View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 163 (2006) 19–29

www.elsevier.com/locate/entcs

a(MDÆ)^2: A Model Driven Approach to Multi-Dimensional Separation of Concerns with OCL

Hans Schippers* and Dirk Janssens

Formal Techniques in Software Engineering
University of Antwerp, Belgium
{hans.schippers,dirk.janssens}@ua.ac.be
** Research Assistant of the Research Foundation - Flanders (FWO)*

Abstract

A typical activity in the object-oriented software engineering process involves the construction of a class structure in terms of which the system behaviour is to be specified. The behaviour, however, commonly consists of multiple tasks, each of which usually needs only part of the information available in that class structure. Additionally, a different representation of the required information may be appropriate. Therefore, it would be useful to be able to have multiple views on the global class structure, each being suitable for the specification of the behaviour related to a certain task. This paper introduces *a(MDÆ)*^{2}, a technique to realise such a strategy. It incorporates OCL as a powerful query language and advocates a model driven development process which relieves the developer from the burden of manually writing a considerable amount of tedious and error-prone code.

Keywords: MDE, Role Modeling, Views, MDSoC

1 Introduction

The use of object-oriented techniques in software engineering yields quite a few advantages, among which understandability and maintainability. More specifically, the fact that reality can be described intuitively in terms of communicating objects can be taken advantage of. Using D. Norman's terminology [8], the *mental model*, which is the representation of reality in a person's mind, could be considered to have object-oriented characteristics. Therefore, the gap between the mental model and an OO software model should be smaller than in other approaches. However, in practice the gap can still be inconveniently large. Indeed, the system encapsulates several *concerns*, a term defined very broadly (and close to its dictionary meaning) as "something" of importance to "someone" involved in the system. More concretely, each use case or task the system is supposed to carry out, can be seen as a concern. In order to maximise understandability and maintainability, the software system

1571-0661 © 2006 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2006.07.007

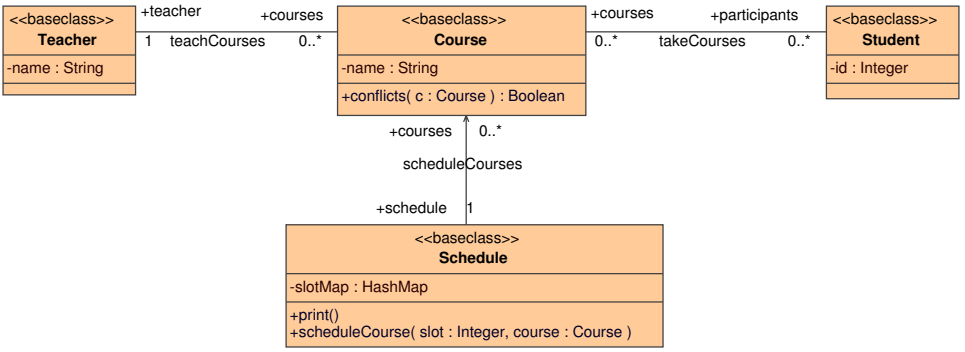

Figure 1. OO software model of a course scheduling application

should be modularised in such a way that different concerns can be specified as independently as possible, a principle often referred to as *separation of concerns* [24].

Every concern involves certain communicating objects, abstractions of which should correspond to entities in the software model. Unfortunately, the abstractions which are fit for one concern, are not necessarily equally adequate for another one. For example, in a real estate context, one concern might need to distinguish between properties based on their price class, while another one could care more about their vacancy. Therefore, a model of the complete system should actually be seen as some kind of compromise, an attempt to provide as convenient abstractions as possible for each concern at the same time, without introducing any ambiguities. In other words, one object-based decomposition must be chosen which has to do for the specification of all tasks, a phenomenon also known as the "tyranny of the dominant decomposition" [6]. This is why, according to R. Brooks, the difficulty in program comprehension lies in the reconstruction of the mapping between the problem domain and the program domain [3] or, equivalently, between the mental model and the software model. Thus, the notion of separation of concerns was later extended to *multi-dimensional separation of concerns* [32] to stress the fact that, ideally, decomposition criteria used for one concern should not influence the decomposition criteria for another one.

Consider a very simple OO software model of a course scheduling system [17], as shown in figure 1. It contains entities such as courses, students participating in courses, and teachers. The purpose of the system, as its name suggests, is to produce a schedule where each course is assigned a time slot so that no two courses which are either taught by the same teacher, or taken by the same student, are scheduled at the same time. Suppose the developer responsible for implementing the scheduling task would like to do so by means of a graph colouring algorithm. Such an algorithm is of course expressed in terms of nodes and edges, where in this case nodes correspond to courses, while edges correspond to a combination of two courses which should not be assigned the same time slot. Each node will then be assigned a colour, so that no two nodes connected by an edge end up with the same colour. Finally,

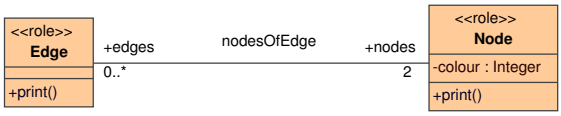

Figure 2. OO software model for the graph colouring task

courses of which the corresponding nodes are assigned different colours, should also be allocated different time slots in the course schedule.

While the entities in the OO software model of figure 1 are probably adequate for tasks like subscribing students to courses or assigning teachers the courses they should teach, the model is clearly suboptimal in the context of the scheduling concern. Indeed, the mental model of the latter makes abstraction of the fact that there are courses involved, and should include "Node" and "Edge" entities instead. Therefore, a model such as the one displayed in figure 2 would probably be more appropriate.

In conclusion, an interesting way to improve program comprehension would be to allow the specification of tasks (i.e. behaviour) in terms of the entities most adequate for that specific task, thus endorsing the idea of multi-dimensional separation of concerns. This should have a double positive effect on understandability, as not only would it narrow the gap between mental and software model, but the implementation of the behaviour of a certain concern would result in easier code as well if it can make use of an optimal software model for that concern.

Although the idea of using multiple models to match different concerns is not necessarily new (as will be pointed out in section 3), the *real challenge* is in making sure the cost of the extra work which is required to specify how all these models are related to each other does not outweigh the benefits of the concept.

This paper presents a(MDÆ)^2, a model driven, OCL-based approach aiming to support multi-dimensional separation of concerns, while minimising additional complexity. It is structured as follows: First, the proposed solution will be discussed in detail and illustrated by means of the course scheduling example which was already briefly introduced above. Next, there will be an overview of related work, finally followed by some issues to take care of in future work as well as some concluding remarks.

2 Proposed Solution

2.1 Conceptual

As mentioned earlier, the basic idea in a(MDÆ)^2 is that behaviour related to a certain concern should be specified in terms of the entities which are most suitable in that case. But while there is a clear separation of concerns with regards to behaviour that way, the same can not be said about data. Indeed, it is obviously inevitable that considering data, there is a substantial overlap between all the concerns. After all, together they do constitute one and the same software system and if there were several completely separated data structures, that clearly would not be the case.

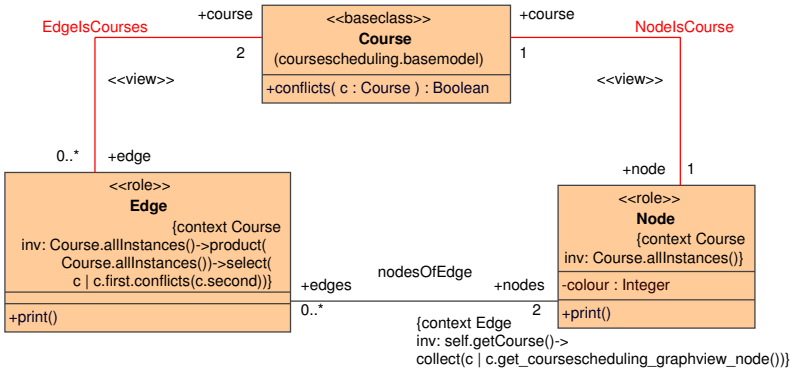

Figure 3. OO software model for the graph colouring task

Consequently, it would seem like a good idea to have one centralised OO decomposition of the problem domain (call this the *base model* (BM)), and define any other decompositions as derivations hereof. That way, each derived entity is a *view* on one or more entities of the base model, much like the (updatable) view concept in (OO) databases [29]. The derived decompositions will serve as a starting point for the specification of behaviour and shall be called *role models*. So each entity in such a role model, apart from defining a view on the BM, is also a *role*, meaning that it directly takes part in the behaviour specification related to the concern associated with that role model. This role is "played" by the BM entities which are involved in the view. It is worth noting that a role model is not necessarily a decomposition of the entire problem domain. Rather, it might ignore certain entities if these are judged to be irrelevant in that case. This, of course, does not hold for the base model.

Recalling the course scheduling example from section 1, the OO software model in figure 1 could serve as the base model, since it contains all entities necessary for the specification of the complete application. The model for the graph colouring concern would then be a role model, of which the entities are all defined in terms of BM entities. This is shown in figure 3; Since both BM and role models are essentially class-based OO models, a(MDÆ)^2 supports UML [22] as its modeling language, especially since the latter includes an extension mechanism in the form of *profiles*. A profile is a collection of stereotypes and/or tagged values, which conceptually extend UML model element definitions for a specific purpose. The $<<$baseclass$>>$ stereotype, for example, denotes that a UML class is part of the BM, while in role models the $<<$role$>>$ stereotype is used to indicate that an entity is in fact derived from the BM. Exactly which BM entities are involved with a certain role, can be deduced from associations marked with the $<<$view$>>$ stereotype.

The view relation between a role model entity and the BM is expressed as an OCL 2.0 [23] query and modeled by means of a UML constraint, as can be seen on the "Edge" and "Node" roles. Associations between roles are defined by a similar query, where special methods are provided in order to navigate to the BM and back if necessary. The "nodesOfEdge" association, for example, declares that each edge,

```
// subscription phase
Course c = new Course(“mathematics”);
...
Student s = new Student(“s854352”);
c.subscribe(s);
...
// graph colouring phase (e.g.  in a method in Schedule class)
Collection nodes = GraphViewManager.switchTo();
for (Iterator i = nodes.iterator()) {
   Node n = (Node) it.next();
   n.setColour(...); // calculate appropriate colour for each node
}
```

Figure 4. Example code for the course scheduling example

which is mapped to two courses c1 and c2 in the BM, should be associated with the two nodes which map onto c1 and c2.

An advantage of the use of OCL is the fact that it provides the ability to express complex view definitions. This is illustrated in the case of the “Edge” role, where the OCL query states that an edge is a tuple of two courses (denoted by the “product” operator, which has cartesian product semantics), which satisfy a certain condition. This is a significant advantage over approaches where only trivial mappings, like one-to-one mappings between a role and a BM class, are allowed (for examples of such approaches, see section 3).

Note that, so far, only platform-independent models have been mentioned. From these models, source code for a specific platform (e.g. Java) can now be generated using a code generation process on which more details will be given in section 2.2. Hence, a(MDÆ)^2 fits well into the model driven engineering paradigm. The most important consequence of this is that the developer does not need to worry about the details of how the mappings between role models and BM are maintained, and can just concentrate on specifying the behaviour in terms of the entities he finds most appropriate.

The developer, however, does have the responsibility to activate the role model he would like to work with by means of a method call to a special *switch manager* class. Behind the scenes, this switch manager will assure that the mappings to that specific role model are up-to-date, and that instances of the involved roles are made available. Chances are that the required information to perform these *view switches* in a fully automated way won’t be available most of the time. It could for example be convenient to use a different variable name for a collection of graph nodes, than for a collection of courses. Besides, nothing guarantees that a collection of all courses is even used during the “subscription” task. On the other hand, it may be useful to pass a certain role instance from role model A to role model B as a starting point (after applying a different wrapper of course), as opposed to just “forgetting” all variables and forcing the execution of queries on the new role model in order to continue. Therefore, even though an investigation of which possibilities for switching to a new role model should be offered is considered future work, the

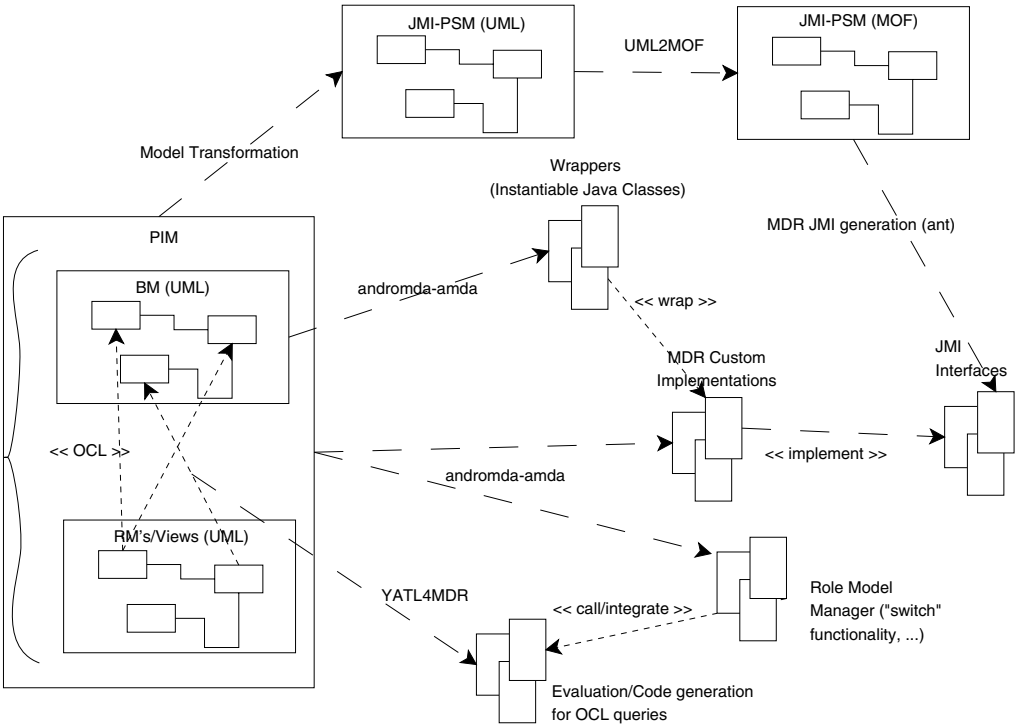

Figure 5. Code Generation Process

introduction of an explicit method call seems warranted.

An example of such a *view switch* can be found in figure 4, where, first, a number of students are subscribed using the BM entities. Next, whenever the developer decides to implement the graph colouring part, a switch call (marked in bold) is executed, after which the appropriate roles are made available.

Note that object creation statements (containing the *new* keyword), are only allowed for BM entities. Roles are automatically created during a view switch, depending on the current state of the BM. This means that creation of BM entities may have an indirect influence on the results of the view switching process.

2.2 Implementation

In order for a(MDÆ)^2 to provide the code generation capabilities already briefly mentioned in the previous section, a tool chain was constructed. An overview of the tools involved in this process, as well as the input they require, is displayed in figure 5. A first important observation is that, in order to support a querying mechanism, some sort of database-like repository is needed to store all objects, as well as a query engine compatible with that repository. For this purpose the YATL4MDR OCL engine [7] was chosen, which operates on Sun's MetaData Repository (MDR [19]). The latter is in fact a model repository, typically used to hold models and metamodels, which reside at levels M1 and M2 of the MOF metadata architecture [21]. However, nothing prevents it from storing objects at the instance-level (M0),

as is needed in this case. The M1-model describing the structure of these instances is then treated as if it were an M2-metamodel, that is, a MOF instance. Since MOF is basically a subset of UML, and includes all constructs which are necessary in this context, this causes no problematic consequences.

Being a MOF compliant model repository, MDR expects a MOF compliant XMI format as its input. Therefore, the BM as well as all role models, which are in fact UML models, are processed by the UML2MOF transformation tool which is included in the MDR distribution and handles this conversion. Afterwards, MDR is instructed to generate Java interfaces in order to provide program-level access to its content, by means of the JMI standard [18]. As it would not be desirable to expose the developer to these kinds of specifics, a(MDÆ)^2 includes a cartridge for the AndroMDA [2] code generator, which is responsible for the generation of wrappers, effectively shielding this complexity.

Finally, a *switch manager* class is generated for each view, which provides the view switch call, and is responsible for executing OCL queries (by calling the YATL4MDR OCL engine) and processing the results.

Note that this is where the model driven aspect really pays off. Requiring a developer to write all the wrappers and the manager class himself, would come at an unacceptable cost. By introducing a number of models at a high level of abstraction, a considerable amount of technical details related to view management can be deduced automatically, and incorporated in the generated code.

Also note that the main consequence of using OCL/MDR as a query mechanism, is that the instances of all modeled entities (classes from the base model, as well as roles) reside in a repository during the whole program lifecycle.

3 Related Work

The concepts of multi-dimensional separation of concerns and hyperspaces [32], which originated in the subject-oriented [10] community, are basically an attempt to tackle the same fundamental problem as the one discussed in this work. The idea is that an appropriate class hierarchy is constructed for each concern, which serves as a base for the specification of the business logic relevant to every concern. This way, the problem domain is decomposed several times, while concentrating on different priorities (one could say the system is decomposed along multiple axes, hence the term multi-dimensional). A meta-language is used to indicate which entities in the different concerns match. This information is then fed as input for a compilation step, where everything is woven together, resulting in a complete implementation of the whole system. From a conceptual point of view, the meta-language actually describes the overlap between concerns. Harrison *et al.* also implemented their ideas in tools such as Hyper/J [32] and the Concern Manipulation Environment (CME) [11], which are often classified as belonging to the field of aspect-orientation [15]. The main difference with more conventional aspect-oriented tools such as AspectJ [14] lies in the fact that AspectJ distinguishes between a base program and aspects. More specifically, an aspect, which describes a concern, contains

information as to how it should be "attached" to the base program. In the case of Hyper/J and CME, on the other hand, no distinction is made between the concerns, and a separate artifact is used to describe the weaving. In other words, the first strategy is asymmetric, while the other one is symmetric.

Looking at the basic problem as described in section 1, the idea of specifying each concern separately and independently, and then weaving them together based on where they overlap, does seem to be a natural candidate solution. Consequently, it should not be surprising that many others have concentrated on variations of this approach. S. Clarke [5] for example, shifts the idea to a higher level of abstraction, above the code. In an MDA [20] context, this could be referred to as the PIM-level. She uses UML as the modeling language, and a combination of template parameter binding and explicit composition directives to drive the composition process.

R. France *et al.* [27], also operate at an abstraction level above the code, and apply the idea to non-functional concerns (i.e. not related to business logic), such as security, fault tolerance or safety. They basically extend the UML metamodel to include e.g. security-related metaclasses, which they call *roles*. Afterwards, the business logic can be "annotated" to include security functionality, by indicating which metaclasses should be instantiated in the business logic.

The term "roles" was borrowed from the concept of *role modeling*, arguably introduced by T. Reenskaug [30], although there is quite some more work on this topic [26,16,25,9]. The idea is that concerns are specified in terms of roles, which represent the relevant entities in that case. These roles are then "played" by OO classes, thus integrating them to become a whole, but the exact way the role models should be composed is more often than not left undiscussed.

The main problem with the "weaving" approaches mentioned so far is, perhaps not unexpectedly, the weaving specification itself. Typically, a fixed set of constructs (called meta-language above) is made available to express the relations between all concerns. However, this often lacks flexibility, especially when things get more complicated than identifying two matching entities belonging to different concerns. This has been recognised among others by M. Mezini and K. Ostermann, who took this opportunity to develop the Caesar approach [17]. The most important difference compared to the previously mentioned approaches is the fact that the concerns are no longer compiled away, but maintained at runtime. More specifically, speaking in terms of roles, roles are present in the form of wrappers, and during program execution, objects are wrapped and unwrapped depending on which task is being accomplished at that moment, and which roles were defined for it. This is actually very similar to what is called *fluid AOP* by G. Kiczales [13]:

> "Fluid AOP involves the ability to temporarily shift a program [...] to a different structure to do some piece of work with it, and then shift it back."

The most important disadvantage of Caesar, however, is the fact that the developer is supposed to write and handle wrappers all the time, at what could be called a

rather low level of abstraction. There is other work following a comparable, dynamic approach to role modeling [25,31,12], but most of these fail to support flexible mappings between roles and objects. In the example on course scheduling for instance, the scheduling task is implemented by means of graph colouring. "Edge" is an obvious candidate for a role in that case, but its mapping is less straightforward: An edge role is played by a combination of two course objects which are in conflict with each other. While Caesar apparently supports this level of complexity, the other work mentioned above does not.

The only work to the author's knowledge which even mentions a model driven strategy, is by O. Caron *et al.* [4], where EJB is used as a target platform, but it is not elaborated. Moreover, it does not support flexible mappings once again.

Finally, the concept of (updatable) views is also present in the world of OO databases [28,29], and an OODB system could therefore be seen as a viable alternative to an OCL engine as far as handling of view relation queries is concerned. However, the number of implementations is very limited, let alone open source, and at best they lack sufficient flexibility.

4 Future Work

As indicated earlier, a topic which should be investigated in the relatively short term, is the flexibility of view switching. In theory, just performing the switch in the repository is sufficient, provided the developer has access to the OCL engine, since that would allow him to collect the necessary role objects from the new role model, and continue. However, it may be possible to reduce the number of useful queries in such case to a few categories, and just provide different methods for these in the switch manager class. After all, offering unrestricted access to the OCL engine would allow for the execution of any query, potentially defeating the idea of concentrating on one role model for each concern.

A different, slightly related, but more fundamental question is whether the fact that only one role model is activated at a time, would not lead to problems during the creation of certain kinds of software systems. Especially in cases where different tasks influence each other significantly, it may be necessary to have several active role models at the same time. Typical non-functional aspects, such as security, persistence or logging come to mind here. In that perspective, a weaving approach might still be the better way to go. Then again, the two strategies might turn out to be complementary, rather than mutually exclusive.

Another issue concerns object deletion. Indeed, since objects reside in a repository all the time, they are ignored by the garbage collector, and an explicit call is needed to actually remove them. This is somewhat unfortunate, as it causes a divergence from the classic Java programming model. On the other hand, it could turn out to be an advantage, considering it may be cumbersome to keep the base model entities in scope when working with role model entities.

Also, inheritance has not been mentioned so far, and although its incorporation would not necessarily cause many additional problems, it still deserves a closer look.

It is worth mentioning that, as an alternative to YATL4MDR, the Kent OCL Engine [1] includes a bridge to plain Java, effectively enabling the evaluation of OCL constraints without requiring a repository. However, the authors mention several serious issues, such as the lack of a translation of the `allInstances` operator since there is no easy way to fetch all instances of a Java class. Code generation may once again offer a solution though, because it could provide some bookkeeping code in order to keep track of all role and base class instances. This would eliminate the need for MDR as well as the base class wrappers, and would probably have a beneficial effect on performance.

Finally, at some point, larger case studies should be applied in order to validate whether the effort of software development is really reduced, or at least if it enhances comprehensibility, which should then reduce evolution efforts.

5 Conclusion

This paper presents a(MDÆ)^2, a model driven approach to multi-dimensional separation of concerns which allows for the specification of the behaviour of each concern in terms of the entities which are judged to be most appropriate for that concern. To this end, one base model (BM) is constructed, quite similar to a typical OO class structure, as well as several role models, which define views on the BM.

The main contributions of this work are in the combination of:

- Support for flexible view mappings, by means of OCL queries
- A model driven development process, where code generation shields the developer from the complexity introduced because of the querying support

In the end, hopes are that the a(MDÆ)^2 approach will allow for better code comprehension and a reduced effort to write programs in the first place.

References

[1] David H. Akehurst and Octavian Patrascoiu. Ocl 2.0 - implementing the standard for multiple metamodels. *Electr. Notes Theor. Comput. Sci.*, 102:21–41, 2004.

[2] M. Bohlen. AndroMDA - from UML to Deployable Components, version 2.1.2, 2003. http://andromda.sourceforge.net.

[3] Ruven E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[4] Olivier Caron, Bernard Carre, Alexis Muller, and Gilles Vanwormhoudt. A framework for supporting views in component oriented information systems. In *International Conference on Object-Oriented Information Systems*, volume 2817 of *Lecture Notes in Computer Sciences*, pages 164–178, Geneva, Switzerland, September 2003. Springer Verlag.

[5] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 325–339. ACM Press, 1999.

[6] M. D'Hondt and T. D'Hondt. The tyranny of the dominant model decomposition, 2002.

[7] Remco Dijkman. YATL4MDR, 2004. http://wwwhome.cs.utwente.nl/~dijkman/YATL4MDR.html.

[8] Donald Norman. *Mental Models*, chapter Some Observations on Mental Models, pages 7–14. LEA, 1983.

[9] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] William Harrison and Harold Ossher. Subject-Oriented Programming (A critique of pure objects). In *OOPSLA'93*, pages 411–428, 1993.

[11] William Harrison, Harold Ossher, Stanley Sutton Jr., and Peri Tarr. Concern Modeling in the Concern Manipulation Environment. Technical report, IBM Research Division, 21 September 2004.

[12] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264. Springer-Verlag, 2003.

[13] G. Kiczales. Aspect-oriented programming - the fun has just begun. In *Workshop on New Visions for Software Design and Productivity: Research and Applications*, December 2001.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[15] 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, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[16] L. T. Nguyen, Liping Zhao, and B. Appelbe. A Set Approach to Role Modeling. In *37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-37'00)*, 2000.

[17] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *OOPSLA*, pages 52–67, 2002.

[18] Sun Microsystems. Java Metadata Interface Specification, June 2002. document ID JSR-40.

[19] Sun Microsystems. NetBeans Metadata Repository, 2002. http://mdr.netbeans.org/.

[20] Object Management Group. Model Driven Architecture (MDA), July 2001. document ID ormsc/01-07-01.

[21] Object Management Group. Meta-Object Facility Specification, April 2002. version 1.4. document ID formal/02-04-03.

[22] Object Management Group. Unified Modeling Language (UML), March 2003. version 1.5. document ID formal/03-03-01.

[23] Object Management Group. Object Constraint Language, 2004. version 2.0. adopted spec. document ID ptc/2003-10-14.

[24] David L. Parnas. On the criteria to be used in decomposing systems into modules (1972). pages 411–427, 2002.

[25] Joel Richardson and Peter Schwarz. Aspects: extending objects to support multiple, independent roles. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 298–307. ACM Press, 1991.

[26] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 117–133. ACM Press, 1998.

[27] Robert France, Geri Georg, and Indrakshi Ray. Supporting multi-dimensional separation of design concerns, 2003.

[28] E. A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92), Vancouver, British Columbia, Canada*, pages 187–198. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1992.

[29] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updatable views in object-oriented databases. In *DOOD*, pages 189–207, 1991.

[30] T. Reenskaug, P. Wold, and O.A. Lehne. Working with objects, The OOram Software Engineering Method. Manning Publications Co., 1995.

[31] T. Tamai, N. Ubayashi, and R. Ichiyama. An Adaptive Object Model with Dynamic Role Binding. In *ICSE'05*, May 2005.

[32] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.