A toolkit for weaving aspect oriented UML designs

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

A Toolkit for Weaving Aspect Oriented UML Designs

Wai-Ming Ho , Jean-Marc Jézéquel, François Pennaneac'h and Noël Plouzeau
Irisa (INRIA & University of Rennes)
Campus de Beaulieu
+33 2 99 84 71 92
Contact: jezequel@irisa.fr

ABSTRACT*

Separation of concerns is a basic engineering principle that is also at the core of object-oriented analysis and design methods in the context of the Unified Modeling Language (UML). The UML gives the designer a rich, but somehow disorganized, set of views on her model as well as many features, such as design pattern occurrences, stereotypes or tag values, allowing her to add non-functional information to a model. Aspect-oriented concepts are applied to manage the multitude of design constraints. However, it can then be an overwhelming task to reconcile the various aspects of a model into a working implementation. In this paper, we present our UMLAUT framework as a toolkit for easily building application specific "weavers" for generating detailed design models from high level, aspect oriented UML models. This is illustrated with a toy example of a distributed multimedia application with a weaving generating an implementation model. More ambitious applications are briefly outlined in the conclusion.

1. INTRODUCTION

Separation of concerns [12] is a basic engineering principle that can provide many benefits: additive, rather than invasive, change; improved comprehension and reduction of complexity; adaptability, customizability, and reuse. With its nine views that can be though of as projections of a whole multi-dimensional system onto separate plans, the Unified Modeling Language (UML) [22] provides the designer with an interesting separation of concerns that Kruchten calls the 4+1 view model (Design view, Component view, Process view, Deployment view, plus Use Case view) [15]. In turn , each of these views has two dimensions, one static and one dynamic. Furthermore the designer can add non-functional information (e.g. persistency requirements) to a model by "stamping" model elements, for instance with design pattern occurrences [8], 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. An aspect-oriented approach to design can help to express these concerns explicitly and we propose our UMLAUT framework as a methodological support for building and manipulating UML models with aspects[1], (Section 2). In Section 3, we describe the *UML All pUrpose Transformer* (UMLAUT) a framework which allows the engineer to program the "weaving" of the aspects at the level of the UML meta-model. In Section 4, we illustrate our aspect-oriented design (AOD) approach with a distributed multimedia application by giving parts of a weaver implementation. We then show how our approach fits into a reflexive viewpoint on the UML in Section 5, illustrating the way users may define transformations in UMLAUT. We discuss related work in Section 6 and conclude on the interest and perspectives of our approach.

2. DESIGNING WITH ASPECTS AND UML

The aim of this section is to extend the ideas expressed in aspect-oriented programming (AOP) [14] to the software modeling level. In [2], the authors explicit the gap that exists between requirements and design on the one hand, and between design and code on the other hand. AOP should then be extended to the modeling level where aspects could be explicitly specified during the design process. Indeed, we believe that with the support of an open transformation framework, it is possible to weave these aspects into a final implementation model.

We use UML as our design language because it is an open standard [22], as well as general purpose object-oriented modeling language. UML supports the concept of multiple views that allow a software designer to express various requirements, design and implementation decisions using each view independently. The design is founded on the meta-model of UML, ensuring the coherence of the various views. The extension features of UML also allows it to be customized for a specific modeling environment.

2.1 Expressing Aspects with UML

The various modeling dimensions of UML can already provide a good separation of concerns when modeling software. But in order to specify additional non-functional information or cross-cutting behavior (e.g. persistency), we need to resort to UML built-in

[1] This work is partly funded by the European QCCS project, IST-1999-20122.

extension mechanisms. Using these, the designer can add a great deal of non-functional information to a model by "hooking" annotations to model elements. Three annotation mechanisms are commonly used: stereotypes, tag values and design pattern occurrences.

Stereotypes can often be used to subtype a given model element type, e.g. for a class to specify that its instances should be persistent (see class *History* in Figure 1). Automatic tools can then identify this element among the other model elements of the same type and process them specifically.

Design pattern occurrence can be use to specify that a given design pattern shall be applied in a specific place in the model [19]. For instance, we may mark in Figure 1 each one of the operations of class *ServiceProvider* that should participate in the application of the *Command* design pattern [8]. In the same example, the class itself may be adorned with the design pattern occurrence. With this annotation, a weaver 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 the 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. This extension and annotation mechanism gives us the flexibility of modeling 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 this non-functional information.

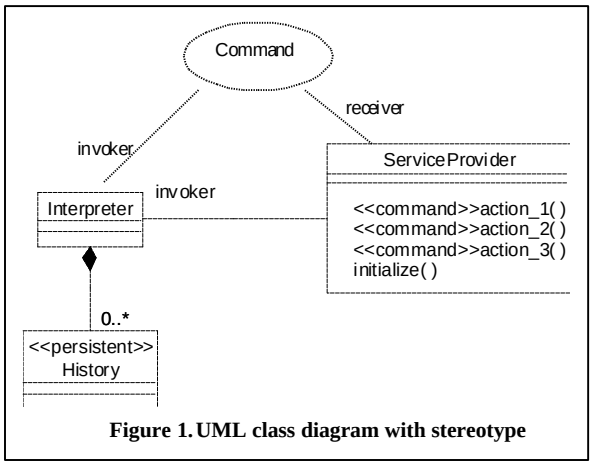

Figure 1. UML class diagram with stereotype

2.2 Aspects at Various Levels of Abstraction

As outlined above, transformations rank into two categories: the ones related to the application domain and those involved in generating efficient implementations for the target platform.

Let's take another example to illustrate the difference: if the designer knows a collection of objects has to be notified when another object changes, then she annotates the corresponding classes as collaborating into an Observer pattern. A generic transformation supporting this pattern adds an *update* method to every class playing the role observer in this pattern occurrence. Specific transformations for implementing the pattern offer designers choices that fit implementation trade-offs: execution speed vs. memory footprint, and point-to-point notification vs. broadcasting depending on requirements on the underlying hardware. This last transformation is not at all related to the application, and must not distract the designer from its application refinement.

The two categories are not exclusive : some transformations bridge the application domain and the implementation domain, thus falling into both categories. These transformations perform the ``weaving" of the two aspects into a single implementation model.

3. WEAVING UML DESIGNS

UMLAUT is a framework dedicated to the manipulation of UML models. Since UML is itself described by a meta-model in UML, manipulating the meta-model is the same as manipulating any model. Hence we deal with the weaving of AOD designs by handling the model at the meta-model level. To this aim we are developing an open framework where the weaving process can be adapted and extended: new weavers can be constructed simply by changing the weaving rules. The framework takes care of the weaver implementation. In our UMLAUT toolbox, a weaving process is implemented as a model transformation process: each weaving step is a transformation step applied to a UML model. Hence the final output is a UML model too (endomorphic transformation). The model transformation engine is itself designed as a configurable and extendible framework.

3.1 General Architecture and Core Engine

UMLAUT's architecture is a three-layered one. The input front end consists of a graphical user interface for interactive editing; another interface deals with importing UML models described in various formats (XMI, Rational Rose™ MDL, Eiffel source, Java source). The middle core engine is made up of the UML meta-model repository and the extendible transformation engine. Finally, the output back end contains various generators (including code generators and an XMI generator). 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 tool's behavior and to meet specific requirements.

3.2 The Extendible Transformation Framework

The transformation engine of UMLAUT is responsible for the weaving process. In an earlier article [11], we have shown that automated transformations of UML models can be used by a designer to derive different refined views of a given software model. We would like to further develop the idea on how it can aid in performing design level aspect-oriented weaving. A weave operation is described as a transformation of an initial model to a final one. A designer specifies the required transformation by

explicitly composing a set of operators from the UMLAUT transformation library. Since the transformation engine is an open framework, users may add new operators and extend the existing library to support new weaving operations. The framework is designed to cater for three different kinds of user:

Model designers are interested in performing a set of weaving operations. Their main concern is what transformation operators are available and useful to the model, and how they should be used.

Transformation architects are responsible for defining how to implement a given transformation for a given implementation requirement. They extend the transformation library by adding new transformation operators.

Framework implementers aim at enhancing the weaver framework to support specific needs of the previous two groups of users.

The transformation framework uses a mix of object-oriented and functional programming paradigms. The object-oriented paradigm allows us to encapsulate our operators as discrete entities, and the functional paradigm provides us with a composition mechanism for these operators. The main architecture consists of three major components:

1. A core structure that provides the logic for operator composition and implicit control flow when a transformation is initiated.
2. A library of iterators for traversing a UML model. An iterator builds a path through a UML model graph so that lazy list operations can be applied.
3. A library of primitive operators for querying, modifying and creating UML model elements.

Each of these components can be augmented and enhanced. In particular, the operator library is likely to be extended by an transformation architect whereas the iterator library will more likely be extended by a framework implementer knowledgeable about the UML meta-model[2].

4. A DISTRIBUTED MULTIMEDIA APPLICATION

As an illustration of how our framework weaves UML designs, we present in this section an application designed with aspects. Figure 2 shows a simple design of a distributed multimedia player. The *PLAYER* type defines an interface from which two implementations are derived: a proxy, *PLAYER_PROXY*, and a server implementation, *PLAYER_IMPL*. The *PLAYER_SUBJECT* type is the server side stub that relays the client requests to the implementation by means of a *Command* design pattern instance. This application is represented by a dotted ellipse with the word *Command*, and dotted lines indicate the role played by the objects involved in the pattern. As in Figure 1, operations in *PLAYER* participating in the *Command* design pattern are stereotyped with «*Command* ». Both *PLAYER_PROXY* and *PLAYER_SUBJECT* are stereotyped with « *REMOTE* » to indicate that the association

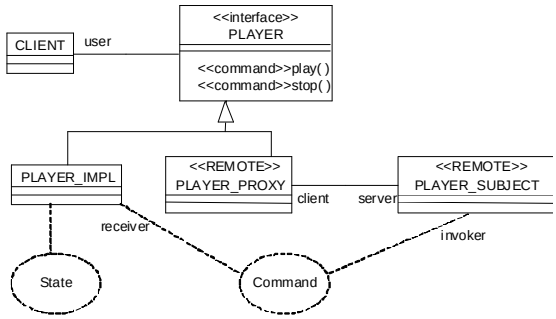

Figure 2. A Distributed Multimedia Application

between the two classes are related by a physical distribution medium. For the dynamic aspect, we include in *PLAYER_IMPL* a statechart to represent its behavior when responding to *play* and *stop* requests (0). To reify this behavior model, we annotate our design with the *State* design pattern.

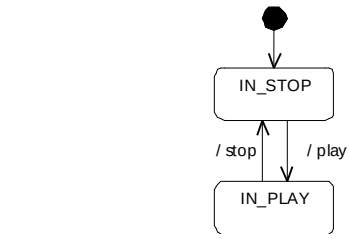

Figure 3. Statechart for PLAYER_IMPL

Once we have attached weaving information to some of our model entities, we need to generate an implementation model, e.g.; the implementation model of Figure 4 where the classes enclosed in the dotted rectangle on the right implement the *State* design pattern.

4.1 Weaving the Implementation Model

In order to generate the implementation model of Figure 4 from the design model of Figures 2 and 3, our weaver has to perform the two following steps, one for the *Remote* aspect, and one for the *Command* aspect.

1. First, we produce a concrete implementation model in UML using the « *REMOTE* » stereotype as a "guideline". In our example, we shall simply move the client association from the *PLAYER* interface to the *PLAYER_PROXY*. However, more transformation details are needed to produce a full implementation for « *REMOTE* ». These details include the choice of an underlying architecture. For example, if CORBA is chosen as a middleware layer then its built-in implementation of the *Proxy* pattern should be used.

[2] See details at http://www.irisa.fr/UMLAUT

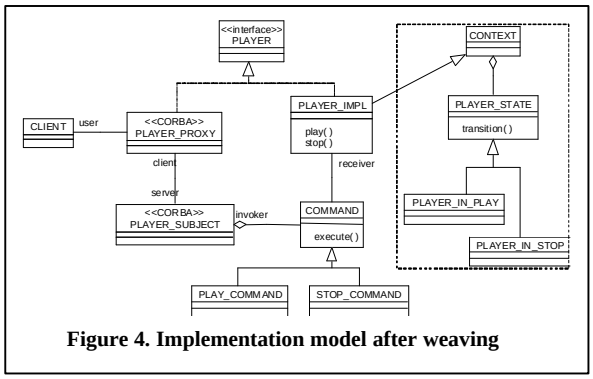

Figure 4. Implementation model after weaving

2. Second, we implement the *Command* pattern on top of the concrete implementation by attaching association roles to the participants of the pattern. In our example, we choose to add the *invoker* and *receiver* roles to the participants, and add a hierarchy of command classes for each of the operations marked with the «*COMMAND*» stereotype in Figure 2. It is important to note that there is often more than one way to implement a design pattern, and a discussion of the possible choices is beyond the scope of this example. In particular, our tool does not try to choose the best design pattern implementation but just to provide the designer with a framework that helps her to implement her choice.

We now explicit these two transformation steps using a composition of *map* and *filter* expressions. The expressions are evaluated on the model in order to transform it by adding, removing model items such as associations, attributes, or classes.

The ***map*** operator applies an operation on each element of an input sequence and returns the results' sequence. It should be noted that the operation applied on each element may have side effects and alter the element (which is an object).

The ***filter*** operator "lazily" returns a subsequence from an input sequence, retaining only the elements for which a boolean function yields true. This *filter* operation is purely functional (no side-effects).

Operators are concatenated using the composition operator noted "o" below. The first step is to move the association of *CLIENT/PLAYER* to *CLIENT/PLAYER_PROXY*. We use the map and filter operators with the two following transformations in sequence:

```
-- Find all model elements named Client and associated
 -- to a an interface playing a proxy role.

(map removeAssociation) o (filter isClient) allElements

-- Then move the user association down to the class
-- implementing this interface

(map associate) o (map getClntNProxy) o
      (filter isClient) allElements
```

The *allElements* term denotes the sequence of all model elements. Such a sequence is generated by an iterator (traversal operation). The filter operation is then applied to the element sequences, retaining only those that are clients on a class with the «*REMOTE*» stereotype. Lastly, the *map removeAssociation* expression removes all associations between the client class and player proxy.

The *isClient* and *getClntNProxy* operators are predicates developed specifically to pinpoint the AOD elements in our model. These application-specific operators may either be written from scratch or adapted from existing operators. On the other hand, operators like *removeAssociation* and *associate* are general library operators that can be reused for similar circumstances. It should be noted that the second expression (*map associate*…) is evaluated on the model obtained from the first expression evaluation (*map removeAssociation*…).

The second step is to create the command classes from the *Command* pattern and to add the role associations for the participants. We use the following code:

```
-- Make a new abstract class named Command
baseClass = newClass 'Command'

-- Find all classes playing a receiver role in the
-- Command design pattern instance, then for each of
-- them build the list of operations that are stereotyped with
-- Command  then transform these operations  into classes

(map map opToClass) o (map allCmdOps)
   o (filter isReceiver) all Elements

-- Find all classes that play an invoker role in the
-- Command design patterninstance, then associate each of
-- them with the Command class created above

(map associate baseClass) o (filter isInvoker) allElements

-- Do the same with the classes playing a receiver role

(map associate baseClass)  o (filter isReceiver) allElements
```

The *isReceiver*, *isInvoker* and *allCmdOps* operations are specific to the weaving of all *Command* design pattern instances, and they must be implemented by the weaver designer. As shown in Section 5, these operations are easily described in UMLAUT thanks to the full access to the UML metamodel.

The sequence of transformations described in this section becomes our weaver for composing the distribution model and command pattern into our implementation model. In other words an application is defined using separate design aspects, our weaving process is also designed using separate transformation steps like the ones described above.

4.2 Summary

From this example, we show that it is possible to develop an application specific weaver by redefining the transformations to be applied to a design model. The base transformation framework of UMLAUT provides the user with a set of primitive, general purpose operators that can be extended and reused for different application specific needs. Each aspect-oriented design may be developed with an application specific weaver that optimizes the weaving process. As illustrated in our previous example, an

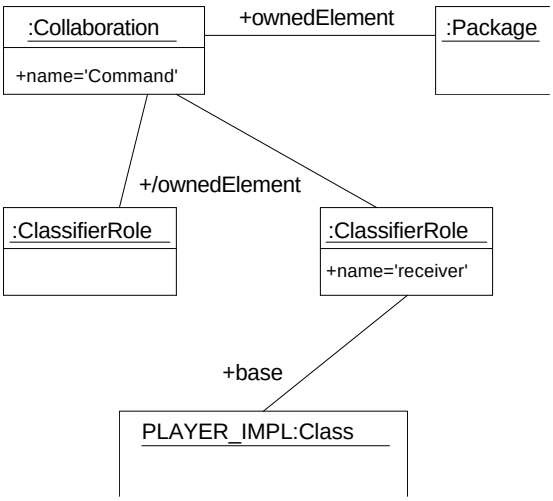

Figure 5. Simplified object model of Command collaboration

application specific weaver need not be developed from scratch. For instance, the weaver for *Command* design pattern instances needs to be implemented only once. Moreover, since our transformations operations (*map*, *filter*, etc) are implemented with classes, a transformation designer can easily build upon existing weavers to build new ones, using inheritance, delegation or any other object-oriented programming technique.

5. TOWARDS REFLECTION IN UML

In the previous section, we introduced a formalism based on set processing to express transformations on a UML model in a functional style. In a tool providing support for such a powerful mechanism of reasoning on models, it would be overwhelming for users if they had to learn new tool-specific languages for the description of these transformations. We advocate that the UML has enough expressive power to fulfil all our needs. In particular, the Object Constraint Language (OCL) [27] which is a standardized part of the UML is the language of choice for expressing the selection criterion of a transformation, as it was specifically designed to provide powerful constructs (such as *select*, *forAll* and other *iterate* operators) dedicated to collection processing.

Filter and *map* operations are easily realized with a *iterate* construct over a *Collection*, returning another *Collection* (or one of its derivatives: *collect*, *select*, etc).

Reduce operations are realized with a *iterate* construct over a *Collection*, returning a basic OCL type *i.e. Integer*, *Boolean*, *String*). Such predefined OCL operations already exists (*size*, *isEmpty*,…).

Writing transformations mostly consists in navigating through instances of UML meta-elements. 05 shows an extracted view of the application of the *Command* pattern depicted in Figure 4, in terms of instances of meta-model elements (and this is the way it is stored in the UMLAUT tool).

For example, retrieving applications of the *Command* pattern (which are *Collaborations*) in a *Package* may be realized with the following filtering operation declaration:

```
context Package::commands()

post: result = self.contents()->select(item:ModelElement |
  item.oclIsKindOf(Collaboration)) -
>select(name="Command")
```

Then finding *Classifiers* playing the role of a receiver in the *Command* pattern is done with the following *receivers*() operation, navigating through the UML metamodel, from *Collaboration* to *ClassifierRole*, then via *ClassifierRole* to *Classifier*:

```
context Package::receivers()

post : result = commands()->ownedElement ->select(name =
"receiver") ->base
```

An OCL interpreter integrated in UMLAUT performs the evaluation of these operations on a model.

However, most transformation operations on UML involves addition, modification or removal of model elements. These operations are not side-effect free and cannot be expressed with the OCL. To deal with this situation, we propose to describe actions with the help of the Action Semantics proposal which is currently being standardized at the OMG [22]. This proposal aims at formalizing the dynamic semantics of the UML, introducing in the UML metamodel classes such as *CreateAction*, *DeleteAction*, *CreateLinkAction*, *DeleteLinkAction*, or *AssignmentAction*, and strongly encourages the use of OCL.

The AS has originally been designed for precisely specifying the behavior of models. We advocate the extension of its scope beyond this basic role, to enable reflexivity in UML for both its static and dynamic definition.

An UML execution engine, i.e. an implementation of the AS model of execution is originally dedicated to the manipulation of instances of UML models (so called M0 level). Such manipulations are specified at the model level (so called M1 level), as part of the whole model of the application. But since both (1) the UML meta-model (M2 level) and (2) the UML execution model for the AS are themselves UML models, we can use the AS to specify the evolution of these models:

? In the first case, thanks to the four-level architecture of the UML, an AS specification would manipulate instances of M2 level, i.e. UML models. Then, an AS specification describes a *model transformation* (meta-programming).

? In the second case, an AS specification would manipulate instances of the execution model, i.e. the objects at runtime (a representation of M0 level called a snapshot). Then, the AS specification describes the transformation from one snapshot to the resulting one, that is the *semantics of the AS itself* (reflexivity applied to the execution engine specification).

Using the UML meta-modeling architecture and the Action Semantics for specifying transformations is appealing: the development of meta-tools capitalizes on experience designers have gained when modeling UML applications.

Some recurrent problems then disappear: portability of transformations is ensured for all UML-compliant tools with access to the meta-model, there is no learning-curve for the writing of new meta-tools, as it is pure UML and any development process supporting the UML applies to the building and reuse of transformations. This paves the way towards off-the-shelf transformation components.

6. RELATED WORK

6.1 Aspect and Subject-Oriented Programming

Adaptive programming [21], aspect-oriented programming [14], and subject-oriented programming [10] have taken software development beyond the class concept of object-oriented programming. They address explicitly additional dimensions that constitute the inherent complexity of software. We believe that these works at the implementation level can be broaden to the entire software cycle and lead to aspect-oriented design (AOD). The use of UML in the context of AO modeling is already evident in [13], [2], [3], [26] and [3] has proposed to explicit multi-dimensional concerns for the entire software development cycle. Our work aims at providing an automated tool to support the expression of aspects at the design model level. The provision of an open framework has the added advantage that the user can redefine weaving strategy by re-composing the transformation operations. Using transformations during the weaving process is demonstrated by [19] and [7]. Relative to their source code oriented approach, UMLAUT addresses transformation with a design oriented, meta-modeling approach.

In short, we use UMLAUT to apply aspect-oriented concepts for the entire software development cycle. We express weaving of software aspects in terms of model transformations. Its implementation as a framework makes it open for extension and customization.

6.2 UML Model Transformation

Using a functional programming paradigm in an object-oriented context has been proven to be a versatile technique (see [4], [18], [16]), especially when flexible composition and list-like processing are involved. The UMLAUT transformation framework has taken this idea to provide an extensible AOD environment. The main interest of this extensibility is the possibility of defining the weaving strategy by recomposition of primitive transformation operators. The transformation of software models is widely applied in tool automation for design patterns , software refactoring [23][24], equivalence transformations [9], [1], [25], and formal reasoning [17]. UMLAUT's transformation incorporates ideas from these works, and extends them to automate the definition of weaving operations in the context of AOD. In addition, UMLAUT exposes the concept of explicit model transformation to a software designer so that she can benefit from the versatility of this open approach.

7. CONCLUSION

We believe that aspect-oriented programming should be extended to the entire software development cycle. 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. We have applied this approach to the development of two real applications:

? A part of the Information System of a large Telecom Company, with the handling of aspects such as concurrency and persistency (taking into account various persistency frameworks that have been previously developed to interface commercial DBMS).

? The UMLAUT tool itself, that has been bootstrapped from the “official” UML 1.3 specification. Because the UML meta-model is expressed as a UML model, we could add many features such as model management (consistent creation/deletion of model elements), user interface connection or XMI generation as so many *aspects* that have been woven together by UMLAUT to build itself in a classical bootstrapping scheme.

8. REFERENCES

[1] Michael Blaha and William Premerlani. A catalog of object model transformation. In 3rd Working Conference on Reverse Engineering, november 1996

[2] 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.

[3] Siobhán Clarke and John Murphy. Developing a tool to support the application of aspect-oriented programming principles to the design phase. In ICSE '98 Workshop Proceedings on Aspect-Oriented Programming Proceedings, 1998.

[4] Laurent Dami. Software Composition: Towards an Integration of Functional and Object-Oriented Approaches. Ph.D. thesis, University of Geneva, 1994.

[5] Philippe Desfray. Automation of design pattern: Concepts, tools and practices. In Jean Bézivin and Pierre-Alain Muller, editors, The Unified Modeling Language, UML'98. First International Workshop, Mulhouse, France, June 1998, volume 1618 of LNCS, pages 107–114. Springer, 1998.

[6] Andy Evans. Reasoning with the Unified Modeling Language. In Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)}, 1998.

[7] Pascal Fradet and Mario Südholt. Aop: towards a generic framework using program transformation and analysis. In ECOOP'98 Workshop Proceedings on Aspect-Oriented Programming Proceedings, 1998.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[9] Martin Gogolla and Mark Richters.Equivalence rules for UML class diagrams. In Jean Bézivin and Pierre-Alain Muller, editors, The Unified Modeling Language, UML'98. First International Workshop, Mulhouse, France, June 1998}, volume 1618 of LNCS, pages 87–96. Springer, 1998.

[10] 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.

[11] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE, 1999.

[12] Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, Northeastern University, February 1995.

[13] Elizabeth Kendall. Aspect-oriented programming for role models. In ECOOP '99 Workshop Proceedings on Aspect-Oriented Programming Proceedings, 1999.

[14] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit 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.

[15] Philip Kruchten, The 4+1 View Model of Architecture, IEEE Software, Vol. 12, No. 6, November 1995.

[16] Thomas Kühne. Internal iteration externalized. In Rachid Guerraoui, editor, ECOOP '99 Object-Oriented Programming 13th European Conference, Lisbon Portugal, volume 1628 of Lecture Notes in Computer Science, pages 329–350. Springer-Verlag, New York, N.Y., June 1999.

[17] Kevin Lano and Juan Bicarregui. Formalising the UML in structured temporal theories. In Haim Kilov and Bernhard Rumpe, editors, Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications), pages 105–121. Technische Universität München, TUM-I9813, 1998.

[18] Konstantin Laufer. A framework for higher-order functions in C++. In USENIX Association, editor, Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS), pages 103–116, Berkeley, CA, USA, June 1995.

[19] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. -- Precise modeling of design patterns. – In Proceedings of UML 2000, volume 1939 of LNCS, pages 482--496. Springer Verlag, 2000.

[20] Anurag Mendhekar, Gregor Kiczales, and John Lamping. Rg: A case-study for aspect oriented programming. Technical report, Xerox Palo Alto Research Center, February 1997. Technical report SPL97-009 P9710044.

[21] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. ACM SIGPLAN Notices}, 33(10):97–116, October 1998.

[22] OMG. UML notation guide.

[23] William F. Opdyke. Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois, 1992.

[24] Donald Bradley Roberts. Practical analysis for refactoring. Technical Report UIUCDCS-R-99-2092, University of Illinois at Urbana-Champaign, April 1999.

[25] Siegfried Schönberger, Rudolf K. Keller, and Ismail Khriss. Algorithmic support for model transformation in object-oriented software development. Theory And Practice of Object Systems, 1999.

[26] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with aspects: Aspect support in the design phase. In ECOOP'99 Workshop Proceedings on Aspect-Oriented Programming Proceedings, 1999.

[27] Jos Warmer and Anneke Kleppe. The Object Constraint Language: Precise Modelling with UML. Addison-Wesley, 1998.