Adaptive Object-Models for Implementing Business Rules

Joseph Yoder, Federico Balaguer, Ralph Johnson

Position Paper for Third Workshop on Best-practices
for Business Rules Design and Implementation

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 run-time configurability. 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. This paper is a submission for the Third Workshop on Best-practices for Business Rules Design and Implementation and describes the Adaptive Object-Model architecture as an example of use for implementing dynamic Business Rules.

I. Introduction

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]. 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 "Active Object-Models" [Foote98] and "Dynamic Object Models" [Riehle2000]. Martin Fowler's "knowledge-level" is usually just another name for an Adaptive Object-Model.

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.

2. Adaptive Object-Models

Adaptive Object-Models provide an alternative to traditional object-oriented design. Traditional object-oriented design use classes for describing 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. *TypeObject* [Johnson98] is used to separate an Entity from an EntityType. *TypeObject* provides a way to dynamically define new business entities for your system. Entities have Attributes, which are implemented with the *Property* pattern, and the *TypeObject* pattern is used a second time to separate Attributes from AttributeTypes. As is common in Entity-Relationship modeling, an Adaptive Object-Model usually separates attributes from relationships.

The *Strategy* pattern is often used to define the behavior of an EntityType. These strategies can evolve to a more powerful rule-based language that gets interpreted at runtime for representing changing behavior. Finally, there needs to be support for reading and interpreting the data representing the business rules that are stored in the database and there is usually an interface for non-programmers to define the new types of objects, attributes and behaviors needed for the specified domain.

- **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).

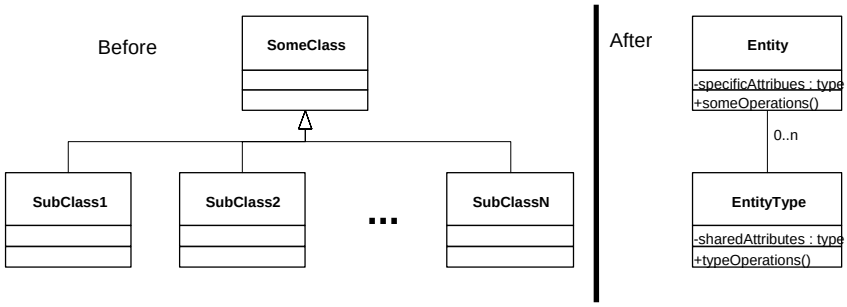

Figure 1 - TypeObject

TypeObjects can be used in a 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. 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 their 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.

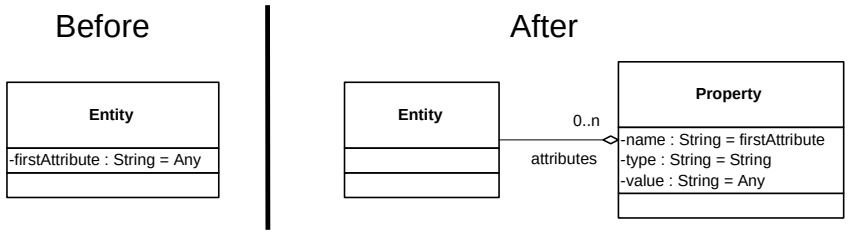

Figure 2 - Properties

  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 represent the relationships and the behavior of objects.

- **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 and Hayes [Fowler97, Hay96]) 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

programming language designers seem to feel the need to represent these relationships, but most designers of systems with Adaptive Object-Models do.

- **Strategies and Rule Objects**

  Business rule for object-oriented systems can be represented in many ways. Some rules will define the types of entities in a system along with their attributes. Other rules may define legal subtypes, which is usually done through subclassing. Other rules will define the legal types of relationships between entities. These rules can also define basic constraints such as the cardinality of relationships and if a certain attribute is required or not. Most of these types of rules deal with the basic structure and have been previously discussed on how AOMs deal with adapting these as runtime.

  However, some rules can not be defined this way. They are more functional or procedural in nature. For example, you may have a rule that describes the legal types of values that an attribute can have. Or, there may be a rule that states that certain entity-relationships are only legal if the entities have certain values and other constraints are met. These business rules become more complex in nature and AOMs use *Strategies* and *RuleObjects* [Arsanj99, 2000] to handle them.

  AOMs often start with some simple *Strategies* that are the basic functions needed for the new EntityTypes. These *Strategies* can be mapped to the EntityType through descriptive information that is interpreted at runtime. 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.

  However, as more powerful business rules are needed, these *Strategies* can evolve to become more complex such that they are built up or interpreted at runtime. These can be either primitive rules or the 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 [Manoles2000]. 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/RuleObjects* for representing the behavior. This resulting architecture is called the *TypeSquare* pattern [Yoder2002] and is often seen in adaptable systems with knowledge levels as described in this chapter.

- **Interpreters of the Metadata**

  Metadata for describing the business rules and objects model is interpreted in two places. The first is where the objects are constructed otherwise known as instantiating the object-model. The second is during the runtime interpretation of the business rules.

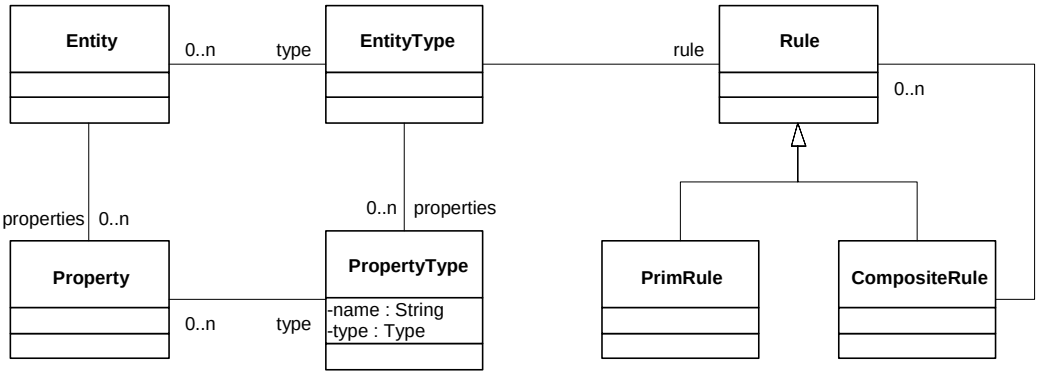

Figure 3 - Type Square

The information for describing the types of entities, properties, relationships, and behaviors are stored in a database for runtime manipulation, thus allowing for the business model to be updated and immediately reflected in applications interpreting the data.

Regardless of how the data is stored, it is necessary for the data to be interpreted to build up the adaptive object-model that represents the real business model. If an object-oriented database is used, the types of objects and relationships can be built up by simply instantiating the *TypeObjects*, *Properties*, and *Strategies.* Otherwise, the metadata is read from the database for building these objects, which are built using the *Interpreter* and *Builder* pattern.

The second place where the *Interpreter* pattern is applied is for the actual behaviors associated with the business entities described in the system. Eventually after new types of objects are created with their respective attributes, some meaningful operations will be applied to these objects. If these are simple *Strategies*, some metadata might describe the method that needs to be invoked along with the appropriate *Strategy*. These *Strategies* can be plugged into the appropriate object during the instantiation of the types.

However, if more dynamic rules are needed, a domain specific language can be designed using rules-objects [Arsanj98, 2000, 2001]. This approach has been called Grammar-oriented Object design (GOOD) [Arsan98, 2001]. For example, primitive rules can be defined and composed together with logical objects that form a tree structure that is interpreted at runtime. This is exactly how the business rules are described in Manolescu's Micro-Workflow architecture.

When dealing with functions, you need to have ways for dealing with constants and variables, along with constraints between values. *SmartVariables* [Foote98] can be useful for triggering automatic updates or validations when setting property values.

Table lookup is often used for dealing with constants or keeping track of variables. Sometimes, no matter how hard you try, the needs of the system become so complicated that the only solution is to create a rule language [Arsanj2000] using grammars, abstract syntax trees, constraint languages, and complex interpreters. The important thing to remember is to only evolve the language as the need dictates. The temptation can overtake a developer to create a new language that actually will make the maintenance and evolution of the application more difficult than if these rules were simply modeled in the base programming language.

- **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 programming environment for this purpose. However, the only way to get non-programmers to maintain the type information is give them user interface that they can use.

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 need to 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 currently no generic framework for building them. A generic framework for building the *TypeObjects*, *Properties*, and their respective relationships could probably be built, but these are fairly easy to define and the hard work is generally associated with rules described by the business language. This is something that is usually very domain-specific and varies quite a bit.

Business rules in an object-oriented system are defined in terms classes, attributes, behaviors, and relationships. AOMs provided an architecture style for defining these classes (*TypeObjects*), attributes (*Properties*), behaviors (*Strategies* and *RuleObjects*), and relationships (*Accountability*) at runtime. These systems can evolve to be a very powerful *meta* language for defining your business rules such as the form-based workflow-system described by [Tilman99].

3. Implementation Issues

The primary implementation issues that need to be addressed when developing AOMs are how to store the model in a database, how to present the domain-elements to the user, and, how to maintain the model?

Adaptive Object-Models expose metadata as regular objects; it means that the metamodel can be stored in databases following well-known techniques. Object-Oriented databases are the easiest way to manage object persistence. However, it is also possible to manage the model persistence using a variety of relational databases. In the example presented in this paper the Adaptive Object-Model was distributed

among a number of different sites. Each site has its own database manager ranging from single user databases to more powerful multi-users database managers.

It is also possible to store the metadata using XML (Extensible Markup Language) or even XMI (XML Metadata Interchange Format). Note that no matter how the metadata is stored, the system has to be able to read from the storage and populate the Adaptive Object-Model with the correct configuration of instances (Figure 4).

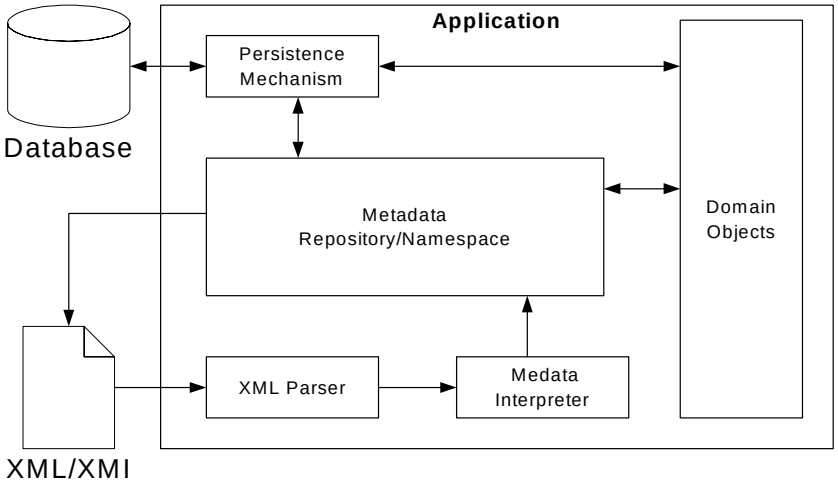

Figure 4 - Storing and Retrieving Metadata

Figure 4 shows two possible solutions for storing a retrieving metadata. Developers can treat the metadata as other domain-objects, which are mapped to a relational schema. In that case the Persistence Mechanism is responsible for storing and retrieving the metadata. Another possible solution is storing the metadata as XML or XMI documents. If no Persistence Mechanism is present for semi-structured data, developers have to build a Metadata Interpreter for populating the metamodel with appropriate objects. The interpreter instantiates business model by plugging together the objects based upon the *TypeObjects*, *Properties*, relationships between the Entities, *Strategies*, and the like. If more powerful business rules are needed such as *RuleObjects*, a second level interpreter may be needed for runtime interpretation of the business rules [Yoder2002].

The model is able to store all the metadata using a well-established mapping to relational databases, but it was not straightforward for a developer or analyst to put this data into the database. They would have to learn how the objects were saved in the database as well as the proper semantics for describing the business rules. A common solution to this is to develop editors and programming tools to assist users with using these black-box components [Roberts98]. This is part of the evolutionary process of Adaptive Object-Models as they are in a sense, "Black-Box" frameworks, and as they mature, they need editors and other support tools to aid in describing and maintaining the business rules.

4. 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]. Generative Programming deals with a wide range of possibilities including those from Aspect Oriented Programming and

Intentional Programming. Although Generative Programming does not exclude AOMs, most of the techniques deal with generating code from descriptions.

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. These techniques are different from Adaptive Object-Models 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. Generative Programming techniques that provide these types of reflection capabilities would be call Adaptive Object-Models.

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. [Riehle2001] describes a UML Virtual Machine that has an AOM to immediately reflect the changes in a metamodel.

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. There was a workshop sponsored at OOPSLA 2000 [OOPSL2000], which focused on just this topic where many papers were presented describing both working systems and research in this area.

5. 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*.

This architectural style can be very useful in systems; specifically systems that emphasizes flexibility and those that need to be dynamically configurable. They are especially useful in describing business rules that need to be changed at runtime.

6. References

[Arsanj98] A. Arsanjani, . "Meta-Modeling and Grammar-oriented Object Design", Comm. at OOPSLA 98 Workshop on Metadata and Active Object Models, October 1998.

[Arsanj99] A. Arsanjani. Analysis, "Design, and Implementation of Distributed Java Business Frameworks Using Domain Patterns", Proceedings of Tools '99 (IEEE Computer Society Press1999), pp. 490-500. http://www.computer.org/proceedings/tools/0278/02780490abs.htm.

[Arsanj2000] A. Arsanjani. "Rule Object Pattern Language". Proceedings of PLoP2000. Technical Report #wucs-00-29, Dept. of Computer Science, Washington University Department of Computer Science, October 2000. URL: http://jerry.cs.uiuc.edu/~plop/plop2k.

[Arsanj2001] A. Arsanjani. Using Grammar-oriented Object Design to Seamlessly Map Business Models to Component -based Software Architectures, Proceedings of The International Association of Science and Technology for Development, 2001, Pittsburgh, PA.

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

[Coad92] Peter Coad, "Object-Oriented Patterns". *Communications of the ACM*. 35(9):152-159, September 1992.

[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.

[Hays96] D. Hays. *Data Model Patterns, Convention of Thought.* Dorset House Publishing. 1996

[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.

[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.

[Riehle2000] D. Riehle, M. Tilman, R. Johnson. "Dynamic Object Model". Proceedings of PLoP2000. Technical Report #wucs-00-29, Dept. of Computer Science, Washington University Department of Computer Science, October 2000. URL: http://jerry.cs.uiuc.edu/~plop/plop2k.

[Riehle2001] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe. "The Architecture of a UML Virtual Machine". Proceedings of the 2001 Conference on Object-Oriented Program Systems, Languages and Applications (OOPSLA '01), October 2001.

[Roberts98] 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.

[Tilman99] M. Tilman, M. Devos. "A Reflective and Repository Based Framework". *Implementing Application Frameworks*, Wiley, 1999. On page(s) 29-64.

[Yoder2002] J. Yoder, R. Johnson. "Implementing Business Rules with Adaptive Object-Models". *Business Rules Approach*. Prentice Hall. 2002.