Architecture and Design of Adaptive Object-Models

Joseph W. Yoder
Software Architecture Group –
Department of Computer Science
Univ. Of Illinois at Urbana-Champaign
Urbana, IL 61801
yoder@refactory.com

Federico Balaguer
Software Architecture Group –
Department of Computer Science
Univ. Of Illinois at Urbana-Champaign
Urbana, IL 61801
balaguer@cs.uiuc.edu

Ralph Johnson
Software Architecture Group –
Department of Computer Science
Univ. Of Illinois at Urbana-Champaign
Urbana, IL 61801
johnson@cs.uiuc.edu

ABSTRACT

Many object-oriented information systems share an architectural style that emphasizes flexibility and run-time adaptability. Business rules are stored externally to the program such as in a database or XML files 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. The real power in Adaptive Object-Models is that they have a definition of a domain model and rules for its integrity and can be configured by domain experts external to the execution of the program. This paper describes the Adaptive Object-Model architecture along with its strengths and weaknesses. It illustrates the Adaptive Object-Model architectural style by describing a framework for Medical Observations (following Fowler's Analysis Patterns) that we built.

Keywords

Adaptive Object-Model, Adaptive Systems, Dynamic Object-Model, Reflection, Reflective Systems Meta-Modeling, Meta-Architectures, Metadata, Patterns.

1. 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 [20].

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 [8]. 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 [12]. These systems have also been called "Active Object-Models" [5] and "Dynamic Object Models" [17]. 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*. 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, 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 [12], to bill for telephone calls, and to check whether an equipment configuration is likely to work. It has been used to model workflow [13, 23], 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 projects applying Adaptive Object-Models sometimes present a serious gap between architects and developers. This gap includes not only different point of views (design vs. implementation) but also a more fundamental one such as abstractions and elements of the problem. This is due partly because an Adaptive Object-Model has several levels of abstraction, so there are several places that could be changed. An Adaptive Object-Model has extra machinery to interpret and execute rules, and to define relationships and attributes of entities. These definitions are external to the running program. So there is a disconnection between the model and its behavior (it is indirect). The programmer is building a machine to execute a model, not building the model itself. This is not what the programmer is used to doing. Perhaps some developers are just not capable of working on these kinds of projects. However, we feel that the problem is caused by a poor understanding of this technology. It has more to do with a clear understanding of the problem, the representation of the solution, and the design to solve it. 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 details of the Adaptive Object-Models architectural style. It also describes the 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. ARCHITECTURUAL STYLE OF AOMS

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. *TypeObject* [11] 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. 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]. Each subclass is an abstraction of an element of the changing domain. The differences between the subclasses are small and can be parameterized by setting values or objects representing algorithms. *TypeObject* makes the unknown subclasses simple instances of a generic class (see Figure 1); new classes can be created dynamically at run-time by means of instantiation of 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 presents a simple example where a given hierarchy is represented with the class EntityType and its instances with the class Entity. Replacing a hierarchy like this is possible when the behavior between the subclasses is very similar or can be broken out into separate objects. Therefore, the primary differences between the subclasses are their attributes.

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 [4]. 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 Figure 5). 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.

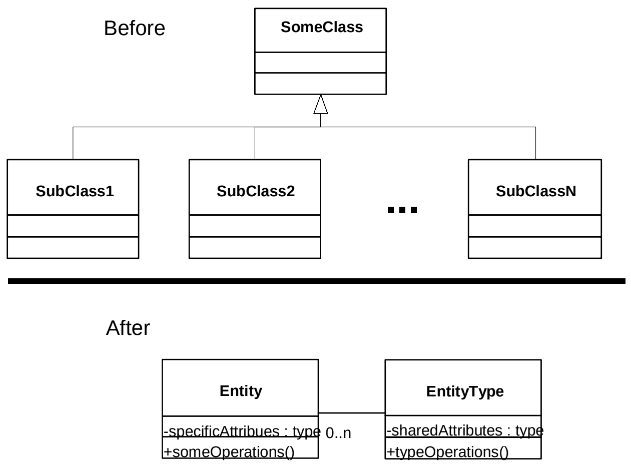


Figure 1 - TypeObject

• Property

The attributes of an object are usually implemented by its instance variables. A class defines its instance variables. 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.

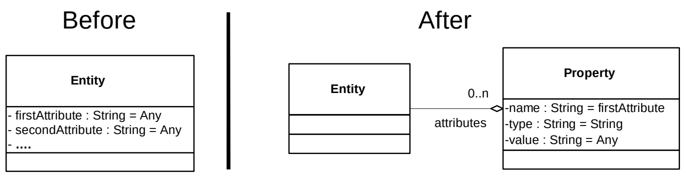


Figure 2 - Properties

Most Adaptive Object-Model architectures contain the *TypeObject* and *Property* [5] patterns. The *TypeObject* pattern 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. Figure 3 represents the resulting architecture after applying these two patterns, which we call *TypeSquare*. 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.

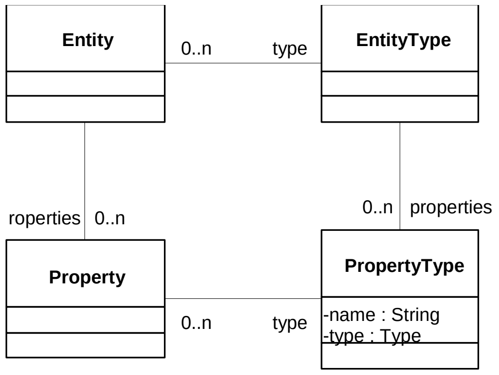


Figure 3 - TypeSquare

Different kinds of objects usually have different kinds of 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 behavior of objects.

• Strategies and RuleObjects

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.

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.

Figure 4 is a UML diagram of applying the *TypeObject* pattern twice with the *Property* pattern and then adding *Strategies* or *RuleObjects* [1] for representing the behavior. This resulting architecture is often seen in adaptable systems with knowledge levels as described in this paper.

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. 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) [2]. These more complicated rules along with dynamic GUIs are usually the hardest part of Adaptive Object-Models and are why there are not generic frameworks for them [27].

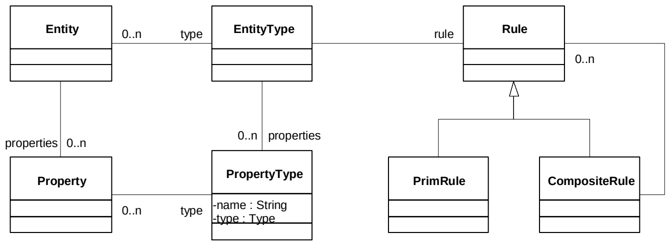


Figure 4 – TypeSquare with Rules

• Entity-Relationship

Attributes are properties that usually refer to primitive data types like numbers, strings, or dates. These associations are usually one way. 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* 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. A Property whose value is an Entity represents an association, while a Property whose value is a primitive data type is an attribute. It is interesting to note that while few language designers seem to feel the need to represent these relationships, most designers of systems with Adaptive Object-Models do. Representing these relationships as objects allows for runtime manipulated, thus allowing for a power user to immediately adapt these entity-relationships to changing business requirements.

• User Interface for Defining Types

One of the main reasons to design an Adaptive Object-Model is to enable users and domain experts to change the behavior of the system by defining new entities, relationships, and rules. Sometimes the goal is to enable users to extend the system without programmers. But even when only the developers will define new entities and rules, it is common to build a specialized user interface for defining them. 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 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 and rule information is to give them simple tools to do so.

• Design of Adaptive Object-Models

The design of Adaptive Object-Models involves three major activities: defining the business entities, rules and relationships; developing a design of an engine for instantiating and manipulating these entities according to their rules in the application; and developing tools for describing these entities, rules and relationships.

These activities are achieved by applying one or more of the previously mentioned patterns in conjunction with other design patterns such as *Composite*, *Interpreter*, and *Builder*. *Composite* [7] 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, and then you know the patterns and can understand them.

3. AOM Example for Medical Observations

Other people have built most of the Adaptive Object-Models that we have seen. However, we have built some and one in particular that will be described in this paper is based on the Observation pattern. 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 [6, 9] about people. 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.

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.

Another application at IDPH is for following up high-risk infants. This application needs to capture 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.

• Type Objects and Properties

Many of our applications shared a similar hierarchy and required that observations captured in one program to be eventually used in other applications. We also noted that there were really two basic kinds of observations. One kind deals with a discrete set of values such as blood type and eye color. The other kind has continuous values such as weight in grams or height in feet and inches.

Figure 5 is a UML class diagram for implementing the first complete observation model presented in Fowler's book (page 43). His model has a class called CategoryObservation, which is linked to a Category. In our model CategoryObservations are replaced by Traits and Categories are replaced by literal values, thus we do not have this class in our model.

Our IDPH System includes the *Property* pattern since an *Observation* is a *Property* of a Person. *Observations* have features such as a duration that most *Properties* do not have, so not all properties are observations, but all observations are properties.

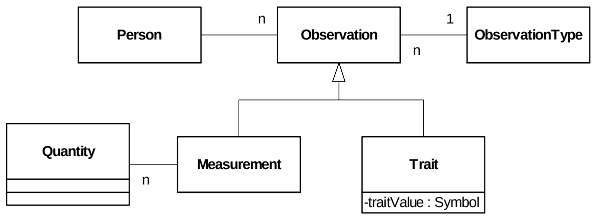


Figure 5 - Class Diagram of the Basic Observation Model

Figure 6 is a simple Instance Diagram with an example of a Person named Smith. Smith has a height observation with a value of 5 feet and an eye color observation with a value of blue. Note that the first Observation has an ObservationType object, which has a phenomenonType of #height, and a Quantity object representing 5 feet, while the second Observation has a value of blue and an ObservationType with phenomenonType of #eyecolor. Whenever a value is entered for an Observation, the Observation will use its ObservationType for validation.

As shown in Figure 6, there is an instance of ObservationType for each kind of observation. Thus, to add a new kind of observation, create a new instance of ObservationType and add it to the model. This allows a new type of observation to be created without requiring a new version of an application.

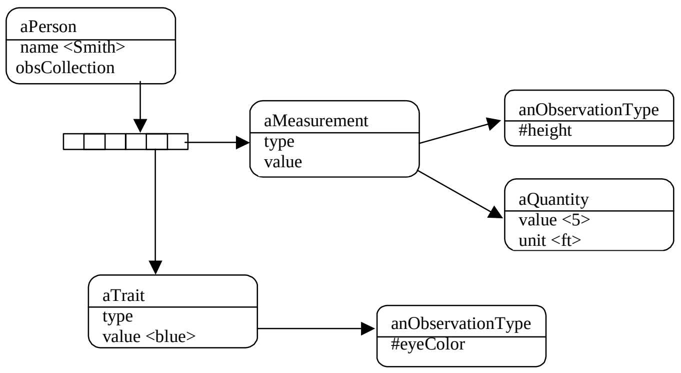


Figure 6 - Instance Diagram of the Basic Observation Model

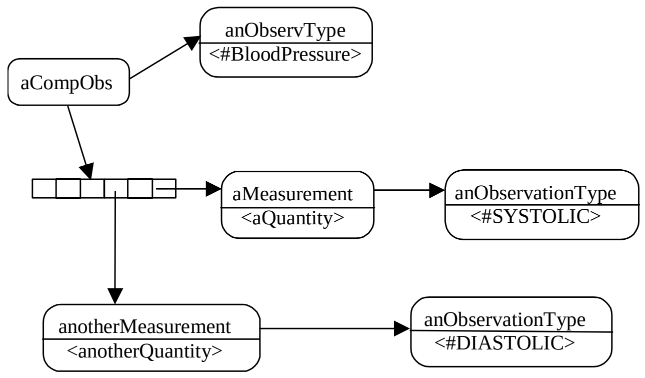


Figure 7 - Example of Blood Pressure

• Applying the Composite Pattern

The model presented in Figure 5 can be used to represent most observations. However, some observations are more complicated. For example, a "cholesterol" observation for a patient is composed of two independent measures; HDL and LDL. HDL and LDL can also be modeled as observations. Often, the HDL observation is used independently and the LDL observation is only considered when the HDL observation value is high.

Another example of a composite observation is blood test results. A single blood test can contain many observations. Some may be measurements such as white blood cell count while some may be traits such as blood types. Others could be an overall observation of a patient's health that includes observations such as blood pressure, pulse, vision, and reflexes.

Fowler's model does not directly consider these multi-value observations. Fowler allows observations like this to be modeled as independent observations, but doesn't treat a group of observations as a single observation. His model allows compound units for observations, which represent values such as area (square yards) and speed (feet per second). But composite observations are observations that are composed of independent observations. From the user's perspective we can say that the units of atomic observations do not need to be combined.

Fowler's model also allows for observations to be associated, which allow for observations to be linked to each other in a diagnostic manner (for example, thirst indicates diabetes). This is a useful concept in the medical domain but it still did not match how we wanted to represent multi-valued observations.

The *Composite* pattern allows observations to be composed of other observations. Therefore, a cholesterol observation can be composed of two atomic observations for HDL and LDL. This composition allowed us to capture the compound units for observations, and allowed us to describe more complicated observations like those above. It also makes it easier to use observations such as HDL both independently of the cholesterol observation and as a part of it.

An instance of CompositeObservation can contain any kind of observation (*Composite Pattern*). In this way it is possible to define a complex Observation based on basic/atomic ones. The "diastolic pressure" and the "systolic pressure" observation of the "blood pressure" observation can be a composite observation as shown in Figure 7.

Notice that the instance of CompositeObservation (with type #BloodPressure) and the instances of Measurement are associated with different instances of ObservationType (in our example there is aMeasurement with #SYSTOLIC and anotherMeasurement with #DIASTOLIC). This helps clarify the difference between these observations.

• Applying the Strategy Pattern

An ObservationType places constraints on the value of an Observation with that type. One way to implement these constraints is to make the ObservationType responsible for checking whether the value of an Observation is legal. Each ObservationType knows its set of possible values. But some of the sets are the same for different ObservationTypes, e.g. any observation quantifying the presence of an illness has three possible values such as YES, NO, UNKNOWN. These sets are Validators; they are objects that can tell whether a value is valid.

Just as there are two kinds of Observation types for Measurements and Traits, there will be two kinds of Validators. One represents a range of Measurements, and one represents a set of Traits. A Validator is just an algorithm telling whether a value is valid, therefore it is a *Strategy*. The resulting architecture for Validators is shown in Figure 8.

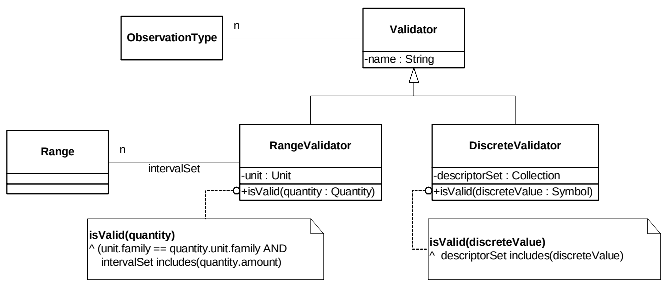

ObservationType
n
Validator
-name : String
Range
n
intervalSet
RangeValidator
-unit : Unit
+isValid(quantity : Quantity)
DiscreteValidator
-descriptorSet : Collection
+isValid(discreteValue : Symbol)
isValid(quantity)
^ (unit.family == quantity.unit.family AND
intervalSet includes(quantity.amount)
isValid(discreteValue)
^ descriptorSet includes(discreteValue)

Figure 8 - Architecture for Observation Validation

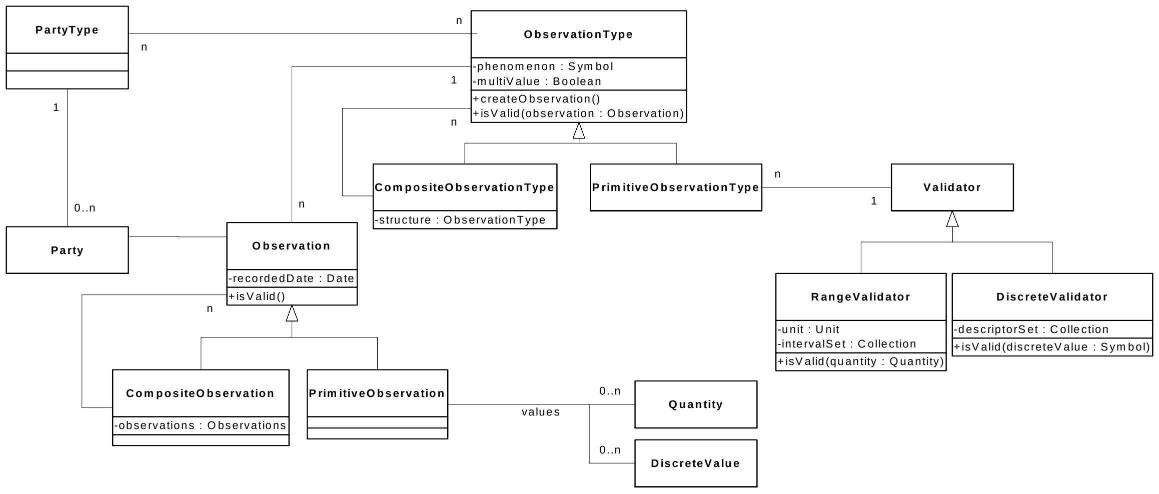


Figure 9 - Class Diagram of the Final Architecture

DiscreteValidator and RangedValidator are just describing the difference between the types of values they are expecting to store in the #observationValue variable. Therefore, rather than having two classes for representing traits and measurements, a single class representing PrimitiveObservations can model Measurements and Traits. Whether an ObservationType represents a trait or a measurement depends upon the class of its Validators (DiscreteValidator and RangedValidator).

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 parameterizing 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.

• Final Observation Architecture

Figure 9 shows the resulting class diagram for the implementation of observations with Validators. Observations can either be PrimitiveObservations or they can be CompositeObservations. 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. Note that instances of PrimitiveObservations class do not understand the #convertTo: method, this responsibility is directly delegated to the Quantity associated with the Observation. 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 ObservationTypes. The validation 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*.

In the final architecture (Figure 9) the relationship between *PartyType* and *ObservationType* is an example of TypeSquare. It is useful for defining the set of observations that any given Party within the system can have attached.

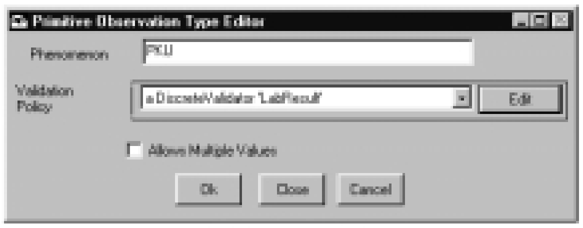
• User Interfaces for Defining Types

We provided tools for creating, and maintaining instances of the ObservationType hierarchy and Validator hierarchy. Software developers and analysts can create new types of observations and define the validation business rules easily.

Instances of ObservationType are created using the PrimitiveObservation Type Editor (Figure 10 a) and CompositeObservation Type Editor (Figure 10 b). They had a developer or business expert define the types of observations used in the application.

Instances of DiscreteValidator and RangedValidator use the Discrete Validator Editor and the Ranged Validator Editor (Figure 11) to assist developers and business experts with creating and maintaining the validation business rules.

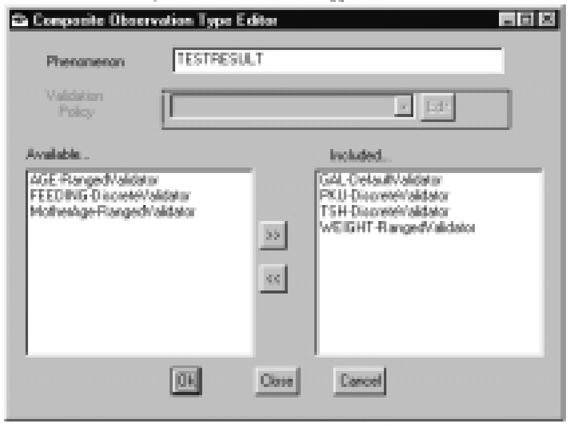
CompositeObservations are created by selecting from a list of observations that are not already part of the component. PrimitiveObservations are named and associated with a Validator. The Validators define the primitive types of observation values and their relevant validating business rules.


Primitive Observation Type Editor

Phenomenon: PKU

Validation Policy: a DiscreteValidator 'LabResult'

a) Primitive Observation Type Editor


Composite Observation Type Editor

Phenomenon: TESTRESULT

Validation Policy

Available...	Included...
AGE-RangedValidator	GAL-DefaultValidator
FEEDING-DiscreteValidator	PKU-DiscreteValidator
MothersAge-RangedValidator	TSH-DiscreteValidator
	WEIGHT-RangedValidator

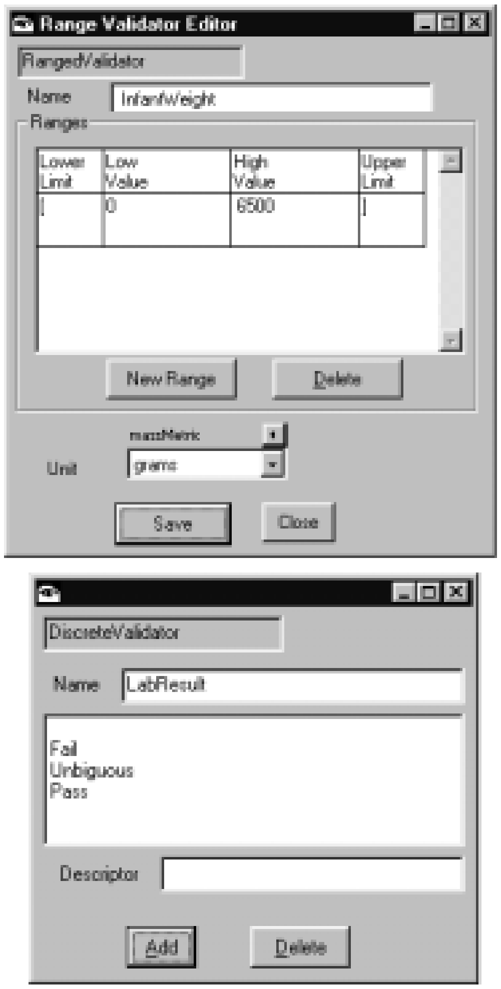
b) Composite Observation Type Editor

Figure 10 ObservationType Editors

The first version of these editors required the analyst/developer to understand many of the business rules of how to build the observations, as the early versions of the editor did not include much error checking. This is a common during software development; how much energy is put into developing the application and how much is put into developing support tools. The answer is never very clear and usually depends upon the amount of resources available, the amount of applications that are being maintained, and the pain caused to those using the editors.

Another example of misusing the model was the result of a new requirement produced after a user review. Users required that some observations could have more than one value; examples of this requirement are observations such as Ethnicity, Language, and Race. The developers decided to create new instances of CompositeObservationType with phenomenonType of CompLanguage, CompEthnicity, and CompRace. Each new ObservationType has a collection of the appropriate primitive type, e.g. Language, Ethnicity and Race. Unfortunately, whenever a new language needed to be added to the system, the CompositeObservationType needed to be edited. Similarly, with CompositeObservationTypes, you needed to select a value such as "NO" for a SpanishLanguage Observation to designate that this was not one of the composites you wanted to select. Just because you don't select an observation, it doesn't imply that you are saying anything about it. However, by modeling values such as Language by creating many different CompositeObservationTypes, certain undesired meaning was added to the system.

We solved this problem by changing the cardinality of the association between Observation and its possible values and making the ObservationType responsible for knowing whether it allows multiple values. Changing our documentation only took a few minutes, changing the Smalltalk code took a few hours of refactoring; but updating the database took several days since we had to change the mapping schema for those classes, update table definitions and, finally, migrate all the affected data.


Range Validator Editor

RangedValidator

Name: InfantWeight

Ranges

Lower Limit	Low Value	High Value	Upper Limit
[	0	6500	]

Unit: massMetric, grams

DiscreteValidator

Name: LabResult

Fail
Unbiguous
Pass

Descriptor

Figure 11 - Validator Editors

4. IMPLEMENTATION ISSUES

The primary implementation issues that need to be addressed when developing Adaptive Object-Models 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 relational database. 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. [10, 22, 24] describe some standard ways for mapping objects to relational databases. We developed our mapping object standards using a combination of these techniques.

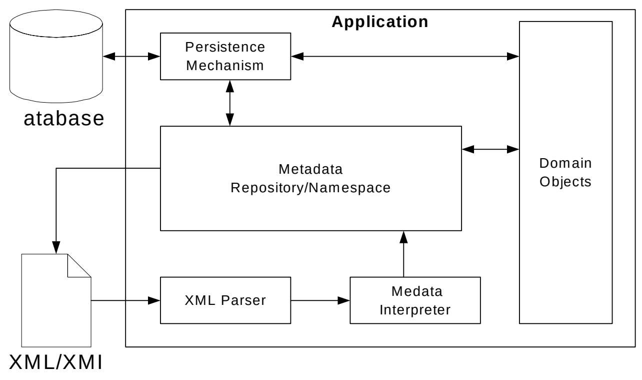


Figure 12 - Storing and Retrieving Metadata

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 12).

GUI issues also needed to be considered when implementing this dynamic architecture. The model we presented as part of the example make it easy to create new observation domain objects but the values still need to be entered from and presented to final users. It was impractical and resource demanding to develop dedicated widgets for each kind of observation. Our solution was to develop a set of widgets that were able to get information from the metamodel and customize themselves based upon the metadata. We found out that we only needed half a dozen different types of widgets that only differed based upon the validation business rules described by the ObservationTypes.

For example, PrimitiveObservationTypes are presenting either a ranged value, or a discrete set of values, which might be presented in a list or as a textual string. Therefore these types of GUIs can be developed and associated with the types. The only differences are the types of values they are either displaying or validating.

An example of this can be seen in the Refugee application developed by IDPH for screening refugees as they are accepted into the country (see Figure 13). This application captures over 100 observations about people as they enter the country. Almost every widget on the bottom half of Figure 13 is an observation. 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).

Introducing the Observation Model into the Refugee Tracking System was a delicate task. Developers found hard to understand the model from the documentation, moreover significant parts of the model is stored as metadata on the database. The result was a growing numbers of mistakes and duplications. For example, observations for measuring Height in different units (inches, centimeters, feet, etc) were created; this included a Quantity model, which is able to handle unit conversions. Part of our solution was to develop test cases in which developers can use running examples in order to understand how the model worked. After the initial period of learning and experimentation, developers of this system were able to successfully create the remaining ninety of one hundred observations in just a few 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 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 [19]. 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.

5. CONSEQUENCES OF AOMS

People are building systems that use metadata and Adaptive Object-Models from the ground up for a variety of reasons. ECOOP and 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 [16, 25, 26].

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 makes it much smaller in the number of classes. 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.

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.

The primary disadvantage is that these systems can be hard to build. 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 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. 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.

Adaptive Object-Models leads to a domain-specific language. Although it is often easier for users to understand a domain-specific language over 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. Other problems are those involved with training. There are ways around these problems but they are problems nonetheless and should be faced.

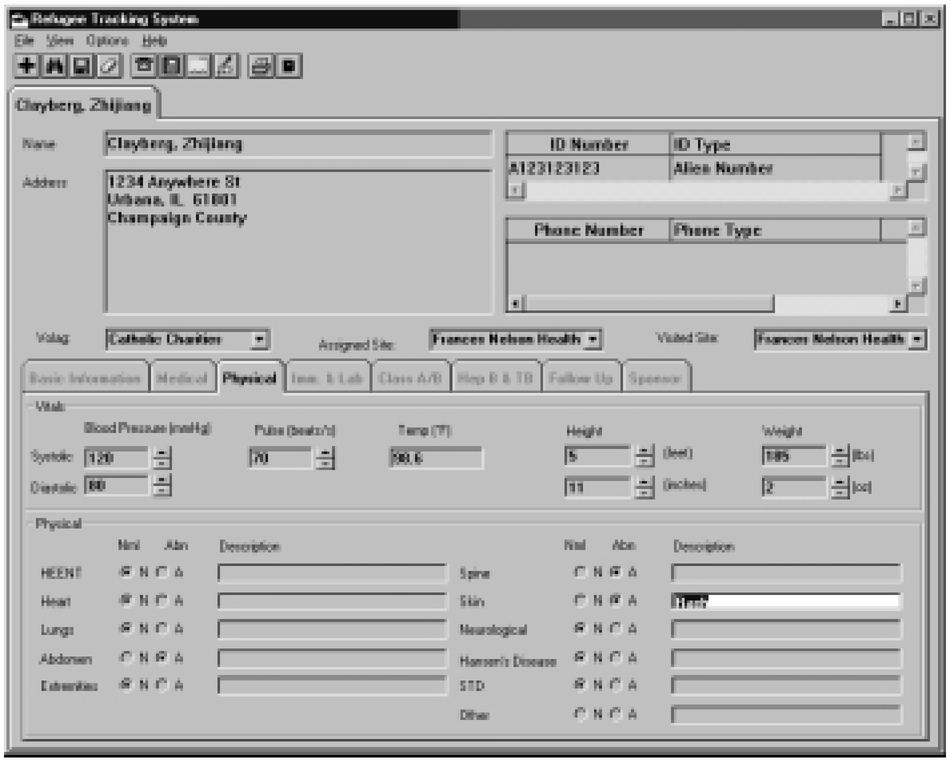
Adaptive Object-Models are not the solution to every problem. They require sustained commitment and investment, and their flexibility and power can make them harder to understand and maintain. They should only be undertaken when either the products or rules associated with products are rapidly changing,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

6. ALTERNATIVES 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 Code Generators, Generative Programming, Metamodeling, Table-Driven systems, and Business Rules research.

Code generators produce either executable-code or source-code. This technique 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 [3] 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 [21].



Name: Clayberg, Zhijiang

Address:
1234 Anywhere St
Urbana, IL 61801
Champaign County

ID Number	ID Type
A123123123	Alien Number

Phone Number	Phone Type

Volag: Catholic Charities
Assigned Site: Frances Nelson Health
Visited Site: Frances Nelson Health

Vitals

Blood Pressure (mmHg) Systolic: 120 Diastolic: 80
Pulse (beats/s): 70
Temp (°F): 98.6
Height: 5 (feet) 11 (inches)
Weight: 185 (lbs) 2 (oz)

Physical

	Nml	Abn	Description		Nml	Abn	Description
HEENT	[✓ N]	A		Spine	N	[✓ A]	
Heart	[✓ N]	A		Skin	N	[✓ A]	[illegible]
Lungs	[✓ N]	A		Neurological	[✓ N]	A	
Abdomen	N	[✓ A]		Hansen's Disease	[✓ N]	A	
Extremities	[✓ N]	A		STD	[✓ N]	A	
				Other	N	A	

Figure 13 - Refugee Tracking System

The central concept of Generative Programming is the generative domain model. In Generative Programming, the model has to cover a family of problems (not just one Solution or application). Furthermore, there has to be some high-level, domain-specific representation that allows you to specify/define a concrete instance of the family. Finally, there is some configuration knowledge that maps the problem description to the solution space.

The key point is that there are many possible technology mappings for a generative domain model. The domain-specific specification means could be implemented as a new textual or graphical language, or could be implemented right in a programming language (e.g., as enumeration types), or could be realized as some wizards or GUI forms. The mapping between the specification and the solution space could be done using code generation, dynamic reflection (including Adaptive Object-Models), or simply using the factory pattern - to name some alternatives.

In some cases (e.g., embedded systems), static configuration is sufficient, in which case code generation is appropriate. In other cases, dynamic configuration and reconfiguration is necessary, in which case dynamic metaprogramming and Adaptive Object-Models is the way to go. Ideally, one would like to encode the configuration knowledge in just one form and be able to decide for static or dynamic configuration independently of this form and based on context only. This closely corresponds to partial evaluation. Think of an Adaptive Object-Model and the possibility to encode an Adaptive Object-Model only once, but being able to either interpret it or to compile all or some of its dynamic flexibility away at any time.

Given this background, we would make the following comparison between Adaptive Object-Models and Generative Programming.

- Generative Programming focuses on system families; Adaptive Object-Models can be used for system families (e.g. to code frameworks), but not only.
- Adaptive Object-Model focuses on objects (as the name already suggests); Generative Programming is independent of OO.
- Adaptive Object-Model can be used as one possible technology mapping to do Generative Programming.

Metamodeling techniques 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 [15]. 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 [24] 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. Riehle et. al [18] describes a UML Virtual Machine that has an Adaptive Object-Model 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 [14]. 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 has been a couple of workshops sponsored at OOPSLA recently, which focused on just this topic where many papers were presented describing both working systems and research in this area.

7. CONCLUSIONS

Adaptive Object-Models provide 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 *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. It is important to be prepared as the end users do more with your system (e.g. they try to model more cases) to add to your rule language and definitions. Expect the Adaptive Object-Model to grow to meet the descriptive power your users need, and keep actively involved in making sure their needs are covered and it doesn’t get out of hand (similar to what a framework author has to do).

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.

8. ACKNOWLEDGMENTS

We would like to thank the many people whose valuable input greatly improved this paper; specifically we would like to thank: Ali Arsanjani, 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] 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.

[2] 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.

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

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

[5] 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.

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

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

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

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

[10] Wolfgang Keller, Jens Coldewey. "Accessing Relational Databases". *Pattern Languages of Program Design 3*. Addisson Wesley, 1998.

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

[15] 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.

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

[17] 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.

[18] 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.

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

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

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

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. "Object-Oriented Modeling and Design" Prentice Hall, 1991.

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

[24] J. Yoder, R. Johnson, Q. Wilson. "Connecting Business Objects to Relational Databases". 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.

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

[26] 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.

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