From Contracts to Aspects in UML Designs

Jean-Marc Jézéquel, and Noël Plouzeau
Irisa (INRIA & University of Rennes)
Campus de Beaulieu
+33 2 99 84 71 92
e-mail: {jezequel, plouzeau}@irisa.fr

Torben Weis, and Kurt Geihs
TU-Berlin (iVS)
Einsteinufer 17, 10587 Berlin,
+49 30 314 79830
e-mail: {weis, geihs}@ivs.tu-berlin.de

ABSTRACT

Separation of concerns is a basic engineering principle that is often applied in object-oriented analysis and design by dissociating functional aspects (business objects) from non-functional ones, such as persistency, fault-tolerance and so on. The Unified Modeling Language (UML) then 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 to add non-functional annotations to a model. In this paper, we explore a possibility to organize all of these features around the central notions of (1) quality of service *contracts* (for specifying non-functional properties *a la* QML) and (2) *aspects* for describing how they can be implemented. We propose to model contracts in UML with a small set of stereotypes, and to represent aspects a bit like design pattern occurrences, that is using parameterized collaborations equipped with transformation rules expressed with meta-level OCL2. The design level aspect weaver is then just a meta-level OCL2 interpreter, that takes a UML model as entry point, processes the various aspect applications as specified by the designers, and outputs another (detailed design level) UML model that can serve as a basis for code generation.

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. In this paper[1], we explore a possibility to organize all of these features around the central notions of quality of service *contracts* (for specifying non-functional properties) and *aspects* for describing how they can be implemented. In Section 2, we propose to model contracts in UML with a small set of stereotypes, and to represent aspects a bit like design pattern occurrences, that is using parameterized collaborations equipped with transformation rules expressed with meta-level OCL2. The design level aspect weaver presented in Section 3 is then just a meta-level OCL2 interpreter, that takes a UML model as entry point, processes the various aspect applications as specified by the designers, and outputs another (detailed design level) UML model that can serve as a basis for code generation. Section 4 gives an illustration of our approach on a toy example. We discuss related work in Section 5 and conclude on the interest and perspectives of our approach.

[1] This work is partly funded by the European QCCS project, IST-1999-2012

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 allow 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 extension mechanisms. Using these, the designer can add a great deal of non-functional information to a model by "hooking"

annotations to model elements. We propose to organize all of these features around the central notions of (1) quality of service *contracts* (for specifying non-functional properties *a la* QML [32]) and (2) *aspects* for describing how they can be implemented.

2.2 Modeling Contracts in UML

With the word *contract* we mean the specification of constraints that have to be met when some entity requests a service from another entity. Document[31] defines four different levels of contracts. The first level deals with parameter type compatibility, the second level with pre and postconditions, the third with synchronization constraints and the fourth with non-functional features such as request completion time, throughput and so on. In this paper we briefly recall our contract metamodel for this fourth level.

Our model is based on the QML concepts for expressing non-functional values, adapted to fit component-based designs. From the point of view of an application designer, contract types are entities used in class diagram to state non-functional constraints. A contract type is an instance for the Contract meta entity that we have added to the UML metamodel. This Contract entity is a specialization of the Interface standard entity of the metamodel. A contract type contains dimensions. Each dimension can be seen as an axis of some quality measure. Hence a contract type defines a quality space. At run time contract instances describe a point in that quality space or a subspace defined with comparison operators.

Our fourth level contract also takes contract dependencies into account. In real life, service providers (components, objects, methods) declare that they will comply with a given contract type if and only if their environment does comply with a set of contract types. For instance, a bound on a request execution time is valid if and only if the maximum network delay is less than 100 milliseconds. So with respect to a contract aware piece of software, we distinguish between provided contracts and required contracts (this notion is similar to the distinction between pre and post conditions). We use the UML dependency association to express this provided/required relationship in UML models.

2.3 Realizing Contracts with Aspects

An aspect can be used to show how a contract is implemented. Typically the pieces of the design that realize a certain contract are scattered across the entire design. That is the reason why AOP is an appropriate solution for separating the implementation of a contract from the rest of the model. At some point in time we have to weave the aspect with the target UML model to produce the final implementation model. Therefore, the aspect weaver has to know where exactly the aspect touches the target model. That brings us to one of the important AOP topics, the question of how to define join-points.

Before we talk about the *how* it has to be resolved *where* join-points are defined. In typical AOP extensions of programming languages the join-points are usually part of the aspect. That means the aspect weaver needs to know about the target and the aspect and then it can start weaving. In our approach that is a bit different, since this approach couples the aspect tightly with the target model. There are little chances to reuse the same aspect for another target model. We decided to remove this shortcoming. Therefore, the definition of join-points is no longer an integral part of the aspect itself. Instead they form a third entity that connects the target model with the aspect. To some degree that blurs the borders between aspects and parametric collaborations.

Since aspects do not have hard coded join-points anymore we have to pass them as arguments to formal parameters of the aspect. In AspectJ join-points are related to constructs of the target language such as functions, classes, etc. Since our aspects live in the context of the UML our join-points are elements of the UML meta-model. Each instance of a meta-class may be passed as an argument to the formal parameter of an aspect. These formal parameters have a type: a meta-class of the UML meta-model. Thus, the formal parameters of an aspect are conceptually very close to the signature of functions in procedural languages like C. The type of the parameters can be automatically derived since each parameter has a corresponding element inside the aspect's definition. Otherwise the type has to be stated explicitly in the aspect's signature. Figure 2 show the three entities together in one diagram: target model, join-points and aspect.

In order to provide support for more complicated aspects a simple list of formal parameters is not sufficient. Imagine that the formal parameters of an aspect are a class and a method. To make it more complicated we want to pass one class and multiple of its methods as join-points (respectively: arguments) to the aspect. That is impossible with a flat list of formal parameters. We allow for the definition of 1:*n* relationships between parameters. In figure 2 this is illustrated by the curly braces around the second formal parameter. The second parameter is in fact a set.

It is important to notice that an aspect has now a well defined interface that explains how it can be woven with the target model. In order of using an aspect that is all a designer needs to know. On the other hand, the implementation is entirely encapsulated inside the aspect. That in turn means that it is possible to reuse an aspect with other – comparable – target models.

In the next chapter we discuss how we can actually define an aspect, because until now we did more or less only declare an aspect and showed how to bind it to the target model. The modeling elements inside the aspect can be thought of as a meta-program that is interpreted by the aspect weaver.

3. WEAVING UML DESIGNS

Kase is a tool for drawing UML models. Unlike other tools it does not enforce the concept of certain diagram types. Everything that is allowed by the meta-model can be drawn. This is especially important for modeling aspects since they do not fit in any existing diagram category. Additionally, Kase can import UML profiles and provide enhanced notation for the stereotyped UML elements. Thus, we can provide a new notation for aspects while we still rely solely on standard UML extension mechanisms.

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 a meta-level OCL2 interpreter (connected to Kase) where to execute the weaving operations as specified in each relevant aspect applied onto the initial model.

We advocate that the UML has indeed 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.

Writing transformations mostly consists in navigating through instances of UML meta-elements. 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 version 1.0. To deal with this situation, we propose to describe actions with the help of the newest version of the OCL, based on the OCL2 proposal which is currently being standardized at the OMG. This introduces in the UML metamodel classes such as *CreateAction*, *DeleteAction*, *CreateLinkAction*, *DeleteLinkAction*, or *AssignmentAction* that can be used as primitive operations for model transformations.

The weaving process is thus 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. Using the UML meta-modelling architecture and OCL2 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.

Using OCL2 as a transformation language is a very powerful choice. However, many developers know how to use most of the UML concepts, but they do not know about the UML meta-model. This knowledge is crucial to write correct OCL expressions. Thus, it would be helpful to provide a notation for the transformation that is close to the standard UML notation and does not require knowledge of the meta-model. We do not intend to provide just a graphical notation for OCL expressions like in [27]. That is just tweaking the notation without simplifying anything. Aspects for programming languages such as AspectJ consist to a large degree of normal program code enriched with some AOP-specific bells and whistles. We propose to follow the same path when modelling design aspects. If for example an aspect consists of two methods that have to be woven with some class, then it seems straight forward to show these two methods in standard UML style inside the aspect's notation. We base our ideas on work that has already been done in the field of modelling collaborations and model composition [29]. In the next chapter we provide a small example that illustrates how this notation of an aspect can look like.

In our approach we combine the benefits of both approaches. Using OCL2 we have full control of the model transformation and we do not need any proprietary aspect weavers since the weaving can be done with an standard OCL2 interpreter. On the other hand we have a smarter notation that makes aspects easy to design and to understand. The trick is to compile the information given by the graphical notation into OCL2 expressions. The graphical notation is used during the design of the aspect's implementation but it is not directly executed. We execute the OCL expressions instead. Additionally, these OCL expressions can be used to exchange aspects between developers and different UML tools since they rely solely on standard (or soon to be standardized) UML technology.

For the Kase tool AOP is just an additional UML profile together with a module that provides the new notation. So internally the model consists of standard UML with some stereotypes and tagged values hidden behind the new notation. The OCL2 counterpart is executed by the UMLAUT framework. Since Kase uses internally an observer pattern the user can follow the changes of the model step by step. In contrast passing the aspects and target model as XMI to some command line tool and reading in the resulting XMI document is less comfortable, since the resulting XMI document won't include any layout information. Furthermore, debugging an aspect is much easier if you can follow the weaving process step by step on the screen.

4. EXAMPLE

In this Section we want to illustrate our ideas with an example. Figure 1 shows one component. It realizes two interfaces. In addition a non-functional contract is associated with the component. Contracts are depicted like a convoluted sheet. The compartment below the name is used for QoS-dimensions. In our example a performance contract defines the maximum response time for a function call in seconds. The little arrow in front of the dimension indicates that a smaller value is better than a larger one.

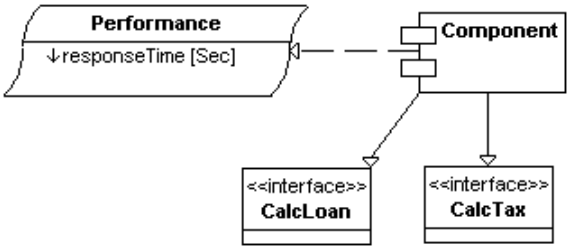

Figure 1: A component with contract and interfaces

The diagram shown in figure 1 is completely declarative. It does not show how the non-functional contract is implemented. Our intention is to separate the non-functional aspect from the remaining design using aspects. Figure 2 shows how that can look like.

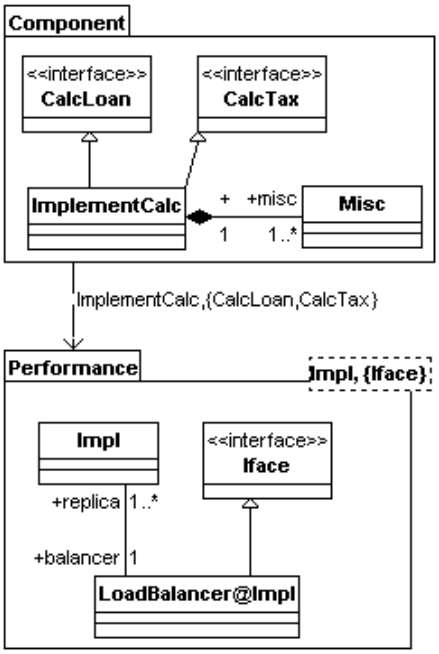

Figure 2: Implementation with aspects

The package called `Component` contains the implementation of the aspect. In our example there is one class that implements the interfaces and one helper class. To achieve a certain performance our aspect introduces the concept of load balancing. The load class that implements the load balancing has to offer the same interfaces as the component. Furthermore it uses the original implementation as a kind of replica. That means the aspect weaver will perform basically three actions:

1. Insert the new class called `LoadBalancer`.
2. Create a generalization between each supported interface and the load balancer class.
3. Create an association between the load balancer and the original implementation class.

The two classes called `Iface` and `Impl` correspond with the parameters of the aspect. Parameters are shown in a dotted box on the top right corner of an aspect. That means they are just placeholders for the classes that have to be passed to the aspect.

As denoted by the curly braces around `Iface` in the aspect's parameter list, we may pass multiple interfaces for one implementation class. This leads to the problem of multiplicities. The aspect weaver can not know whether it should create a new `LoadBalancer` class for every interface or one for every implementation class. This problem does not appear in simple parameter lists. To solve this we have to provide the aspect weaver with additional information. By appending `@Impl` to the name of the load balancer class, we indicate that the multiplicity for this new UML element is the same as for the parameter `Impl`. The multiplicities for the new generalization and the new association can be derived automatically from the multiplicities of the classes they connect.

The aspect is now transformed into a set of OCL2 expressions, which are then executed. The result is shown in figure 3. The `LoadBalancer` derives from both interfaces and is connected via an association with the `ImplementCalc` class.

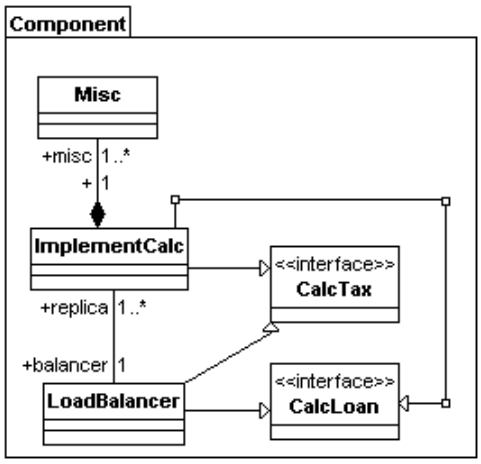

Figure 3: The final model

In this example we have shown how aspects can be used in class diagrams. The same principles can be applied to all other kind of UML diagrams such as state charts and activity diagrams. For example we can pass two states to the formal parameters of another aspect. This aspect can then add new states and transitions to a state chart. This way aspects can be used to model behaviour, too.

5. RELATED WORK

5.1 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 meta-level interpreter has the added advantage that the user can define weaving strategies by 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-modelling 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 meta-level OCL2 interpreter makes it open for extension and customization.

5.2 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.

6. CONCLUSION

We have proposed to organize non-functional and cross-cutting behavior in UML model around the central notions of quality of service *contracts* (for specifying non-functional properties) and *aspects* for describing how they can be implemented. Based on QML concepts, we propose to model contracts in UML with a small set of stereotypes, and to represent using parameterized collaborations equipped with transformation rules expressed with meta-level OCL2. The design level aspect weaver presented is then just a meta-level OCL2 interpreter.

With this possibility of modeling aspects and contracts in UML, we believe that aspect-oriented programming core ideas could be extended to the entire software development cycle. Each aspect of design and implementation should be declared as a contract linked to the business model, and expanded during the design phase so that there is clear traceability from requirements through source code.

We are currently developing a proof of concept tool, connecting the UMLAUT tool from INRIA and the Kase tool from TU-Berlin in the framework of the IST Quality Controlled Component based Software engineering (see http://www.qccs.org).

7. 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.

[28] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer, A Visualization of OCL using Collaborations. – In Proceedings of UML 2001, volume 2185 of LNCS, pages 257-271. Springer Verlag, 2001.

[29] Siobhán Clarke, Robert J. Walker. "Composition Patterns: An Approach to Designing Reusable Aspects" In proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, May 2001

[30] Meyer, B., *Applying Design by Contract.* IEEE Computer Special Issue on Inheritance and Classification, 1992. **25**(10): p. 40-52.

[31] Beugnard, A., et al., *Making Components Contract Aware.* IEEE Computer Special Issue on Components, 1999. **13**(7).

[32] Frolund, S. and J. Koistinen, *Quality of service Specification in Distributed Object Systems.* Distributed Systems Engineering Journal, 1998. **5**(4).

[33] Weis, T, Becker Ch., Geihs, K. and Plouzeau, N., An UML Metamodel for Contract Aware Components, Proc. of UML'2001 conference.