The Architectural Style of Adaptive Object-Models

Presented for ECOOP 2001 Workshop on Adaptive Object-Models and Metamodeling Techniques

Joseph Yoder, Federico Balaguer, Ralph Johnson

Software Architecture Group – Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
yoder@refactory.com, [balaguer|johnson]@cs.uiuc.edu

Abstract

Many object-oriented information systems share an architectural style that emphasizes flexibility and dynamically configurable. Business rules are stored in a database instead of in code. The object model that the user cares about is part of the database, and the object model of the code is just an interpreter of the users' object model. We call these systems "Adaptive Object-Models", because the users' object model is interpreted at runtime and can be changed with immediate (but controlled) effects on the system interpreting it.

I. Introduction

The era where business rules are buried in code is coming to an end. Users often want to change their business rules without writing new code. Customers require systems to adapt more easily to changing business needs, that they meet their unique requirements, and to scale to large and small installations [Rouvell2000].

Many information systems today need to be dynamic and configurable so that they can quickly change to adapt to new business needs. This is usually done by moving certain aspects of the system, such as business rules, into a database so they can be easily changed. The resulting model allows for a system to quickly adapt to changing business needs by simply changing values in the database rather than code. It also encourages the development of tools that allow decision-makers and administrators to introduce new products without programming and to make changes to their business models at runtime. This can reduce time-to-market of new ideas from months, to weeks and days. Therefore, the power to customize the system is placed in the hands of those who have the business knowledge to do it effectively.

Architectures that can dynamically adapt at runtime to new user requirement are sometimes called a "reflective architecture" or a "meta-architecture". This paper focuses on a particular kind of reflective architecture that has been given many names. It was called the "Type Instance pattern" in a tutorial at OOPSLA'95 [GHV95]. This paper calls it the "Adaptive Object-Model (AOM) architecture". Most of the systems we have seen with an Adaptive Object-Model are business systems that manage products of some sort and are extended to add new products, and we have called it "User Defined Product architecture" in the past [JO98]. These systems have also been called "Dynamic Object Models" and "Active Object-Models". Martin Fowler's "knowledge-level" is usually just another name for an Adaptive Object-Model; any system with observation types and accountability types is probably an Adaptive Object-Model [Fowler97].

An Adaptive Object-Model is a system that represents classes, attributes, and relationships as *metadata*. It is a model based on instances rather than classes. Users change the *metadata* (object model) to reflect changes in the domain. These changes modify the system's behavior. In other word, it stores its *Object-Model* in a database and interprets it. Consequently, the object model is active, when you change it, the system changes immediately.

This kind of architecture has been used to represent insurance policies [JO98], to bill for telephone calls, and to check whether an equipment configuration is likely to work. It has been used to model workflow [Devos98] [Manoles2000], to model documents, and to model databases. It has probably been used for a lot more things; these are just the ones we have seen and are allowed to talk about.

We have noticed that the architects of a system with Adaptive Object-Models often claim this is the best system they have ever created, and they brag about its flexibility, power, and eloquence. At the same time, many of the developers find them confusing and hard to work with. This is due partly because an Adaptive Object-Model has several levels of abstraction, so there are several places that could be changed. Perhaps some developers are just not capable of working on these kinds of projects. However, we feel that the problem is just as likely caused by poor documentation and lack of training of developers. The Adaptive Object-Model style has never been described, and most of the architects that use it don't realize how widely it is used.

This paper describes the Adaptive Object-Models architectural style in detail. It also describes results of implementing an Adaptive Object-Model using the *Observation* pattern at the Illinois Department of Public Health. Our goal is to help people understand Adaptive Object-Models so that they only use them when they are appropriate and so that when they use them, they use them correctly.

2. Adaptive Object-Models

Adaptive Object-Models provide an alternative to traditional object-oriented design. Traditional object-oriented design generates classes for the different types of business entities and associates attributes and methods with them. The classes model the business, so a change in the business causes a change to the

code and leads to a new version of the application. An Adaptive Object-Model does not model these business entities as classes. Rather, they are modeled by descriptions that are interpreted at run-time. Thus, whenever a business change is needed, these descriptions are changed which are then immediately reflected in the running application.

Adaptive Object-Model architectures are usually made up of several smaller patterns. The most important is *TypeObject* [Johnson98], which separates an Entity from an EntityType. Entities have Attributes, which are implemented with the *Property* pattern, and the *TypeObject* pattern is used a second time to separate Attributes from AttributeTypes. The *Strategy* pattern is often used to define the behavior of an EntityType. As is common in Entity-Relationship modeling, an Adaptive Object-Model usually separates attributes from relationships. Finally, there is usually an interface for non-programmers to define new EntityTypes.

- **TypeObject**

  Most object-oriented languages structure a program as a set of classes. A class defines the structure and behavior of objects. Most object-oriented systems use a separate class for each kind of object, so introducing a new kind of object requires making a new class, which requires programming. However, if the difference between different types of classes is small enough, the objects can be generalized and the difference between them described by parameters. The *TypeObject* pattern splits a class into two classes, one the type of the first, and then to replace subclasses of the original with instances of the type class (see Figure 1).

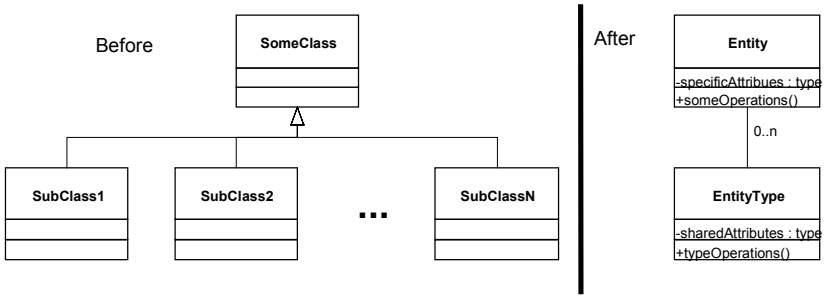

Figure 1 - TypeObject

TypeObjects can be used in the factory scheduling system to replace subclasses of Product and Machine with instances of ProductType and MachineType. It can be used in an airline scheduling system to replace subclasses of Airplane with instances of AirplaneType [Coad1992]. It can be used in a telecommunications billing system to replace subclasses of NetworkEvent with instances of NetworkEventType. We also used it in the Observation model to represent the relationship between Observation and ObservationType (see **Erreur ! Source du renvoi introuvable.**). In all these cases, the difference between one type of object and another is primarily their data values, not their behavior, so the *TypeObject* pattern works well.

- **Property**

  The attributes of an object are usually implemented by its instance variables. A class defines the instance variables of its instances. If objects of different types are all the same class, how can their attributes vary? The solution is to implement attributes differently. Instead of each attribute being a different instance variable, make an instance variable that holds a collection of attributes (Figure 2).

  The core of an Adaptive Object-Model is a combination of *TypeObject* and *Property* [Foote98]. The *TypeObject* pattern divides the system into Entities and EntityTypes. Entities have properties. But usually each property has a type, too, and each EntityType then specifies the types of the properties of its entities. A PropertyType is usually more like a variable declaration than like an

abstract data type. It often keeps track of the name of the property, and also whether the value of the property is a number, a date, a string, etc. The result is an object model similar to the following: Sometimes objects differ only in having different properties. For example, a system that just reads and writes a database can use a Record with a set of *Properties* to represent a single record, and can use RecordType and PropertyType to represent a table.

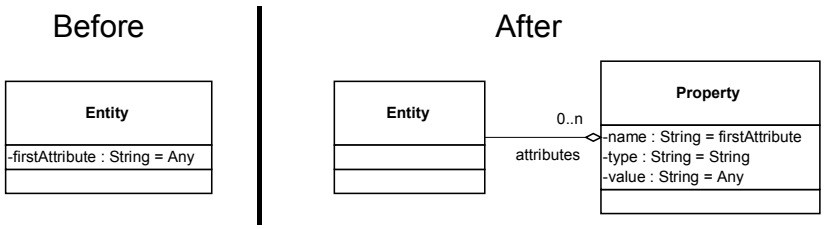

Figure 2 - Properties

But usually different kinds of objects have different kinds of behaviors. For example, maybe records need to be checked for consistency before being written to a database. Although many tables will have a simple consistency check, such as ensuring that numbers are within a certain range, a few will have a complex consistency checking algorithm. Thus, *Property* isn't enough to eliminate the need for subclasses. An Adaptive Object-Model needs a way to change the behavior of objects.

- **Strategy**

A strategy is an object that represents an algorithm. The strategy pattern defines a standard interface for a family of algorithms so that clients can work with any of them. If an object's behavior is defined by one or more strategies then that behavior is easy to change.

Each application of the *Strategy* pattern leads to a different interface, and thus to a different class hierarchy of *Strategies*. In a database system, strategies might be associated with each property and used to validate them. The *Strategies* would then have one public operation, validate. But *Strategies* are more often associated with the fundamental entities being modeled, where they implement the operations on the methods.

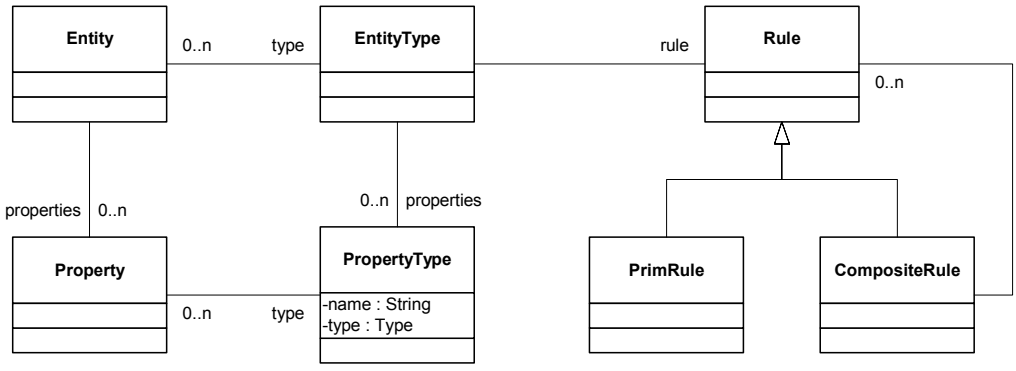

Figure 3 - Type Square

These Strategies can evolve to become more complex business rules that are built up or interpreted at runtime. These can be either primitive rules or combination of business rules through application of the *Composite* pattern. If the business rules are workflow in nature, you can use the

Micro-Workflow architecture as described by Manolescu. Micro-Workflow describes classes that represent workflow structure as a combination of rules such as repetition, conditional, sequential, forking, and primitive rules. These rules can be built up at runtime to represent a particular workflow process.

Figure 3 is a UML diagram of applying the *TypeObject* pattern twice with the *Property* pattern and then adding *Strategies* for representing the behavior. This resulting architecture is called the *TypeSquare* pattern and is often seen in adaptable systems with knowledge levels as described in this paper.

In the Observation model, Validators represent domain-related strategies for deciding whether the value of a given Observation is valid or not. The business rules for the Validators can be represented by parameterize the differences of the valid results with two different types of Validators; RangedValidator and DiscreteValidator. Analysis revealed that future models could have more complicated business rules that allowed for composite business rules that could use and/or logic.

- **Entity-Relationship**

Attributes are properties that refer to immutable values like numbers, strings, or colors. Relationships are properties that refer to other entities. Relationships are usually two-way; if Gene is the father of Carol then Carol is the daughter of Gene. This distinction, which has long been a part of classic entity-relationship modeling and which has been carried over into modern object-oriented modeling notations, is usually a part of an Adaptive Object-Model architecture. The distinction often leads to two subclasses of properties, one for attributes and one for relationships.

One way to separate attributes from associations is to use the *Property* pattern twice, once for attributes and once for associations. Another way is to make two subclasses of Property, Attribute and Association. An Association (called *Accountability* by Fowler) would know its cardinality. A third way to separate attributes from associations is by the value of the Property. Suppose there is a class Value whose subclasses are all immutable. Typical values would be numbers, strings, quantities (numbers with units), and colors. *Properties* whose value is an Entity are associations, while properties whose value is a Value are attributes. It is interesting that few language designers seem to feel the need to represent these relationships, but most designers of systems with Adaptive Object-Models do.

- **User Interface for Defining Types**

One of the main reasons to design an Adaptive Object-Model is to extend the system by defining new types without programming. Sometimes the goal is to enable users to extend the system without programmers. But even when only the developers will define new types, it is common to build a specialized user interface for defining types. For example, the insurance framework at the Hartford has a user interface for defining new kinds of insurance, including the rules for calculating their price. Innoverse, a telephone billing system, has a user interface for defining geographical regions, monetary units, and billing rules for different geographical regions expressed in various monetary units. The Argos school administration system lets has a user interface for defining new document types and workflows.

Types are often stored in a centralized database. This means that when someone defines new types, applications can use them without having to be recompiled. Often applications are able to use the new types immediately, while other times they cache type information and must refresh their caches before they will be able to use the new types.

The alternative to having a user interface for creating and editing type information is write programs to do it. In fact, if programmers are the only ones creating type information then it is often easier to let them do it by writing programs, since they can use their usual programming environment for this purpose. But the only way to get non-programmers to maintain the type information is give it a user interface.

Adaptive Object-Models are usually built from applying one or more of the above patterns in conjunction with other design patterns such as *Composite*, *Interpreter*, and *Builder*. *Composite* [GOF95] is used for either building dynamic tree structure types or rules. For example, if your entities can be composed in a dynamic tree like structure, the *Composite* pattern is applied. *Builders* and *Interpreters* are commonly used for building the structures from the meta-model or interpreting the results.

But, these are just patterns; they are not a framework for building Adaptive Object-Models. Every Adaptive Object-Model is a framework of a sort but there is no generic framework for building them. Rather, these set of patterns are commonly applied to the Adaptive Object-Model architectural style. The first Adaptive Object-Model you build is the hardest, then you know the patterns and can understand them.

3. Advantages and Disadvantages of Adaptive Object-Models

Every architecture style has both advantages and disadvantages. The main advantage of the Adaptive Object-Model is ease of change. An Adaptive Object-Model is good if your system is constantly changing, or if you want to allow your users the ability to dynamically configure and extend their system. An Adaptive Object-Model can lead to a system that allows users to "program without programming". Alternatively, an Adaptive Object Model can evolve into a domain-specific language.

Turning a program into an Adaptive Object-Model usually makes it much smaller. Information that was encoded in the program is now encoded in the database. This new class structure doesn't change. Instead, changes to the specification lead to changes in the content of the database.

The primary disadvantage is that these systems can be hard to build. They can also be harder to understand. To understand them completely, you must understand two object systems; the interpreter written in the object-oriented programming language and the Adaptive Object-Model that is interpreted. Classes do not represent business abstractions because all information about the business is in the database.

Adaptive Object-Models generally need tools and support GUIs for defining the objects in your system. Adaptive Object-Models can also be slower since they are usually based upon interpreting the representation of your object model. However, our experience is that lack of understanding is a bigger problem than lack of speed.

As stated before, you are in a sense building a domain-specific language. And although it is often easier for users to understand than a general-purpose language, you are still developing a language. This means that you inherit all of the problems associated with developing any language such as needing debuggers, version control, and documentation tools. Other problems are those involved with training. There are ways around these problems but they are problems non-the less and should be faced.

People are building systems that use metadata and Adaptive Object-Models from the ground up for a variety of reasons. OOPSLA Workshop participants identified the issues relating to when you want to build these types of systems and reasons that cause Adaptive Object-Models to fail [OOPSLA '98].

The main business case for an Adaptive Object-Model is to make it possible to develop and to change software quickly. If software takes too long to change or develop, either:

- time-to-market is possibly too long: the window of opportunity is missed; products, in particular short-lived ones do not pay back the investment made into their development, etc.
- the software is inadequate: what may have been defined on paper may not be what the user gets, in particular, if the product is complex.
- innovation is hindered: if it takes a few weeks or even month until a product can be handled using a computer, no playful exploration and what-if scenarios are possible.

Adaptive Object-Models seek to overcome these problems by drastically reducing time-to-market, by giving immediate feedback on what a new application looks like and how it works, and by allowing users of the system to experiment with new product types.

It should be noted however, that there could be a higher initial cost associated with developing an Adaptive Object-Model. This is primarily because it is harder to develop a general solution to a problem. It usually requires iterations. Customer feedback is also very important. It is important to know who your customers really are. Are they going to be programmers taking the changes and using your framework to add in the new business rules, or are they going to be users of the application? Iterating by regularly releasing software and can help ease the problem of paying for the framework and can help give the needed customer feedback as the framework evolves.

Obviously, an Adaptive Object-Model requires a system to interpret the model. The Adaptive Object-Model is embedded in a system, and effects its execution. Thus, Adaptive Object-Models require adequate software support. The software must provide the following functionality:

- a metamodel that provides the model elements from which object models can be built;
- tools that let users define object models;

- a model engine that interprets or just-in-time-compiles-and-runs object models;
- a well-defined connection between an object model definition and its execution by a model engine.

The main consequence of basing a system on an Adaptive Object-Model is that changes to an object model can be effective immediately. The model engine may immediately execute the new model, which means, for example, that a new product type is in place with a minimal delay.

The positive consequences of this minimal turn-around time are:

- time-to-market is as short as possible;
- costs of developing software support for new products are drastically reduced;
- misunderstanding between customers and IT is reduced, as immediate feedback is provided;
- innovation is fostered: given a safe setting, changes to a model can be explored.

The negative consequence of this approach is:

- generic tools may be inadequate to deal with an Adaptive Object-Model.
- Product-specific tools typically do a better job in supporting the handling of a new product than generic tools do. But then, if product-specific tools are needed (which may not be the case for short-lived products), they can always be implemented later, after one has learned from using the generic tools what the requirements for these product-specific tools.

Adaptive Object-Models are not the solution to every problem. They require sustained commitment and investment, and their flexibility and power can make them harder to understand and maintain. They should only be undertaken when either the products or rules associated with products are rapidly changing, or, when users demand to have highly configurable systems.

4. A Process for Developing Adaptive Object-Models

Applications that use the Adaptive Object-Model architecture are almost always developed iteratively. It is important to not loose focus and to continue to receive feedback from real users. Adaptive Object-Models tend to evolve from frameworks. [Roberts1998] gives a simple overview of the process by which frameworks evolve. If your system is only going to change a couple of times over its lifetime, then the effort entailed in constructing a framework may not be worth the cost and bother. However, when business rules change frequently, there is a decided advantage to be gained from letting changes to the system be released rapidly, and an Adaptive Object-Model may be right for you. It is often said that in today's business, things change so fast that it is almost impossible to keep up. An Adaptive Object-Model can help position a business to adapt to a fast changing business climate.

When you see that you need additional flexibility, it is often tempting to try and figure out exactly where an application will need its flexibility. The problem with this approach is that too much time can be spent on an area that may possibly give diminishing returns for the amount of time spent on the development. Or, worse yet, the flexibility that was added is not what the customer really needs.

Therefore it is important to build these frameworks by releasing applications to customers and evolving the frameworks as you figure out where the flexibility is needed and the similarities between applications. This evolutionary process provides some important benefits:

- the customers get released software to meet their current business needs;
- the framework is funded through the meeting of these different customer needs and releasing working applications for them;
- the true needs of the applications are realized.

An important part of this process is to occasionally go back and retrofit the framework as it evolves to earlier releases. This allows the customers to get the additional benefits of the framework and it also potentially minimizes maintenance costs.

Since Adaptive Object-Models are generally more complicated designs, they usually require the better than average developer. However, you can evolve your framework with a smaller number of developers. Developers generally find this type of work more interesting so there is a good incentive to help keep developers interested and working on these frameworks.

As seen through OOPSLA and ECOOP workshops [OOPSLA98] [OOPSLA99] [ECOOP2000], many companies have been successful at building Adaptive Object-Models and very good business cases can be made for building these frameworks. However it should be undertaken with careful analysis. It is an alternative to traditional object-oriented design that, like any architecture, has both advantages and disadvantages.

5. Alternatives to Adaptive Object-Models and Related Work

There have been many techniques applied over the years for moving business rules out of the code, making systems more adaptable to new requirements. The best-known alternatives or related techniques for building these types of systems are Generative Programming, Metamodeling, Table-Driven systems, and Business Rules research.

Generative Programming [Czarn2000] provides infrastructure for transforming descriptions of a system into code. Descriptions are based on provided primitive structures or elements [Roy98]. Code generators produce either executable-code or source-code. Generative Programming approach focuses on the automatic generation of systems from high-level descriptions. In this context it is arguable whether the high-level description acts like the meta-model of the generated system. It is related to Adaptive Object-Model in that the functionality of systems is not directly produced by programmers but specified using domain-related constructs. There are also editors commonly built for describing the metadata for generating code. This technique is different from Adaptive Object-Model primarily because it decouples the meta-model from the system itself. Adaptive Object-Models immediately reflect the changed business requirement without any code generation or recompilation.

Metamodeling techniques [MetaMol2000] include a variety of approaches most of which are generative in nature. In general, these techniques focus on manipulating the model and meta-model behind a system as well as supporting valid transformations between representations [Revault2000]. Quite often the attention is more on the meta-model, or a model for generating a model, rather than the final application that will reflect the business requirements.

They are related to Adaptive Object-Models in that they both have a “meta” model that is used for describing (or reflect) the business domain, there are usually special GUI tools for manipulating the model, and metadata is usually interpreted for the creation of the actual model. The primary difference is that Metamodeling techniques as provided by CASE tools generate the code from the descriptive information [Rouvell2000] while Adaptive Object-Models interpret the descriptive information at run-time. Thus, if you change your business information with a CASE tool, you will generate a new program, compile and release it to your users. While in an Adaptive Object-Model, you change your business information, which is usually stored in a shared database that the running systems have access to. Then, once the information becomes available, the system immediately reflects the new changes without having to release a new system.

Table-Driven systems have been around since the early database days in the 1970’s. Quite often the differences in the business rules can be parameterized [Perkins2000]. By storing these differences in a database, the running system can either interpret these changes from a database table or the appropriate function can be called with the differing values from the database. Sometimes these are built with triggers and stored procedures.

A lot of recent work has been done towards looking at ways to represent business rules, specifically allowing for the rules to dynamically change. OOPSLA 2000 sponsored a workshop [OOPSL2000], which focused on just this topic where many papers were presented describing both working systems and research in this area.

6. Summary

Adaptive Object-Models provide an alternative to traditional object-oriented design. Traditional object-oriented design generates classes for the different types of business entity and associate attributes and methods with them. These are such that whenever a business change to the system is needed, a developer has to change the code and release a new version of the application for the change to take affect. An Adaptive Object-Model does not model these business entities as first class objects. Rather, they are modeled by a description of structures, constraints and rules within the domain. The description is interpreted and translated into the meta-model that drives the way the system behaves. Thus, whenever a business change is needed, these descriptions can change and be immediately reflected in the running application. The most important design patterns needed for implementing these types of dynamic systems are *Type-Object*, *Properties*, *Composite*, and *Strategy*.

Architects that develop these types of systems are usually very proud of them and claim that they are some of the best systems they have ever developed. However, developers that have to use, extend or maintain them, usually complain that they are hard to understand and are not convinced that they are as great as the architect claims. This is usually because of lack of understanding of these types of systems.

This architectural style can be very useful in systems; specifically systems that emphasizes flexibility and those that need to be dynamically configurable. However, this style has not been well documented and is hard to understand; primarily due to the many levels of abstraction.

This paper describes the architectural style of Adaptive Object-Models, including the process for developing them along with advantages and disadvantages. We hope that this paper will help both architects and developers to understand, develop, and maintain systems based on an Adaptive Object-Model.

7. References

[Czarn2000] Krzysztof Czarnecki & Ulrich W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*, 2000. Addison-Wesley, 2000.

[Devos98] Martine Devos and Michel Tilman, *Repository-based Framework for Evolutionary Developmen*t, 1998. http://www.argo.be/OoFrame/

[ECOOP2000] Joseph W. Yoder & Reza Razavi. "Metadata and Adaptive Object-Models", ECOOP'2000 Workshop Reader; Lecture Notes in Computer Science, vol. no. 1964; Springer Verlag 2000.

[Foote98] B. Foote, J. Yoder. "Metadata and Active Object Models". Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998. URL: http://jerry.cs.uiuc.edu/~plop/plop98.

[Fowler97] M. Fowler. *Analysis Patterns, Reusable Object Models*. Addison-Wesley. 1997.

[GOF95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1995.

[GHV95] Erich Gamma, Richard Helm, and John Vlissides, *Design Patterns Applie*d, tutorial notes from OOPSLA'95.

[Johnson98] R. Johnson, B. Wolf. "Type Object". *Pattern Languages of Program Design 3*. Addisson Wesley, 1998.

[JO98] Ralph E. Johnson and Jeff Oakes, The User-Defined Product Framework, 1998. URL: http://st.cs.uiuc.edu/pub/papers/frameworks/udp.

[Manoles2000] D. Manolescu. "Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development". PhD thesis, Computer Science Technical Report UIUCDCS-R-2000-2186. University of Illinois at Urbana-Champaign, October 2000, Urbana, Illinois.

[MetaMol2000] MetaModeling and Model Engineering. URL: http://www.metamodel.com.

[OOPSLA98] Joseph W. Yoder, Brian Foote, Dirk Riehle, and Michel Tilman. Metadata and Active Object-Models Workshop Results Submission; OOPSLA Addendum, 1998.

[OOPSLA99] Joseph W. Yoder, Brian Foote, Dirk Riehle, Martin Fowler and Michel Tilman. Metadata and Active Object-Models Workshop; OOPSLA, 1999. URL: http://www.adaptiveobjectmodel.com/OOPSLA99.

[OOPSL2000] Ali Arsanjani and Joseph W. Yoder. Best–practices in Business Rule Design and Implementation; OOPSLA, 2000. URL: http://www.mum.edu/cs_dept/aarsanjani/oopsla2000/business-rules.html.

[Perkins2000] Business rules=meta-data. Proceedings of 34th International Conference on Technology of Object-Oriented Languages and Systems, 2000. On page(s): 285–294.

[Revault2000] N. Revault, X. Blanc & J-F. Perrot. "On Meta-Modeling Formalisms and Rule-Based Model Transforms", Comm. at Ecoop'2K workshop Iwme'00, Sophia Antipolis & Cannes, France, June, 2000.

[Roberts1998] D. Roberts, R. Johnson. "Patterns for Evolving Frameworks". *Pattern Languages of Program Design 3*. Addisson Wesley, 1998.

[Rouvell2000] Rouvellou, I.; Degenaro, L.; Rasmus, K.; Ehnebuske, D.; McKee, B. Extending business objects with business rules. Proceedings on Technology of Object-Oriented Languages, 2000. On page(s): 238 – 249,

[Roy98] Roy, G.G.; Kelso, J.; Standing, C. Towards a visual programming environment for software development. Proceedings on Software Engineering: Education & Practice, 1998. Page(s): 381 – 388.