriented programming, role-modeling, and subject-oriented programming have enhanced object-oriented programming by providing separation of concerns along additional dimensions, beyond the class concept. We aim at building upon the emerging success of this approach by adopting AOP for the entire software development cycle, in particular at the design level. In order to achieve separation of concerns for multiple aspects during the design process, we propose to use the multiple view modeling capabilities of UML. Each aspect of design and implementation should be declared during the design phase so that there is clear traceability from requirements through source code.

We propose to use UML as the design language and with the help of an open framework as our weaver, to provide an aspect-oriented design environment. UMLAUT builds on the idea of integrating the functional programming paradigm into an object-oriented context. UMLAUT offers the designer a versatile framework for transforming UML models in a way that closely resembles the metaprogramming approach of software refactoring[8–10].

References

1. Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Separating concerns throughout the development lifecycle. In *ECOOP '99 Workshop Proceedings on Aspect-Oriented Programming Proceedings*, 1999.
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
3. Object Management Group. UML version 1.1, July 1997.
4. William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *OOPSLA 1993 Conference Proceedings*, volume 28 of *ACM SIGPLAN Notices*, pages 411–428. ACM Press, October 1993.
5. Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, Northeastern University, february 1995.
6. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, N.Y., June 1997.
7. OMG. Uml notation guide.
8. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.
9. Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.
10. Lance Tokuda and Don Batory. Evolving object-oriented applications with refactorings. Technical Report CS-TR-99-09, University of Texas, Austin, March 1, 1999.