The Adaptive Object-Model Architectural Style

Joseph W. Yoder & Ralph Johnson
Software Architecture Group –
Department of Computer Science
Univ. Of Illinois at Urbana-Champaign
Urbana, IL 61801
yoder@refactory.com & johnson@cs.uiuc.edu

Abstract: We have noticed a common architecture in many systems that emphasize flexibility and run-time configuration. In these systems, business rules are stored externally to the program such as in a database or XML files. 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. The real power in Adaptive Object-Models is that the definition of a domain model and rules for its integrity can be configured by domain experts external to the execution of the program. These systems are important when flexibility and dynamic runtime configuration is needed, but their architectures have yet to be described in detail. This paper describes the Adaptive Object-Model architecture style along with its strengths and weaknesses. It illustrates the Adaptive Object-Model architectural style by outlining examples of production systems.

Key words: Adaptive Object-Model, Adaptive Systems, Analysis and Design Patterns, Domain-Specific Language, Architectural Styles, Components, Dynamic Object-Model, Frameworks, Meta-Modeling, Meta-Architectures, Metadata, Metalevel, Reflection, Reflective Systems.

1. INTRODUCTION

Architectures that are designed to adapt at runtime to new user requirement by retrieving descriptive information that can be interpreted at runtime are sometimes called a "reflective architecture" or a "meta-architecture". This paper focuses on a particular kind of reflective architecture that we call "Adaptive Object-Model (AOM) architecture".

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35607-5_15

J. Bosch et al. (eds.), *Software 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 with the appropriate business rules. It has been called "User Defined Product architecture" in the past [12]. These systems have also been called "Active Object-Models" [6, 20] and "Dynamic Object Models" [16]

An Adaptive Object-Model is a system that represents classes, attributes, relationships, and behavior as *metadata*. The system 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, the system stores its *Object-Model* in a database and interprets it. Consequently, the object model is adaptable; when the descriptive information is modified, the system immediately reflects those changes similar to a UML Virtual Machine described by Riehle et. al [17].

Many architects who have designed a system with Adaptive Object-Models claim that it is the best system they have ever created, and they brag about its flexibility, power, and eloquence. At the same time, many developers find them confusing and hard to work with. This is partly because an Adaptive Object-Model has several levels of abstraction, so there are several places that could be changed. The most common way to change the system is by changing the metadata, but sometimes a change is implemented by changing the interpreter of the metadata. Most programmers don't have any experience with systems like this. We hope that a clear description of the Adaptive Object-Model will help developers who have to use one. The Adaptive Object-Model architectural style has not been well 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 and the consequences of using it. It also describes four different implementations of Adaptive Object-Models that have been used for building production systems.

2. ARCHITECTURUAL STYLE OF AOMS

The design of Adaptive Object-Models differs from most object-oriented designs. Normally, object-oriented design would have 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, which 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 (*metadata*) 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* [11] provides a way to dynamically define new business entities for the system. *TypeObject* is used to separate an Entity from an EntityType. Entities have Attributes, which are implemented with the *Property* pattern [6]. The *TypeObject* pattern is used a second time in order to define the legal types of Attributes, called AttributeTypes. As is common in Entity-Relationship modeling, an Adaptive Object-Model usually separates attributes from relationships.

The *Strategy* pattern [8] is used to define the behavior of EntityTypes. These strategies can evolve into a rule-based language that gets interpreted at runtime. Finally, 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. Object-oriented systems generally 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, developers of large systems usually face the problem of having a class from which they should create an unknown number of subclasses [11].

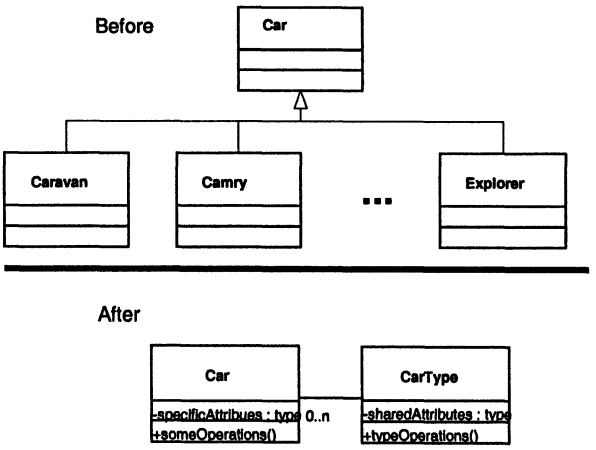

Figure 1. TypeObject

Each subclass is an abstraction of an element of the changing domain. *TypeObject* makes the unknown subclasses simple instances of a generic class (see *Figure 1*); new classes can be created at run-time by instantiating

the generic class. Objects created from the traditional hierarchy can still be created but making explicit the relationship between them and their type.

Figure 1 shows an example of how a Car class with a set of subclasses such as Caravan, Camry, and Explorer is transformed into a pair of classes, Car and CarType. These transformed classes represent the class model of the interpreter and are used at runtime to represent the Entities and EntityTypes for the system. Replacing a hierarchy like this is possible when the behavior between the subclasses is very similar or can be broken out into separate objects. In these cases, the primary differences between the subclasses are the values of their attributes.

• Property

The attributes of an object are usually implemented by its instance variables. These variables are usually defined in each subclass. 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*). This can be done using a dictionary, vector, or lookup table. In our example, the Property holds onto the name of the attribute, its type, and its current value.

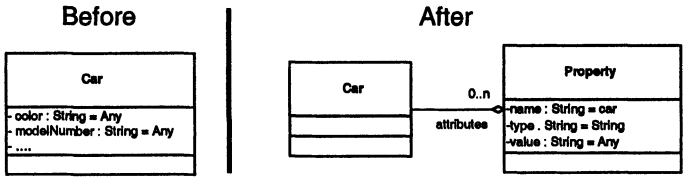

Figure 2. Properties

In most Adaptive Object Models, *TypeObject* is used twice, once before using the *Property* pattern, and once after it. *TypeObject* divides the system into Entities and EntityTypes. Entities have attributes that can be defined using *Properties*. Each property has a type, called PropertyType, and each EntityType can then specify the types of the properties for its entities. Figure3 represents the resulting architecture after applying these two patterns, which we call *TypeSquare* [22]. 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.

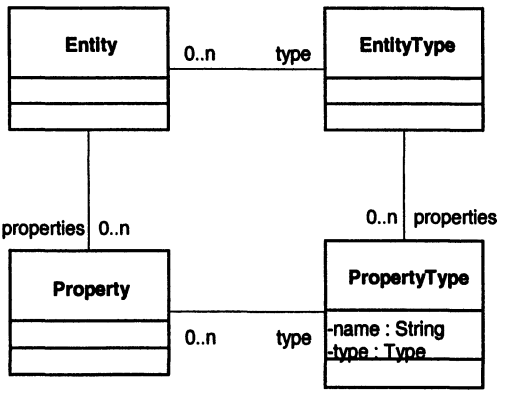

Figure 3. TypeSquare

Different kinds of objects usually have different kinds of relationships and 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 describe and change the relationships and behavior of objects.

• Entity-Relationship

Attributes are properties that refer to primitive data types like numbers, strings, or colors. Entities usually have a one-way association with their respective attributes. Relationships are properties that refer to other entities. Relationships are usually two-way associations; 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*** (see *Figure 4*) by Fowler and Hayes [7, 10]) 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. A ***Property*** whose value is an Entity represents an Association, while ***Properties*** whose value is a primitive data type are Attributes.

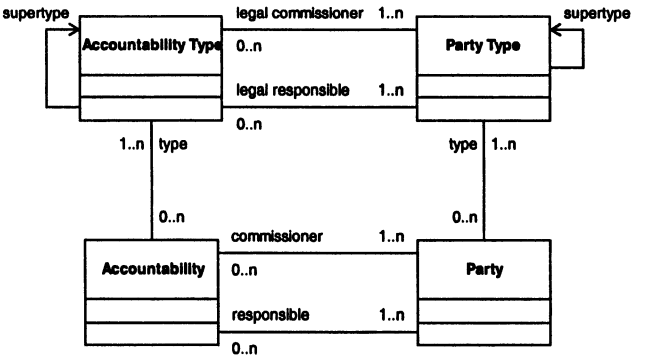

Figure 4. Accountability Pattern

• Strategies and RuleObjects

Business rules 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 Adaptive Object-Models deal with adapting these as runtime.

However, some rules cannot be defined this way. They are more functional or procedural in nature. For example, there can be 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 Adaptive Object-Models use *Strategies* and *RuleObjects* [2] to handle them.

Adaptive Object-Models 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. These can be either primitive rules or the combination of business rules through application of the ***Composite*** pattern. Rules that represent predicates are composed of conjunctions and disjunctions, rules that represent numerical values are composed by addition and subtraction rules, rules that represent sets are composed by union and intersection rules. These more complex ***Strategies*** are called ***RuleObjects.***

Figure 5 is a UML diagram created by applying the ***TypeObject*** pattern twice with the ***Property*** pattern and then adding ***Strategies*** (Rules) for representing the behavior. This resulting architecture is often seen in adaptable systems.

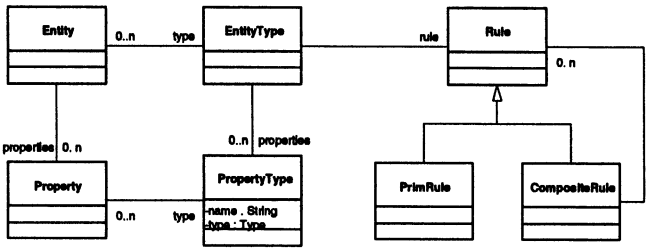

Figure 5. TypeSquare with Rules

If the business rules describe workflow, Micro-Workflow architecture as described by Manolescu [13] can be used. 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. These rules can also be built up from table-driven systems or they may be more grammar-oriented. This grammar-oriented approach has been called Grammar-oriented Object design (GOOD) [3].

• Interpreters of the Metadata

Metadata for describing the business rules and object model is interpreted in two places. The first is where the objects are constructed, i.e when the object-model is instantiated. The second is during the interpretation of the business rules at runtime.

The information about the types of entities, properties, relationships, and behaviors is stored in a database. Sometimes it is stored in XML files and we can use XDT building tools for runtime manipulation, thus allowing the model to be updated and immediately reflected in applications interpreting the data.

Regardless of how the data is stored, it must 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 *RuleObjects*. 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. 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, there needs to be ways for dealing with constants and variables, along with constraints between values. *SmartVariables* [6] can be useful for triggering automatic updates or validations when setting property values. *SmartVariables* are variables that trigger events when their values are referenced or changed.

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 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.

Rules and grammars require skill to write and maintain. But if intuitive (and following normal precedence rules that humans are used to) they can be easy for "end users" to write; at least easier than having them write a subset of a programming language. Also, special tools and visual languages can be built to support "end users" to maintain the business rules in a fashion they are familiar with.

• Architecture of Adaptive Object-Models

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* [8] is used for building dynamic tree structure types or rules. For example, if the 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.

3. EXAMPLES OF ADAPTIVE OBJECT-MODELS

We have examined many different systems that have an Adaptive Object-Model. A workshop sponsored by Caterpillar and The University of Illinois in 1998 reviewed three production frameworks that will be discussed in this paper; The Hartford's User-Defined Product Framework, Argo's Document Workflow Framework, and Objectiva's Telephony Billing System. We will also describe one that we have built and put in production for the Illinois Department of Public Health (IDPH) that is based on the *Party*, *Accountability*, and *Observation* pattern [7].

• User-Defined Product Framework

The User-Defined Product (UDP) framework is an example of an Adaptive Object-Model, which was developed at The Hartford, where it was used to represent insurance policies [12]. The UDP framework is a generic framework for "attributed composite objects". This framework makes it easy to specify, represent, and manipulate complex objects with attributes that are a function of their components. For example, an insurance policy has a price, which depends on whether it is home insurance or car insurance, the value of the home or car, the location of the home or car, the size of deductibles, and various options such as flood insurance. A bicycle manufacturer needs to describe all the models it sells, and each model has a price that is a function of the parts and options that are on it, which state the bicycle was purchased in and whether the customer is buying at retail or wholesale. Either of these systems could be built using the framework.

The UDP framework allowed users to construct a complex business object (like a new policy or a new model of bicycle) from existing components and to let users define a new kind of component without

programming. Thus, insurance managers can invent a new policy rider and an engineer at a bicycle manufacturer can invent a new add-on like a cellular phone for a bike, and neither one of them needs a programmer. Salespeople can then use these new components to specify a policy or bicycle for an order. The framework automates the computation of attributes such as price. Moreover, it keeps track of how an object changes over time, so that you know how deductions were changed on an insurance policy, and how the price of a bike changed.

The primary architecture of the UDP framework can be seen in *Figure 6* (put into UML format and modified up for this paper). Note that this is very similar to the Adaptive Object-Model architecture as described in the previous section. New insurance policies or new bicycle models can be described by creating new instances of the ComponentTypes. These can in turn define what their legal AttributesTypes. We have applied the *Strategy* pattern on ComponentTypes and AttributeTypes in order to define the legal rules (behaviors) that are associated with them. These rules can become very dynamic in nature and can be composed together. They can consist of binary operations, table-lookup, or simple constraints.

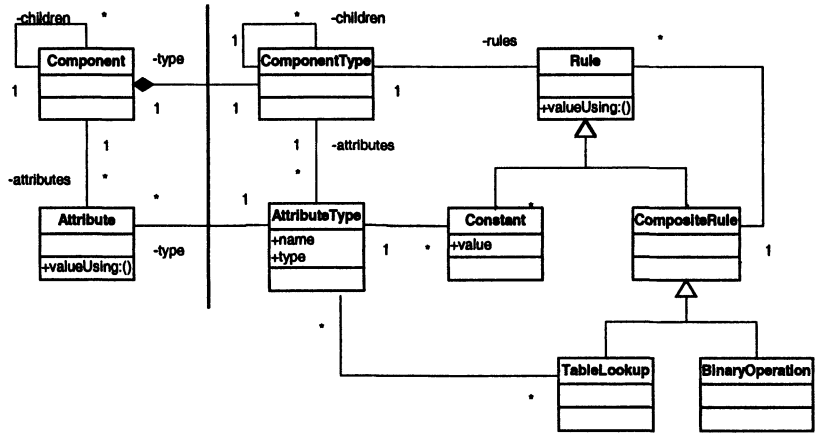

Figure 6. UDP Framework Architecture

• Argo Document Workflow Framework

This Argo framework was developed to support the Argo administration in Belgium. Argo is a semi-government organization managing several hundred public schools. It uses this framework to develop its applications, which share a common business model, and require database, electronic document, workflow and Internet functionality.

The framework is based on a repository in two ways. First, it consists of a set of tools for managing a repository of documents, data and processes, including their history and status. These tools enable users to select

applications, enter and view data, query the repository, access the thesaurus and manage electronic documents, workflow processes, and task assignments. More importantly, the framework behavior is driven by the metadata stored in the repository. The repository captures formal and informal knowledge of the business model. None of the business model is hard coded. The tools consult the repository at runtime. End-users can change the behavior of the system by using high-level tools to change the business model. Thus we separate descriptions of an organization's business logic from the application functionality.

The framework helps to develop applications iterative and interactively. It enables the evolution of increasingly complete specifications of applications that can be executed at once. The resulting system is easy to adapt to changes in the business. Argo can develop new applications through modeling and configuration, rather than through coding.

The primary architecture of the framework is based upon a meta-layer where there is support for defining new types of objects, with their attributes and behaviors (see *Figure 8*). This architecture is very powerful as the meta-level was pushed to the limit for defining new types of objects with their respective behaviors.

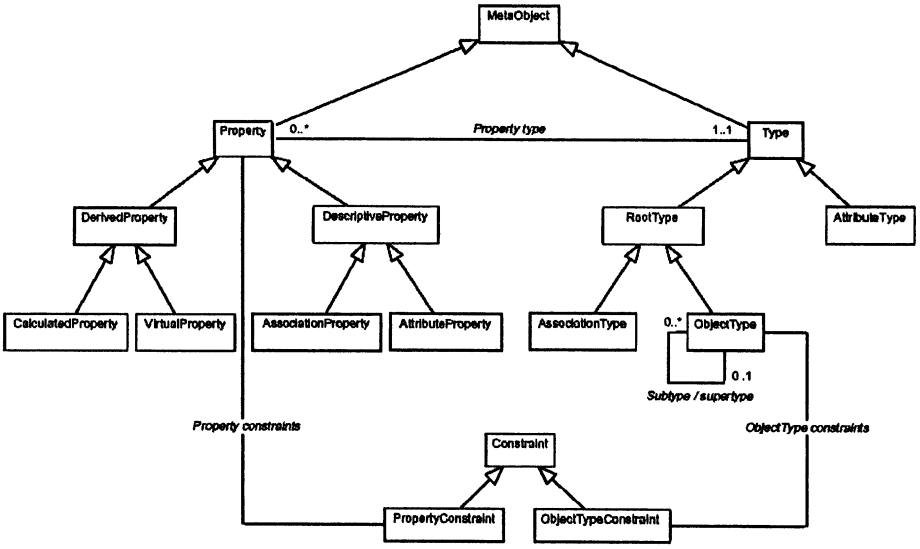

Figure 7. Argos Meta-Architecture

The design of the system involves a MetaObject class, which has two subclasses of Type and Property. Types can have zero or more Properties. There are subclasses of Type, which are AttributeType, ObjectType, and AssociationType. And, there are subclasses of Property, which are DerivedProperty and DescriptiveProperty that in turn have subclasses of CalculatedProperty, VirtualProperty, AssociationProperty, and AttributeProperty. There are also Constraints between the Properties and Types.

The rules are defined by a more dynamic type of ***Strategy*** that allow for ScriptRules, SystemEvents, and Constraints to be configured together and associated with different types of objects and attributes (see Figure 8). These are more closely related to ***RuleObjects*** rather than ***Strategy*** as they define a set of scripting rules and events for defining the specific rules for each ObjectType.

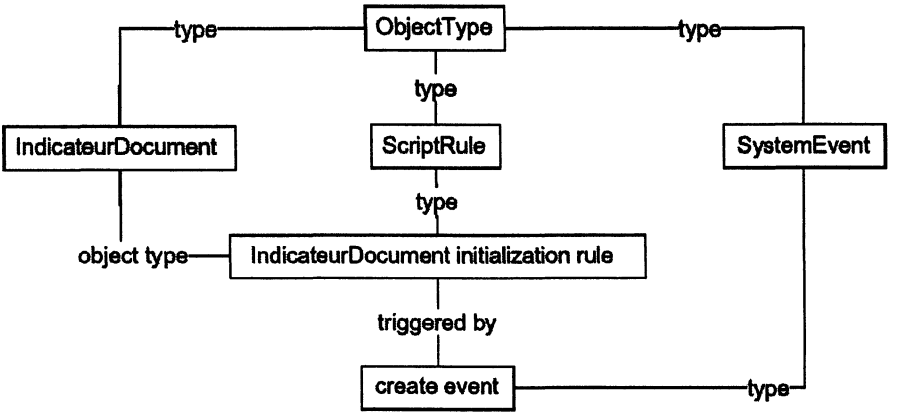

Figure 8. Argos Business Rules

• Objectiva's Telephony Billing System

Objectiva is a black-box framework for telecommunications billing. This system allows developers to build applications primarily by reusing existing classes and does not force them to create new ones. Objectiva makes it possible to quickly produce billing systems for all kinds of telecom services, including cellular, PCS, local number portability, conventional local and long distance, and satellite services. It also makes it possible to quickly customize an existing system to respond to changing conditions and to provide new services. It is a "convergent billing" system that makes it possible for a single billing system to handle any kind of telecommunications service.

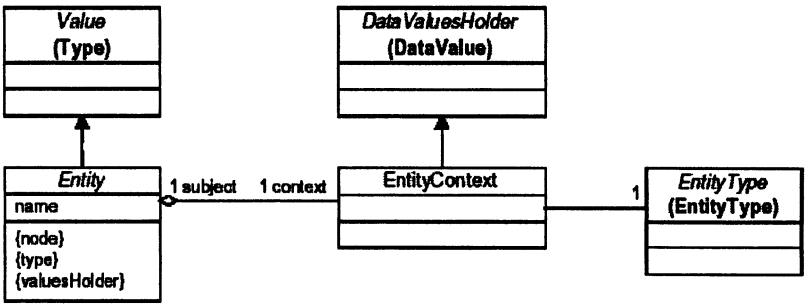

Figure 9. Objectiva Business Entity Model

A billing system has many parts, some technical in nature, and some that solely implement the business rules of billing. The purpose of the Objectiva architecture is to organize the parts of the system as effectively as possible in order to maximize their reuse. In Objectiva, as in Smalltalk, everything is an object, with all the jargon that one expects in order to be fully object-oriented: inheritance, encapsulation, polymorphism, responsibilities, collaboration, etc.

Objectiva keeps track of a company's customers. This includes their addresses and other contact information, the agreements that each customer has with the Enterprise (which can change frequently), the network events that cause a charge (like a local or long distance call, a page, or an e-mail message), taxes, discounts, invoices sent, and payments received. It manages equipment that is being rented or purchased, which means not only charging for it, but keeping track of its location and managing an inventory of equipment available for rental or purchase, and scheduling repairs on equipment that is broken. Objectiva manages products, which are combinations of the various pricing plans that a company is offering to customers. It connects to other systems for accounting, to get network events, and to load subscriber information on the switch. Objectiva is a complex information system, but it is made up of a fairly small number of highly reusable classes. This is a key to its flexibility and power.

Figure 9 outline the core architecture needed for creating new types of Entities for the system. Notice that *TypeObject* is being used for declaring new types of objects. Here we can see a variation from the Adaptive Object-Model architecture mentioned in the previous section. Notice that Entities have a Context and the context hold onto the DataValues (attributes).

Attributes are broken down into many different types based upon the DataValue types as can be seen in Figure 10. Here we are looking at primarily DiscreteAttributes though there are also MeasurementAttributes, and ContinuousAttributes. An Adaptive Object-Model can vary the *Properties* quite a lot by the use of subclasses as outlined in Figure 10.

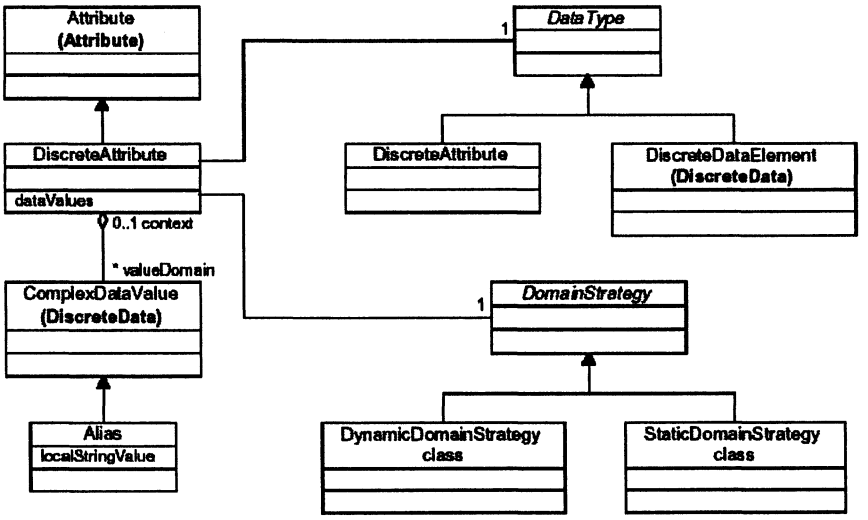

Figure 10. Objectiva Data Values (Attribute) Model

Relationships were modeled as first class entities as shown in Figure 11. The knowledge level (business rules) of this system primarily focused around EntityTypes, Attributes, and their Relationships. Those there are also EventTypes (not shown) that were basic *Strategies* to be fired under certain events or conditions.

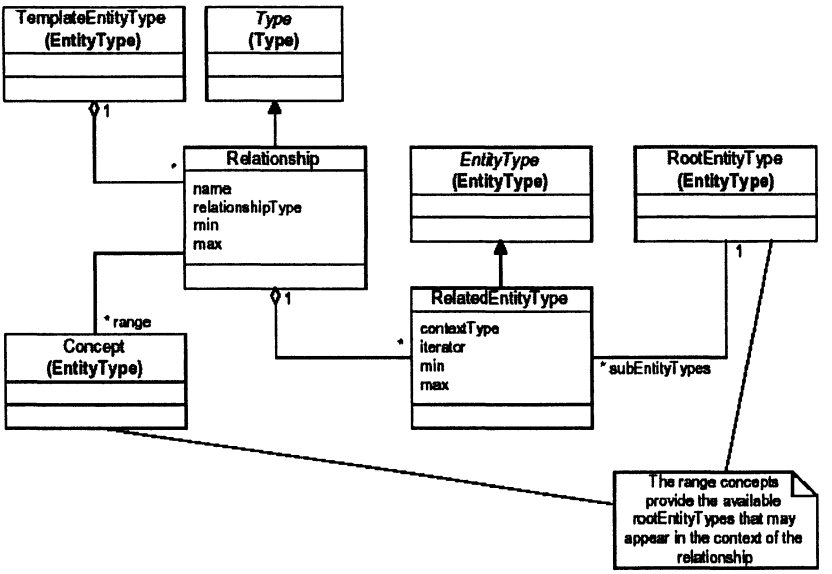

Figure 11. Objectiva Entity-Relationship Model

Objectiva has dozens of subclasses of Entity and EntityType. This is quite different from the other examples we have described. It should also be noted that the primary architecture of Objectiva had different versions of Smalltalk code for describing the metadata and new versions of the business rules (by writing methods) rather than pushing the data into a database. This was because it was just as easy and fast to build and release this part of the application through the evolution of Smalltalk code rather than being forced to push these changes to a database.

• IDPH Medical Domain Framework

Many applications at the Illinois Department of Public Health (IDPH) manage information about patients and people close to the patient, such as parents and doctors. The programs vary in the kind of information (and the representation) they manage. However, there are core pieces of information that is common between the applications and can be shared between applications.

Typically, the information being shared for the IDPH applications is a set of observations [7, 10] about people and also relationships between people and organizations. An observation describes phenomenon during a given period of time. Observations play a large role in the medical domain because they associate specific conditions and measurements with people at a given point in time. Some typical medical observations are eye color, blood pressure, height and weight. Some more specific types of information can be seen for following up high-risk infants. This application captures observations about the infant and the infant's mother such as HIV status, drug and alcohol history, hepatitis background, gestational age, weight at birth and the like.

One way to implement observations is to build a class hierarchy describing each kind of observation and associate these observations with patients. This approach works when the domain is well known and there is little or no change in the set of observations. For example, one of the applications at IDPH screens newborn infants for genetic diseases. In this application, certain observations about the infant are made at the time of birth such as height, weight, feeding type, gestational age, and mother's hepatitis B indication. A natural implementation would be to create a class for the infant, and create another set of classes and associate them with the infant to capture the observations (see Figure 12).

However, whenever a business rules changed, or new type of information needed to be collected, either a new class would need to be developed or an existing class would need to have its description and methods changed, and then the system would need to be recompiled and released to users of the system. For example, the Refugee System that we developed for tracking

Refugee's as they entered the country has over 100 different types of observations.

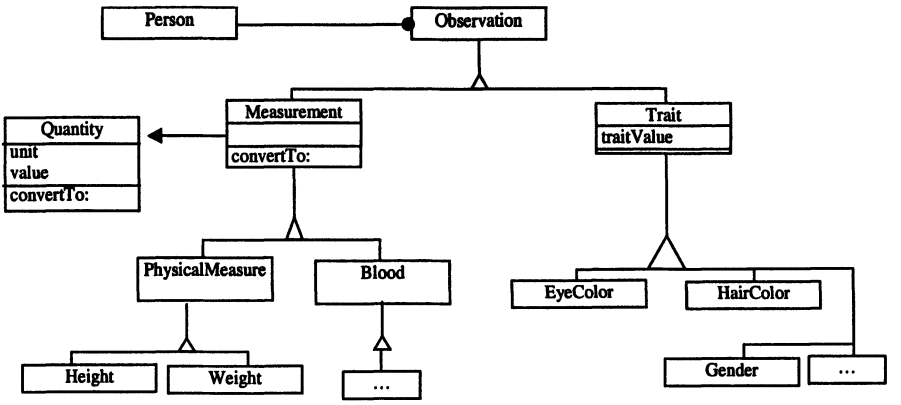

Figure 12. Basic Observation Model

This led to the creation of an Adaptive Object-Model by applying and evolving Fowler's Observation Model [7]. Figure 13 shows the resulting class diagram for the implementation of observations with Validators. Observations can either be PrimitiveObservations (basic values such as eye color, weight, etc.) or they can be CompositeObservations (composed from other Observations such as Blood Pressure which is a Systolic value over a Diastolic value). Each Observation has its ObservationType associated with it, which describes the structure of the Observation and hangs on to the validation rules through its relevant Validator. Therefore, the ObservationType is used to validate the structure and the values.

The ObservationType takes care of the structural properties of the Observation that it is describing. For example, CompositeObservationType is used to create and validate any kind of CompositeObservation and defines the structure of the CompositeObservation. PrimitiveObservationType is used to describe the possible quantity or discrete values and the validation rules for each. PrimitiveObservations also have been enhanced to allow for multiple values. For example, the language(s) that a person understands could be a set of DiscreteValues.

RangedValidators also have an interval set of Quantities, which describe the sets of valid values for the Measurement ObservationTypes. The validation business rules for CompositeObservationType checks if all of its components are valid. This could be enhanced to allow for a composite function for validating these types of observations.

Each ObservationType knows how to create instances with its type. PrimitiveObservations have a trivial structure, but CompositeObservations,

the structure has to be correctly established. This is a typical implementation of a *Factory* for creating Observations when using *TypeObjects.*

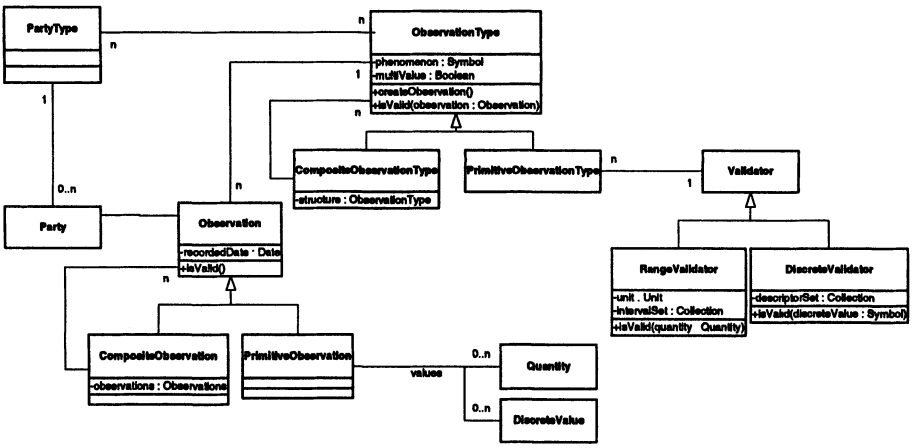

Figure 13. Class Diagram of the Observation Adaptive Object-Model Architecture

In this system, there are Parties (people and organizations) that represent the Entities for the Adaptive Object-Model, while the *Properties* are represented by the Observations. The Observations are a special type of *Propertiy*, which are values with history. There are many other business rules that are described elsewhere [22, 23] that involve the types of legal people and organizations (Parties), relationships between the Entities, and the valid set of *Properties.*

• Similarities and Differences of Examples

All these systems allow for the creation of objects with attributes using the *TypeObject* and *Property* patterns. The Argos, IDPH, and Objectiva Systems modeled relationships as first class entities, while the UDP Framework did not.

All systems had some type of *Strategy* for defining the business rules, however this is where the architectures varied the most. Different problem domains require different kinds of strategies and lead to different kind of business rules. The business language becomes very domain-specific for each Adaptive Object-Model and the way the rules are described depends on the domain.

Also, GUI issues and tools are domain specific. The types of GUIs needed and the types of tools vary among these systems. There are related ideas that can be shared but in general, this is the hard part of building an Adaptive Object-Model.

4. IMPLEMENTATION ISSUES

The primary implementation issues that need to be addressed when developing Adaptive Object-Models are how to store and represent the model in a database, how to present the domain-elements to the user, and, how to maintain the model.

• Making Models Persistent

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 relational database.

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 it and populate the Adaptive Object-Model with the correct configuration of instances (Figure 7). This is sometimes done using builders and interpreters for creating and configuring the Entities, Attributes, Relationships, and Behaviors from the meta-description.

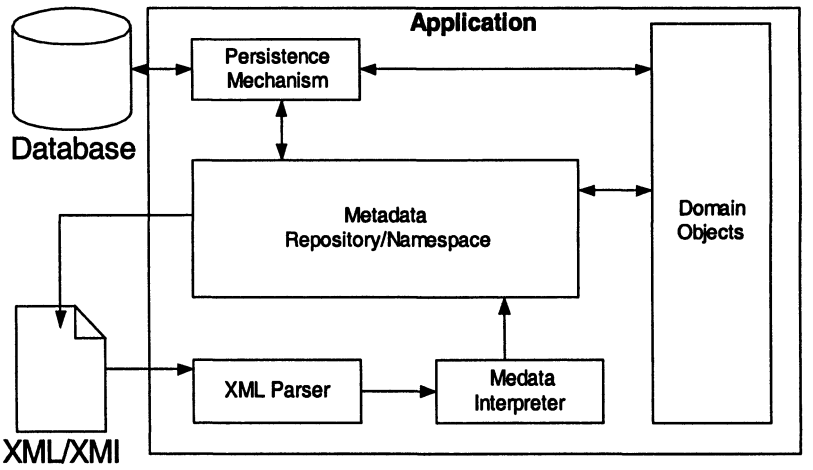

Figure 14. Storing and Retrieving Metadata

• Presenting the Model to Users

GUI issues also need to be considered when implementing dynamic architectures. The Adaptive Object-Model described in the previous section makes it easy to create new domain objects but the values still need to be entered and presented to an "end users" through some user interface.

Regardless of the types of objects the application needs, some sort of view needs to be developed for the user. These views could all be developed for each application without getting any benefits of reuse. However, it is possible to extend EntityTypes to allow for some standard views and make building GUIs for the business objects easier.

An example of this can be seen in the Refugee application that we built for IDPH. This application captures over 100 observations about refugees as they enter the country. The observations in this example are both ranged values (such as Blood Pressure, Pulse, and Temperature), and discrete values (such as Heart, Lungs, Abdomen, and Skin).

However, even when the team learned how to use the model, they still had to decide how to present these observations to an end-user and to make sure that only meaningful values were collected about refugees. It took the team five weeks of hard work to wire the first set of observations to a GUI. While the problems were being worked out, management was wondering why new ideas such as *Mediators* and *Adaptors* [8] were being presented late in the game and why was it taking us so long. In reality it was just the naming issue that was new to management and we were using well known patterns for solving a mapping problem that the analysis model never revealed The real gain came after they worked this problem out, as they were able to then hook up over 90 observations in less than two days. People then started seeing the power of the model as it was now easier to change the business rules and there were not 100 classes to code, debug, and maintain.

The moral of the story is that it can be easy to overlook GUI design issues. Analysts and developers can get caught up in the design of the business objects and overlook the user interface. Adaptive Object-Models provide powerful techniques for building the business objects, but mapping them to a GUI can be difficult since there are higher-level abstractions. It is possible to design dynamic GUIs but this can be difficult and is usually domain specific.

• Maintaining the Model

The observation 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 [18]. 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.

- **History of Rules and Data**

Some other important implementation issues are how to deal with the history of the actual data values and how to deal with versions and history of the rules.

A common way to deal with history for data values is to keep track of how the values change over time and change the *Interpreter* so that it always picks the correct version when reading the values. The rules can be treated similarly by including history information with the rules and make sure that the interpreter always uses the right version of the rules. This is especially important when values might be valid at one point in time, but invalid at another. There are many patterns for dealing with history of this sort[1, 4].

5. CONSEQUENCES OF AOMS

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 users 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 reduce the number of classes, and so make it 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 main business case for an Adaptive Object-Model is to make it possible to develop and to change software quickly. Adaptive Object-Models reduces 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.

An Adaptive Object-Model also has several disadvantages. They can take more effort to build. Adaptive Object-Models generally need tools and support GUIs for defining the objects in your system. 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.

They can also be harder to understand since there are two co-existing 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 most information about the business is in the database.

Adaptive Object-Models leads to a domain-specific language. Although it is often easier for users to understand a domain-specific language than a general-purpose language, developers of an Adaptive Object-Model inherit all of the problems associated with developing any language such as providing debuggers, version control, and documentation tools.

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.

Finally Adaptive Object-Models can be harder to maintain. This is usually the case when the primary architect leaves and the developers designated to maintain the system do not understand these types of architectures. However, developers that understand these architectures find Adaptive Object-Models easier to maintain since there is generally less code to maintain and typically a small change to the system can make for a large change in the running application.

6. ALTERNATIVES AND RELATED WORK

The best-known alternatives or related techniques to Adaptive Object-Model Architectures are Code Generators, Generative Programming, Metamodeling, and Reflective Techniques.

Code generators produce either executable-code or source-code. This technique focuses on the automatic generation of systems from high-level descriptions. 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 and Metamodeling are fields of study that have been looking at different techniques for giving users the same power as those given by Adaptive Object-Models. It is interesting to note that none of the literature from these fields has described Adaptive Object-Models, although Adaptive Objects-Models are widely used for building related architectures that are used for production systems.

Generative Programming [5] provides infrastructure for transforming descriptions of a system into code. Generative Programming deals with a wide range of possibilities including those from Aspect Oriented Programming and Intentional Programming. Although Generative Programming does not exclude Adaptive Object-Models, most of the techniques deal with generating code from descriptions. Descriptions are based on provided primitive structures or elements and can evolve to become a visual language for the domain.

Metamodeling techniques include a variety of approaches most of which are generative in nature. So far, metamodeling has been more theoretical in nature focusing on ways to create metamodels for creating models. In other words, it is describing ways to create languages for generating models that can then be realized. In general, these techniques focus on manipulating the model and meta-model behind a system as well as supporting valid transformations between representations [14]. Quite often the attention is on the meta-model, or a model for generating a model, rather than the final application that will reflect the business requirements.

Metamodeling techniques 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 [15]. The primary difference is that Metamodeling techniques as provided by CASE tools generate the code from the descriptive information 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. There has been related work with UML, which is the closest metamodeling language towards being realized for working systems. A good example can be seen by the work on the UML Virtual Machine that has an Adaptive Object-Model, which immediately reflect the changes in a metamodel.

Finally, one could make the argument that database systems, are examples of Adaptive Object-Models. Database schemas are not hard-wired, but are interpreted. Database objects are not objects really, but data stored in a database. The key problem with databases is attaching method to these objects.

7. CONCLUSIONS

The Adaptive Object-Model Architectural Style provides an alternative to usual object-oriented design. Conventional 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 *TypeObject*, *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 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. We think that part of this mismatch is because the architectural style is not widely understood.

This paper describes the architectural style of Adaptive Object-Models, including some examples 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.

8. ACKNOWLEDGEMENTS

We would like to thank the many people whose valuable input greatly improved this paper; specifically we would like to thank: Ali Arsanjani, Federico Balaguer, John Brant, Krzysztof Czarnecki, Brian Foote, Martin Fowler, Alejandra Garrido, Mike Hewner, Dragos Manolescu, Brian Marick, Reza Razavi, Nicolas Revault, Dirk Riehle, Don Roberts, Andrew Rosenfeld, Gustavo Rossi, Weerasak Witthawaskul, and Rebecca Wirfs-Brock.

9. REFERENCES

[1] Francis Anderson. "A Collection of History Patterns". *Pattern Languages of Program Design 4.* Addisson Wesley, 2000.

[2] Ali 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.

[3] Ali 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.

[4] Andy Carlson, Sharon Estepp, and Marin Fowler. "Temporal Patterns". *Pattern Languages of Program Design 4.* Addisson Wesley, 2000.

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

[6] Brian Foote, Joseph W. 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.

[7] Martin Fowler. *Analysis Patterns, Reusable Object Models.* Addison-Wesley. 1997.

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

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

[10] David Hay. *Data Model Patterns, Convention of Thought.* Dorset House Publishing. 1996

[11] Ralph Johnson, Bobby Wolf. "Type Object". *Pattern Languages of Program Design 3.* Addisson Wesley, 1998.

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

[13] 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.

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

[15] Nicolas Revault & Joseph W. Yoder. "Adaptive Object-Models and Metamodeling Techniques", ECOOP'2001 Workshop Reader; Lecture Notes in Computer Science, Springer Verlag 2001.

[16] 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.

[17] 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.

[18] Don Roberts, Ralph Johnson. "Patterns for Evolving Frameworks". *Pattern Languages of Program Design 3.* Addisson Wesley, 1998.

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

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

[21] 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.

[22] Joseph W. Yoder, Federico Balaguer, Ralph Johnson. "Architecture and Design of Adaptive Object-Models", Intriguing Technology Presentation at the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01); ACM SIGPLAN Notices, ACM Press, December 2001.

[23] Joseph W. Yoder, Ralph Johnson. "Implementing Business Rules with Adaptive Object-Models". *Business Rules Approach.* Prentice Hall. 2002.