Integrating Shadows in Model Driven Engineering for Agile Software Development

Marc Conrad, Marianne Huchard, Thomas Preuss

Integrating Shadows in Model Driven Engineering for Agile Software Development

Marc Conrad
University of Bedfordshire
Luton, United Kingdom
marc.conrad@beds.ac.uk

Marianne Huchard
Université Montpellier II
Montpellier, France
huchard@lirmm.fr

Thomas Preuss
FH Brandenburg
Brandenburg, Germany
th.preuss@googlemail.com

Abstract

Shadows are well known as a programming language feature in the application area of MUDs (a certain type of multi-player online game). We argue that agile methodologies can be enhanced by the use of shadows as this feature because of its inherent ability to dynamically change the behavior of classes and objects, provides mechanisms to ease common tasks such as prototyping, deprecating, dynamic classification and interclassing at run-time. Therefore shadows should be considered a notion beyond any specific programming languages in order to facilitate its use in model-driven software engineering. For this we introduce Shadows-UML, an UML extension, that would help to push forward the convergence between model-driven and agile methodologies.

1. Introduction

The efficient production of quality software artifacts has been an evergreen aim and ongoing topic for debate amongst both the software development and consumer communities alike for many years. Despite the enormous investment made by both communities in seeking to design strategies and tools to deliver quality software on time and within budget, any cursory examination of the academic literature, or indeed popular press will confirm the self-evident fact that quality software artifacts emerge only rather rarely, given real resource constraints and the complexities of organizational needs, implementation tools, methodologies, paradigms and of course their changing industrial contexts of use. Behind this seemingly intractable problem (how to embed quality attributes within software as the norm, rather than the exception) it can be reasonably claimed that two modern day strategies now play a major role in IT system design and implementation. Firstly, the UML-based, Model-Driven approach [23], [31], [30] that provides soft-ware production with models, supported by notations, and diagrammatic visualizations that ensure that the system is explicitly designed before it is built, i.e. before the code is generated either automatically using the designs as input(s) or otherwise manually using the designs as templates and points of reference. On the other hand, partly as a reaction to what some have claimed to be "overly structured" processes inherent to the UML driven approach, agile methods exemplified by groups such as the Agile Alliance [1] seek to move the focus from a semi-formal design and notation driven approach to softer and dynamical organizational issues such as the need to engender active customer collaboration, use of rapid development software tools, and the need to develop systems in a totally flexible reactive and timely manner.

At first glance, these two software engineering strategies may seem to be either partially or fully incompatible. However a closer examination reveals a more subtle picture whereby integration of the two approaches has been advocated for example by Rumpe [28], who proposes a pragmatic approach to link agile methods with the UML model-based software development. Essentially, Rumpe suggests that the UML can play a supporting role within an agile approach by assisting with requirements capture, refinement, in early design documentation, as well as playing a later and vital role in code generation and test case definition (see also [29]). Recent advances in the automatic generation of code, namely the concept of an executable UML [19] suggest that what many traditional programmers might consider to be classical program coding activities using Java, C#, C++, et al., will be largely replaced by producing UML models where the expected behavior of the objects is predetermined via the Object Constraint Language [17], rather than determined by a human interpretation of a set of functional requirements or other notations, and documentation. Ideogramic to cite but one prominent Company active in this field, has for instance developed a gesture based diagramming tool, Ideogramic UML (TM), which allows users to sketch UML diagrams, explicitly promoting the use of

their visualization tool within agile methods such as XP, Crystal and DSDM [32, 12].

Hence, there may indeed be good pragmatic reasons to suggest that the UML will continue to play a significant role within the agile software development community. However, while the UML is self-evidently founded on the concept of Object Orientation it is also intimately linked reliant upon the object oriented programming paradigm itself. Hence, the UML is de facto ultimately constrained by the object-oriented programming features provided in widely used contemporary programming languages as Java, C#, C++ et al. In this context, the question raised here is whether the languages themselves may be too restrictive in their instantiation of the object oriented paradigm. In particular, it can be observed that the basic language constructs in these common languages may be far too restrictive to adequately address various heterogeneous demands that arise from prototyping, method deprecation, and specifically, support for the dynamic change of inheritance relationships at run-time program execution. These demands can be best served in a unified way via the concept of shadows [6].

We will show in Section 3 that a language with shadows as a core feature is strongly supportive of Agile Software Development strategies. Whilst shadows are tailored specifically to support Agile Software Development, it is perhaps also self-evident in view of the previous discussion regarding the future integrative approach many are now adopting to UML and agile methods that it is also necessary to demonstrate how to introduce this feature into the UML notation. In that situation shadows will provide a construct that truly integrates Agile Development and the UML.

The remainder of the paper is organized as follows. In the next section we give an introduction into the concept of shadows. This is followed in Section 3 by a brief discussion how shadows serve to enhance agile development. The integration of shadows into the UML is then demonstrated in Section 4 which is followed by a conclusion.

2. Shadows as a Programming Language Feature

The term shadow has been coined by the interpreted language LPC [26] that has been created in 1988 by Lars Pensjö (and later further developed by other contributors) for his invention LPMUD, an interactive multi-user environment mainly used for text based multi-user adventure games, so called "MUDs".

The core concept behind the shadow functionality is to mask one or more methods in a target object (the "shadowed" object) [8]. Every invocation of a shadowed method is first received by the shadow. The shadow can then forward this call to the shadowed object or do something else.

This is illustrated in Figure 1. A shadow object as shown in Figure 1 is able to intercept the method `display()` so that any such message call to jack is first dealt with in the `CaterpillarShadow` object.
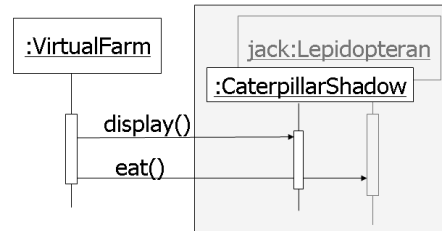

Figure 1. "Enhanced" Sequence Diagram illustrating the shadow functionality. The message `display()` is intercepted by the shadow `:CaterpillarShadow`. The message `eat()` is not shadowed and hence is received by jack the lepidopteran.

The evolution of LPC has been highly pragmatic driven by the demand of the active programmers in various MUDs rather then by a systematic, academically based, concept for designing a programming language. The shadow concept in LPC must be seen in this context: It has been proven to be useful "as is" but it has merely been evaluated academically. Indeed the programming "methodology" used in MUDs has aspects that can easily identified with professional agile development – in particular the progressive elaboration of software is an inherent feature in these MUDs as "playing the game" happens in parallel and cannot be separated from "further development of the underlying software".

Typical examples in computer games are special clothes such as an "invisibility cloak" that hides a player from others when worn. This invisibility cloak would then setup a shadow on the player object that, for instance, shadows the method that returns the description of the player. Called from another object (e.g. a different player) the shadow would then return something like "There is nothing to see there" instead of the real description. In general shadows are a useful feature wherever dynamic change of behavior is to be added to an otherwise pre-determined library.

Obviously such a concept needs clarification in a number of issues, for instance if attributes are shadowed, or which object is allowed to add shadows to other objects etc. On the programming language level these issues have been discussed in [6] and we will explicitly address them in Section 4.2 in the context of the UML.

A Java package that implements shadows can be found on the web site [5] that also contains a number of examples. It extends the idea of shadowing objects in LPC to the con-

cept of shadowing classes: A shadow of a class means that a shadow is added to every object that is instantiated from this class. In order to use the shadow functionality a special compiler is necessary that is also available on [5].

It should also be mentioned that the concept of "Posing" in Objective-C is somehow similar to shadows in LPC with the difference that "posed" methods change behavior only in subclasses of the "posed" class. A good introduction of the posing concept can be found in [16].

3. Shadows as a core feature to support agile development

For a number of agile strategies we can identify similar generic situations as described in the previous section where existing classes are to be enhanced by means of, say, customer feedback. For instance in Adaptive Software Development [15] each development cycle contains a "Speculation" phase (a form of adaptive planning) where it is essential to explore alternative solutions in so called joint application development sessions where developer and customers decide on desired product features. Shadows are also useful for prototyping during the Exploration phase of XP [2]. Equally, they allow for dynamic adaptation of features of classes of existing software libraries which is desirable in Feature Driven Development [25] .

In the following we want to focus on four areas where shadows can facilitate implementation tasks. While the first two ones (prototyping and deprecating) are common tasks in most agile methodologies, the latter two, namely interclassing and reclassification will be shown to be useful in agile context as well. Only the fact that these two latter features are neither available in most major programming languages nor are they commonly used in the UML itself may have prevented them to play any major role in current software engineering practice. Here, the integration of shadows in the modelling process could well initiate a change of thinking.

In the following we give only a broad overview - for details we refer the reader to [6] and [7]. It should however be noted that in contrast to the existing solitary solutions that currently exist and are referenced below shadows comprise a *unified* approach that serves a diversity of applications such as deprecating methods, prototyping, reclassification, interclassing and other concepts involving dynamic change of the behavior of classes and objects.

Deprecated methods. Not only in agile contexts software libraries are under constant evolution and it is a matter of fact that methods over time are subject to replacement by other methods for a variety of reasons. However, existing legacy code often still uses these deprecated methods. A shadow system could help the provider of the software to separate an object in an "official" version that is not messed up with any deprecated methods and a shadow for this object that contains deprecated methods. Hence the overhead of having the additional method is only locally where the deprecated method is needed.

Prototyping. Similar as shadows could be used for "fading out" deprecated methods, shadows could also be used for prototyping in software development. Especially in the case that a development process starts from an existing library and it is vital that the library is not to be changed (or that it is not possible to change the library for instance because of, say, that it is bought from an external supplier or because of copyright issues). A shadow then temporarily changes the behavior of a class or object in a well defined situation during development. We could even imagine an integration of automatically applied shadows into a concurrent version system (eg. CVS) that would help to implement branching in software development.

Reclassification and Dynamic Inheritance. Reclassification and a special case of it, dynamic inheritance, means to change the class of an object at run-time. It is not a coincidence that the reclassification example in [11] is located in the context of a computer game: A *Player* that is (an instance of) a *Frog* is reclassified to a *Prince* after being kissed. As already mentioned in Section 2 interactive, multi-user computer games where the development of the game is inherently weaved into "using" (i.e. playing) the game excellently mirror agile development methods.

It should be noted however that dynamic inheritance is not a new feature. A work-around is already discussed in [9] and role assignment via dynamic classification is advocated in [24]. Automatic reclassification based on the value of predicates is implemented as *predicate classes* [3] in Cecil [4]. In [18] a Java extension featuring Dynamic Inheritance is proposed while the most consequent approach for reclassification can be found in *Fickle* (e.g. [13, 14] and [11]).

Interclassing. For a motivation of interclassing we refer to [27, 10] (in a general context), or [6] (in a mathematical context). In general interclassing denotes the insertion of a new class in an existing inheritance hierarchy. This usually happens in a situation where the inheritance hierarchy to be modified is in the context of an existing library that cannot be changed (for instance because of a copyright or that it has to be left unchanged for existing applications etc). Interclassing is a useful feature in methodologies that advocate some kind of incremental or staged development as it allows to systematically "build up" software from stage to stage by adding new classes in any position of a given class hierarchy while at the same time keeping an intact library at any given stage. The implementation of interclassing with shadows has been discussed in [6].

4. Enhancing the UML with Shadows

As we have seen in Section 3 shadows are an invaluable means to enhance the *unanticipated evolution of written programs* in particular in reference to interclassing, dynamic instance reclassification, method deprecation and prototyping. In the context of model-driven development however they are as well valuable for solving *Design-To-Code* problems in the sense that shadows help to translate almost unconstrained models (unconstrained because they use the richness of expression of UML) into programs written in a family of current, widespread programming languages (Java, C#, C++ to name but a few) with strong constraints. Typically those constraints include single inheritance, static structure of the inheritance hierarchy, poor naming strategy for attributes and methods, no multi-instantiation, no dynamic reclassification. By extending these languages with the single new feature of shadows (as shown in [6] for Java) a much richer variety of models can be implemented.

However with the increasing tendency to develop software driven by modeling in even agile contexts it is clear that shadows must be somehow integrated into the UML itself. An UML-based notation for shadows (from now on called Shadows-UML) is useful both for unanticipated evolution (as described in Section 3) and design-to-code translation. For the latter these Shadows-UML diagrams will specify the use of shadows in translating an unconstrained UML class diagram (independently of a specific programming language). Only after this, code will be written. In this context patterns of use of shadows can be defined for easing translation.

4.1 Dynamic Classification in the Lepidopteran Example

In the left of Figure 2 we present the lepidopteran as a classical example of the so-called "dynamic classification" [20]: An instance of `Lepidopteran` is to change dynamically (at run time) its subclass. The lepidopteran starts its life as a caterpillar, then becomes a chrysalis and finally a butterfly. The proposed model here is a well defined and correct UML representation since at any given time the lepidopteran set is divided into chrysalis, caterpillars and butterflies. A consequence of this representation is that an instance of `Caterpillar` may evolve, when the system runs, to become an instance of `Chrysalis`. Current main-stream programming languages cannot manage such a situation except destroying the first instance and creating the second one, but then the identity of objects in the system is not ensured. In the classical case the UML (analysis level) model has to be transformed into another (design level) model to conform to the target programming

language. This is known in Model-Driven Engineering as a PIM to PSM (Platform Independent Model to Platform Specific Model) model transformation.

In Figure 2 (right and bottom) we see how specialization has been replaced by the association `hasTheForm`, a usual bypass used to manage dynamic specialization, except that here the association has the stereotype `shadowableBy`, meaning that an instance of the class `Chrysalis` acts as a shadow for `jack` the lepidopteran: `jack` now has new attributes (e.g. thickness of the cocoon) and methods (sleeping) coming from class `Chrysalis`. The deep semantics of instances is changed (as we will describe in detail later in the definition of Shadows-UML): The set of properties of a shadowed instance is enhanced by the set of properties of the shadow and messages sent to `jack` are filtered and (possibly) dealt with by the chrysalis shadow.

4.2 Shadows-UML

We have chosen to extend the UML in the standard way by providing a profile, composed of stereotypes, tagged values and constraints that specify the new semantics following the UML superstructure specification [21]. This has obvious advantages against the two other alternative ways namely defining a new meta-model using the MOF specification [22], which would require the redefinition of the whole language or the extension of the UML meta-model by means of specialization which is generally uncommon.

The new profile has already been introduced intuitively in the previous section in the example on dynamic classification. We go on to discuss some more details.

Two stereotypes are defined for classes: <<Shadowable>> for classes whose instances are able to admit shadows and <<Shadow>> for classes that describe shadows. Similarly, two stereotypes are also defined for instances because shadows are operational at the instance level. The stereotype <<ShadowedInstance>> is associated to an instance that is masked by one or more shadow instances (which themselves are of stereotype <<ShadowInstance>>). We also define the stereotype <<ShadowableBy>> for associations that relate shadowable objects and their shadows. Finally the stereotype <<nomask>> is reserved for operations that are not allowed to be shadowed. Experience shows that this feature is often useful to secure the integrity of core functionality of objects.

The constraints that ensure the well-formedness of models including shadows are as follows.

1. The classifier of a <<ShadowedInstance>> (resp. <<ShadowInstance>>) is a <<Shadowable>> class (resp. a <<Shadow>> class).
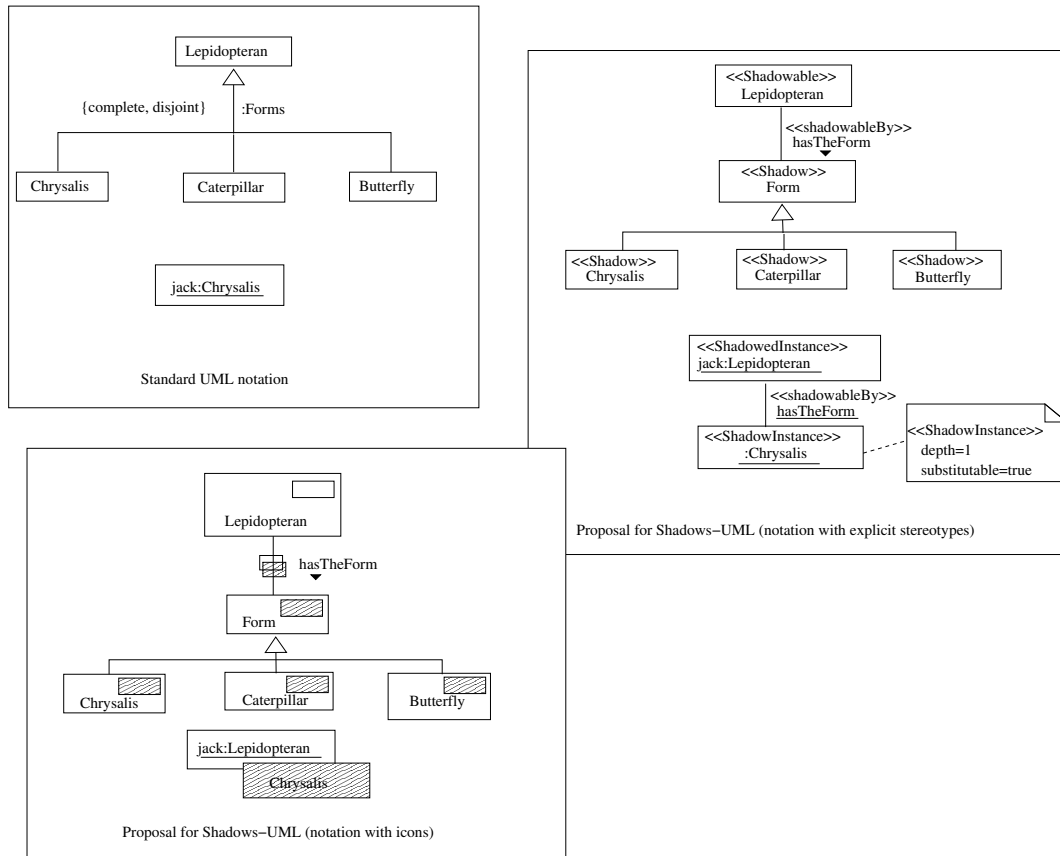
Figure 2. Lepidopterans in UML (left) and Shadows-UML (right)

2. If an instance admits several shadows, these shadows are totally ordered (tag `depth` indicates this order). Values of the tag `depth` for shadows associated with a shadowed instance form a totally ordered set (no repeated values).

3. The `shadowableBy` association is binary: it connects a <<Shadowable>> class with a <<Shadow>> class.

Finally we define the constraints about the new added semantics.

1. As specified in the UML meta-model, an instance has one slot per structural feature of its class, including inherited features. In addition, a shadowed instance owns the slots of its shadows.

2. To a shadowed instance messages can be sent that correspond to operations owned by its class or to operations owned by the classes of its shadows.

3. Features (both structural and behavioral) owned by a shadowed instance are ordered using the depth of its associated shadows: features of shadows come first in the order given by the depth of the shadow (lower numbers come first), then features coming from the instance class.

5. Conclusion

Keeping in mind the current trends in software engineering there are indeed pragmatic reasons to expect the UML to play a significant role in the agile software community. In this paper we identified shadows, as they have been introduced in LPC, to be an excellent means to support Agile Software Development strategies. Hence the necessity has been suggested to enhance the UML by integrating shadows. In doing so, we were able to illustrate the use of shadows in the UML on such ostensibly esoteric features such as for instance interclassing or dynamic classification.

Reversely however we may as well deduce that – as such features could be straightforwardly modeled with the help of shadows – interclassing, dynamic classification (and as well other features such as multiple inheritance and dynamic change of behavior) may well play a more central role in Agile Development in the future and hence will be-

come less "esoteric" in mainstream software development.

We have shown that shadows are a feature that really reflects the "spirit" of agile development methods. Hence we feel it is necessary that they are considered a notion beyond any specific programming languages. Not only that shadows enhance the design of software as they allow to translate almost unconstraint models into programs written in mainstream languages if these languages have been extended in a suitable way to support shadows (as in [6] for Java), they are also particularly valuable in case of an unanticipated evolution of existing software libraries. Hence we feel that an extension of the UML in the way described in this paper supports a desirable convergence process between model-driven and agile methodologies not least because it facilitates a PIM to PSM model transformation that allows to derive part of the code automatically.

References

[1] Agile Alliance. Manifesto for agile software development, 2006. http://www.agilemanifesto.org.

[2] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[3] C. Chambers. Predicate classes. *Lecture Notes in Computer Science*, 707:268–296, 1993.

[4] C. Chambers. *The Cecil language specification and rationale: Version 2.0.* University of Washington, December 1995.

[5] M. Conrad. Implementing a java shadow using a jikes extension, 2004. http://www.perisic.com/shadow/jshadow.

[6] M. Conrad, T. French, M. Huchard, C. Maple, and S. Pott. Enriching the object-oriented paradigm via shadows in the context of mathematics. *Journal of Object Technology*, 5(6):107–126, July-August 2006.

[7] M. Conrad, T. French, and C. Maple. Object shadowing - a key concept for a modern programming language. In *Proc of 2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity*, 2004.

[8] M. Conrad and M. Huchard. Enhancing the uml with shadows for agile development. In *Proceedings of SE2007, Hamburg (to appear)*, 2007.

[9] J. Coplien. *Advanced C++ programming styles and idioms*. Addison-Wesley, 1992.

[10] P. Crescenzo and P. Lahire. Using both specialisation and generalisation in a programming language: Why an how, 2002.

[11] F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. On re-classification and multithreading. *JOT-04*, 3(11):5–30, 2004.

[12] C. H. Damm, K. M. Hansen, and M. Thomsen. Tool support for cooperative object-oriented design: gesture based modelling on an electronic whiteboard. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 518–525, New York, NY, USA, 2000. ACM Press.

[13] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Electronic proceedings of FOOL8 (http://www.cs.williams.edu/ kim/FOOL/)*, 2001.

[14] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *ECOOP'01*, LNCS 2072, pages 130–149. Springer, 2001.

[15] J. A. Highsmith. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing, 2000.

[16] A. Isbell. Objective-c posing and categories in rhapsody development – a real-world example. Technical report, Stepwise, 1998.

[17] A. Kleppe, J. Warmer, and S. Cook. Informal formality? the object constraint language and its application in the uml metamodel. In *Selected papers from the First International Workshop on The Unified Modeling Language <<UML>>'98*, pages 148–161, London, UK, 1999. Springer-Verlag.

[18] G. Kniesel. Darwin & and lava – object-based dynamic inheritance ... in java, 2002. Poster presentation at ECOOP 2002.

[19] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for ModelDriven Architectures*. Addison-Wesley, 2002.

[20] P.-A. Muller and N. Gaertner. *Modlisation objet avec UML*. Eyrolles, 2000.

[21] Object Management Group. *UML 2.0 Superstructure Specification. ptc/04-10-02*, October 2004.

[22] Object Management Group. Meta object facility (mof) 2.0 core specification, 2006. http://www.omg.org/technology/documents/formal/MOF_Core.htm.

[23] Object Management Group. Uml® resource page, 2006. http://www.uml.org.

[24] J. Odell, H. V. D. Parunak, S. Brueckner, and J. A. Sauter. Changing roles: Dynamic role assignment. *Journal of Object Technology*, 2(5):77–86, 2003.

[25] S. R. Palmer and J. M. Felsing. *A Practical Guide to Feature-Driven Development*. Prentice-Hall, 2002.

[26] L. Pensjö et al. Lpc, 1998. http://www.lysator.liu.se/mud/lpc.html.

[27] P. Rapicault and A. Napoli. Evolution d'une hirarchie de classes par interclassement. *Confrence Langages et Modles Objets - LMO 2001, Le Croisic $France$ - publi dans la revue l'Objet (Hermes Eds)*, 7(1-2):215–232, 24-26 janvier 2001.

[28] B. Rumpe. Agile modeling with the uml. In *RISSEF*, pages 297–309, 2002.

[29] B. Rumpe. Agile test-based modeling. In *Proceedings of the 2006 International Conference on Software Engineering Research & Practice. SERP'2006.s*, USA, 2006. CSREA Press.

[30] J. Siegel and OMG Staff Strategy Group. *Model-Driven Architecture. OMG document omg/2000-11-05*. Object Management Group, 2000.

[31] J. Siegel and OMG Staff Strategy Group. *Developing in OMG's Model-Driven Architecture. OMG document omg/2000-11-05*. Object Management Group, 2001.

[32] I. Tyrsted and P. Gerken. Ideogramic, 2002. http://www.ideogramic.com.