RULES-BASED ARCHITECTURE FOR SYSTEMS DEVELOPMENT

Dr. Paul Dorsey, Dulcian, Inc.

What are the requirements for a computer system? How should we think about it? Without some way of organizing the detailed requirements, it is easy to get buried by millions of requirements.

The Zachman Framework (http://www.zifa.com/framework.html) can help analysts to ask all the right questions, but it does not necessarily help architects design the ultimate system.

Currently the only complete rule capturing mechanism is the system itself. All other tools and approaches (from use cases to rules products like IBM WebSphere-formerly ILOG- or Oracle Business Rules) can help organize the requirements or capture a small percentage of the business rules, but do not provide a mechanism for capturing all of the system rules.

I. What is a business rule?

This is a harder question to answer than you might think. The main problem is that there is no consistently accepted definition throughout the industry. There are many different groups, all using the term “business rules,” with different meanings in mind.

The origins of the term and the reasons for its popularity in the industry began with the “logical business rules” community founded by Ron Ross and Terry Moriarty using the name “The Business Rules Group.” Their perspective on business rules is quite similar to what a traditional systems person might call a “system requirement.” The term "business rules" does not imply specific system functionality, but rather the rules that the system needs to support. However, this is what good analysts have always meant by a system requirement.

The Business Rules Group popularized the term “business rules” and spurred thought about these rules, for which we owe them a debt of gratitude. We also owe them for trying to create the first formal taxonomies of business rules. The initial substantive contribution in this area was Ron Ross’ Business Rule Grammar which was published in *Business Rule Concepts* (1998, Business Rule Solutions).

For decades, the best systems analysts approached a project by first trying to understand what the system they were designing needed to accomplish. Only after this solid foundation of the “business rules” of the organization was collected and codified, did the analyst, working with users, come up with the actual system-level requirements. From the perspective of this first group, “a business rules approach” is simply a renaming of the process that good analysts have been using for years.

The term “business rules” in most products ("business rule tools") usually refers to a specific subset of rules. For example, almost all products assume that the database already exists, so rules that can be implemented in the database are usually not considered in rules products. The common rules that most tools to try to support are process flow rules. Others may try to support data validation rules. Finally, a third type attempts to develop a nice grammar or repository, and whatever rules are easily supported by that grammar are defined as the “business rules.”

In this paper, the term “business rule” encompasses the entire system in order to cast all functional requirements as rules. What distinguishes a “business rule” from the traditional definition of a “requirement” is that the business rule will be a formal representation of the requirement. The rule must be precise enough that it can act as part of the system, or be used to directly generate part of the system.

II. There are LOTS of rules

High-level business rules or "analysis rules" serve as a common communication point between users and IT professionals with little expectation that these rules can be used to generate code.

Ron Ross' approach using RuleSpeak® for a project would gather about 1,000 of these high-level rules to complete the analysis phase. Other business rule practitioners might gather many more detailed rules (10,000-20,000). In neither case are there enough rules for full system specification.

An enterprise-wide system requires hundreds of thousands, if not millions of rules. If this seems like an exaggerated number, consider that a typical enterprise system contains approximately 20,000 unique data elements (attributes). If each system object moves through an average of 10 states, the organization has at least 10 roles, and then (for each data element in each state) it is necessary to keep track of which role(s) is allowed to edit an attribute, then 2,000,000 questions would need to be asked for this single type of data access rule.

It is possible to make a series of assumptions to decrease the complexity of rule gathering. One can assume that privileges assigned at the class level can be applied to all attributes in all states. At this point, it is only necessary to override those assumptions with your rules; however, no matter how you describe your system, there are LOTS of rules.

Rules can be applied against collections of objects such as "A Senior HR Manager can edit anything in the HR area," but this presupposes that the HR area has been clearly defined. Printing out a rule repository for a complex system requires hundreds or thousands of pages, representing hundreds of thousands or millions of rules.

III. Existing Business Rules Approaches

The database community has been working towards what has currently evolved into the "business rules approach." Oracle database professionals use the term "business rules" assuming that the database already exists and the business rules that should be enforced are the data rules that cannot be easily represented within a database or other repository.

Current business rule products fall into two groups:

1. Analysis rule products (Ex: Ron Ross's RuleSpeak®): These products capture logical rules in a natural language-like grammar. They work well for capturing high-level requirements, but would collapse under the volume of rules if they were used for complete system specification.
2. Rules subset products (Ex. IBM Websphere ILOG JRules (formerly ILOG), Pegasystems' PegaRules, etc.): These products take a specific subset of the rules of the system and capture them in a repository or grammar. The problem with this product type is that these tools provide yet another place to hide business rules.

The Business Rules Group & RuleSpeak®

Ronald Ross' grammar attempted to formalize the English-language representation of a business rule such as "Every purchase order over $50,000 must go through three levels of approval" and precisely represent it in a machine readable fashion. Why didn't this grammar catch on? First, the basics of the grammar are very complex, making it difficult to use this grammar effectively. Even those very familiar with the grammar were sometimes unable to represent many business rules using the Ron Ross system. If this grammar had evolved and been in widespread use, the entire idea of a grammar for the representation of system rules eventually fails, whether it is language or diagram-based. There are just too many rules needed for most systems.

The other contribution of The Business Rules Group is represented by its published papers, which have attempted to create a structural taxonomy of business rules where rules are typed and, at some level, parsed. Again, in a full development environment, the existing taxonomy is not directly usable but does represent some excellent thinking and helped focus attention on the issue of structuring and parsing business rules.

In summary, the contributions of the Business Rules Group were that its members popularized the term and concept of business rules, proved that it was possible to build a precise rule grammar, and provided the intellectual foundation for structuring and parsing the business rules of an organization.

IV. Using a Framework

Frameworks help us to understand the world, but once they become entrenched, they can also limit the way we think:

1. Relational data models provide a great way to model a relational database but limit our thinking in terms of data structures that translate into a relational database.
2. The Unified Modeling Language (UML) helped to build Object-Oriented (OO) systems, but also framed our thinking in terms of OO programming.
3. The Model-View-Controller framework helped categorize parts of our user interface architecture, but limited us from seeing a broader total system perspective.

Goals of the proposed framework

There are several goals for the proposed framework:

1. **Rules Based:** The framework must be the foundation for a rules-based grammar or repository to support a precise specification of the system.
2. **Comprehensive:** The framework should allow developers and architects to completely describe the system. If used to generate the system, the framework must be able to generate everything from the data structures to the end user screens. It must also support the generation of web services or other between-system interactions.
3. **User-Centric, not implementation technology-specific:** The framework must support system specification from the users' perspective, not in support of any particular technology or approach. The framework should not imply any particular technology (database, OO, etc.) but should be implementation technology-agnostic.
4. **Manageable:** It does no good for a framework to be complete if it results in an unmanageable set of rules. The framework must scale to easily manage a million or more rules. Rules must be easy to find and maintain. Rule specification must be convenient and efficient.

Proposed Solution

The following outline describes the proposed framework:

I. Object
- A. Structure
- B. Process
  - 1. State
  - 2. State Event
  - 3. Rules
- C. Data validation

II. User Interface
- A. Model
- B. View
  - 1. Layout
  - 2. Event/Action
- C. Controller

III. Object Interaction
- A. Object Mapping

Rule Types

This framework includes three classes of rules:

A. Object rules concern the things that the system is being built to support.
B. User interface rules include screen and report rules that are not inherent in the definition of the object
C. Object interaction rules support internal and external interfaces between sets of objects.

Each will be briefly discussed.

A. OBJECT RULES

Object rules involve what the system is designed to support. Anything about an object that is not UI-dependent is an object-level rule. This includes the following:

1) Structural - Rules normally stored in a data model, such as "An employee has a mandatory last name that is up to 30 characters in length."
2) Process related – Example: The approval process for a purchase order.
3) Data validation – Example: A person with “single” marital status must not have an associated relative of type “spouse.”

Object-level rules are where most of the design of the system takes place. This idea should be the core of our thinking about application development. Traditionally business rules about the objects are not even captured coherently, let alone coded so they are clearly attached to the objects.

Rules associated with objects always comprise a large portion of the total system rules. Many people think of object rules as only those that are contained in the data model. While it is true that object rules will define the data structure of the database, these rules can specify much more than what is contained in a traditional ERD. For example, the logical process flow for handling an object such a purchase order can be defined at the object level. The process should also be independent of any particular representation in the user interface. Data validation rules such as “any contract actions should take place between the start date and end date of the associated contract” should be articulated with the object so that any representation would conform to the rule.

B. USER INTERFACE (UI) RULES

User interface rules are those that govern screen layout and behavior that has not already been specified at the object level. For example, at the object level, a process flow would have already been defined but at the UI level, the developer needs to decide if the process flow alternatives at each point should be shown as buttons, a drop down list of values, or in a child screen pops up when a user clicks the “Process” button.

If rules are placed at the object level, why are additional rules needed at the user interface level? It is possible to generate an entire default user interface based solely upon knowledge of the associated objects. However, what is generated will not be particularly user-friendly. There are additional rules that are concerned more with the needs of the user that are not properly associated with system objects at all. For example, determining which attributes will be displayed in what positions on the screen, what areas of the screen are automatically populated when the screen opens, whether or not rules are validated when tabbing out of a field, are rules that are not dependent upon system objects, but instead are UI design decisions.

C. Object Interaction (Mapping)

Object interaction is defined as rules associated with data migration (ETL), web services, and similar situations where one set of objects must be mapped to a second set. The association need not even be to an actual system. Object interaction rules can also define how the various systems map to the Virtual Operational Data Store (VODS).

This is one of the most versatile ideas described in this paper. The same structure can be used to generate code for many different purposes and architectures.

V. Detailed Description of Framework Elements

This section describes each sub-element of the framework.

Object-Structure

Structural rules describe the static nature of the objects in the system. What are the objects of interest? What are the attributes associated with them? This is the kind of information that is traditionally captured in ERDs, although in recent years, UML class diagrams have become more widely used for diagramming systems.

UML Class diagrams (with some extensions) are somewhat better suited for structural modeling than ERDs. ERD proponents argue that UML is an implementation tool and that cannot be used for logical modeling. UML proponents argue that ERDs are too limited and do not support inheritance well enough. This debate is beyond the scope of this paper, but the choice of diagramming syntax is of far less importance than deciding what should be captured.

Data modeling has traditionally only been concerned with capturing the information required to generate the tables, columns, and constraints for the DBMS. Structural rules include a much broader range of information than that limited scope. Structural rules should encompass the following areas not traditionally included in data modeling:

1) Derived attributes
2) Objects defined using other objects (for example, to generate database views)
3) Audit, security, and history requirements
4) Default object display
5) Triggering events based on Insert, Update, and Deletion of objects or attribute values

The idea is to partition all of the rules required to specify a system, and not simply identify the rules necessary to generate the database. Focusing on database generation rather than on the structure of the objects is a primary reason for the limitation of CASE tools such as Oracle Designer. Rather than thinking about capturing rules of a specific type (i.e. structural), CASE tools focused on the purpose (i.e. generating a database), hence the tools never evolved into full system specification tools.

For the structural rules area of the repository, imagine a standard UML class diagram meta-model as shown in Figure 1. Of course the real model would be more complex, but for the purposes of this paper, showing the key classes is sufficient.

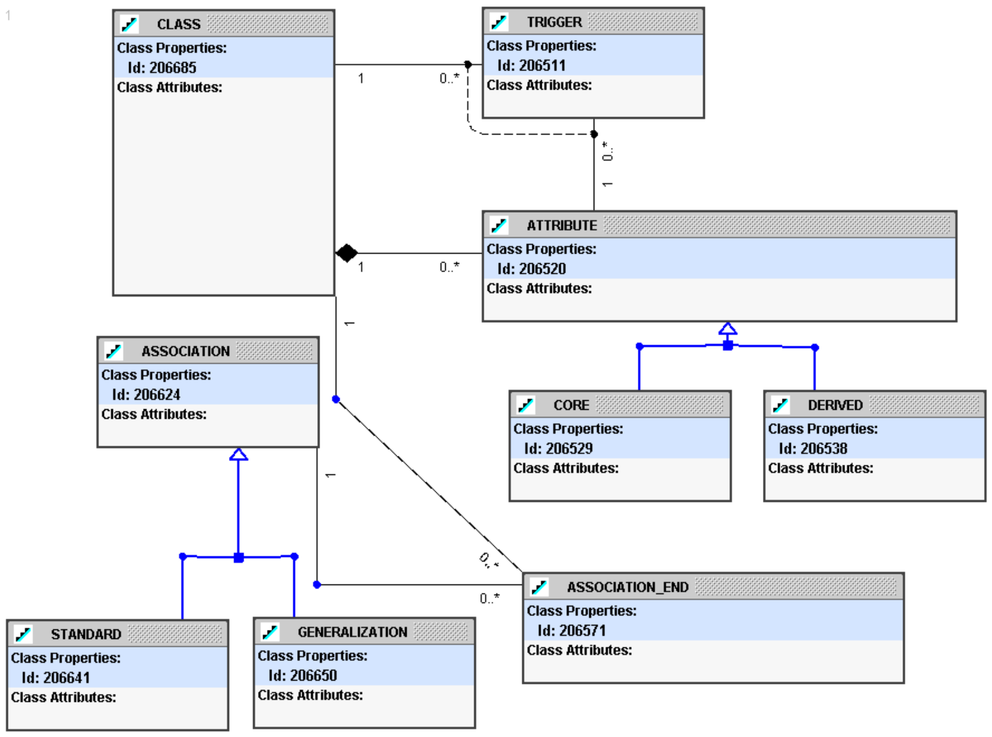

Figure 1: UML Class Diagram Meta-Model

NOTE: The only part of the model that may seem odd is having “ASSOCIATION_END” as its own class. This allows generalizations to have one head end class and multiple end classes.

Object-Process

Process rules define the process flow of an object from one state to another. A particular object may have states associated with different aspects of the object. For example, a person may simultaneously have a physical health as well as an economic health. Independent process flows can be defined for each aspect.

Process rules should be described using some type of flowchart or process flow rather than a declarative mechanism. There are simply too many rules associated with an object to manage without dividing the object flow into states. If process rules are specified using a declarative mechanism, it is possible to end up with thousands (or even tens of thousands) of rules. These rules interact, may cancel each other out, and are impossible to manage. Traditional State Transition Engine (STE) diagrams or flowcharts are similarly hard to manage.

University classes in programming often assigned students to create flowcharts to document programming assignments. The code was written first and the flowchart afterward. The flowchart for a simple 200-line program had 60-80 little boxes on it and the flowchart was harder to read than the program code itself.

The solution to making an STE work is to embed some logic in the state itself. This allows the number of states to remain manageable. Perhaps the best way to do this is to view the state from the users' perspective. Think of a state as being a desk in an office. It has an inbox and an outbox. When the object arrives at the inbox state, various actions can be taken. If the object sits in any one state for too long, it can automatically be routed to another state. Having the ability to create a state that is a much richer object drastically reduces the required number of states used to describe a process. Rather than requiring hundreds of states to describe a process, even a very complex process can be described using a few dozen states.
The formal structure used is to add the idea of an "event" on a state (similar to the idea of a database trigger on a table). When the event is triggered, actions can be executed. Rules can be attached to an event to prevent it from occurring (for example security to prevent an object from being opened).

States are associated with a class of object. Process flows are not independent things. The process flow defines the allowable states for a particular class of objects. The high-level process flow repository model is shown in Figure 2.

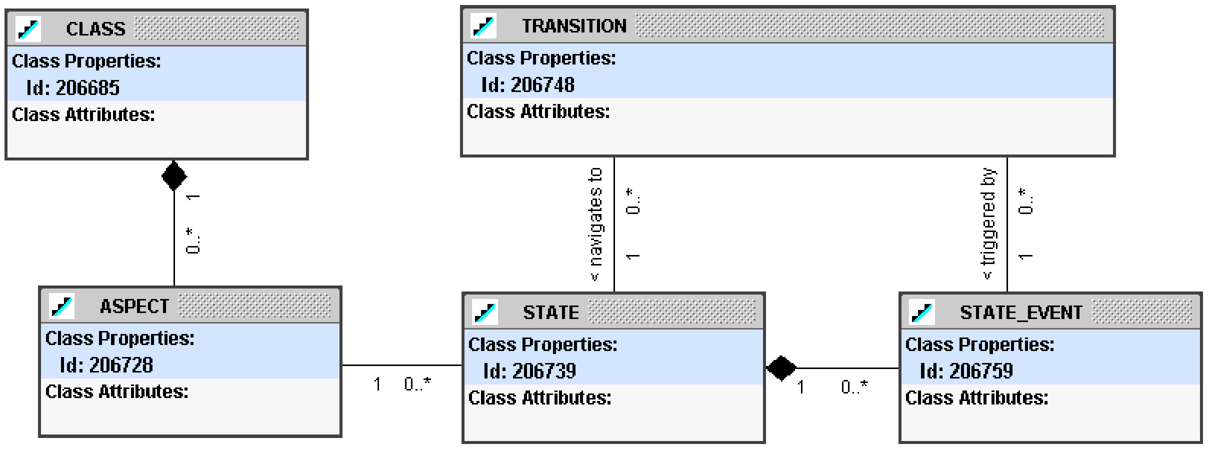

Figure 2: Process Flow Repository

Object-Data Validation

Data validation is a complex topic in its own right. A system may easily contain hundreds of data validation rules. These rules may always need to be enforced or only contingently enforced based upon some condition or state of the object. The rules may only require looking at the object being validated or accessing objects in other classes. Rule failure may trigger a user warning or may prevent data modification entirely.

The difficulty is in creating a grammar to help specify the rules. Natural language is not precise enough and code is too hard to read. The solution is to place the rules at the object level, but support an Object Constraint Language (OCL)-like syntax that allows validation across classes. For example, the rule to say that a department must have at least one employee (in the standard EMP/DEPT 1-to-many model) would be written as:

```
:_child.emp.count >= 1
```

This grammar can be easily extended to support 99% of all rules encountered.
Validation rules are often only contingently required. Therefore, these rules can be invoked at the object state level and may be contingently executed based upon some condition. The high-level model to support the data validation repository is shown in Figure 3.

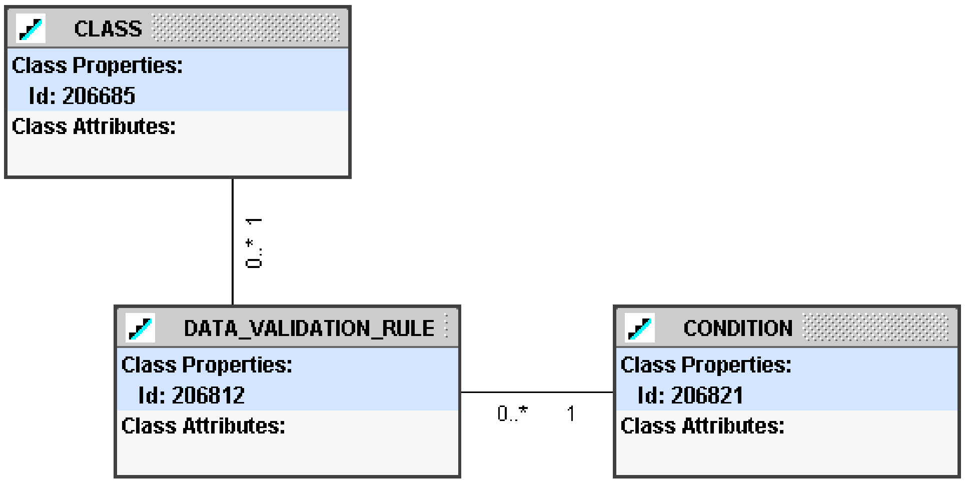

Figure 3: Data Validation Repository Model

User Interface

Once the object rules are collected, there are some additional rules required to specify the UI. For the purposes of this discussion, the UI rules will be kept to a minimum so that rules described at the object level will not be repeated at the UI level.

The model-view-controller (MVC) pattern architecture itself will not be discussed here, nor will the decision to use it be defended. However, the author’s perspective on MVC may be slightly different from the industry standard in that it is viewed as a logical design pattern, and not simply a way to write code. The goal is to define the application independent of any technology or implementation considerations.

1. Model Layer

The model portion of the logical UI rules is not difficult to specify. There is already so much information at the object level that little extra information is required at the UI level. Classes, attributes, and associations have already been defined at the object level. As a result, the only requirement at the UI level is to select a subset of objects (classes, attributes, associations) from the object level for use in the UI specification. This approach runs counter to the way in which most systems are built.

Most tools specializing in model development support very sophisticated object specification in the model portion of the UI. This approach does not preclude a “thick” UI model level for implementation; it merely implies that the structure of the UI model should properly be defined at the object-level.

Using this approach means that the structural rules at the object level will be quite sophisticated, requiring not only standard views, but also views that are dynamically altered or generated based on the values of some passed parameters. All that remains for the UI model specification is to point to existing structural object specifications. The model to support this is shown in Figure 4.

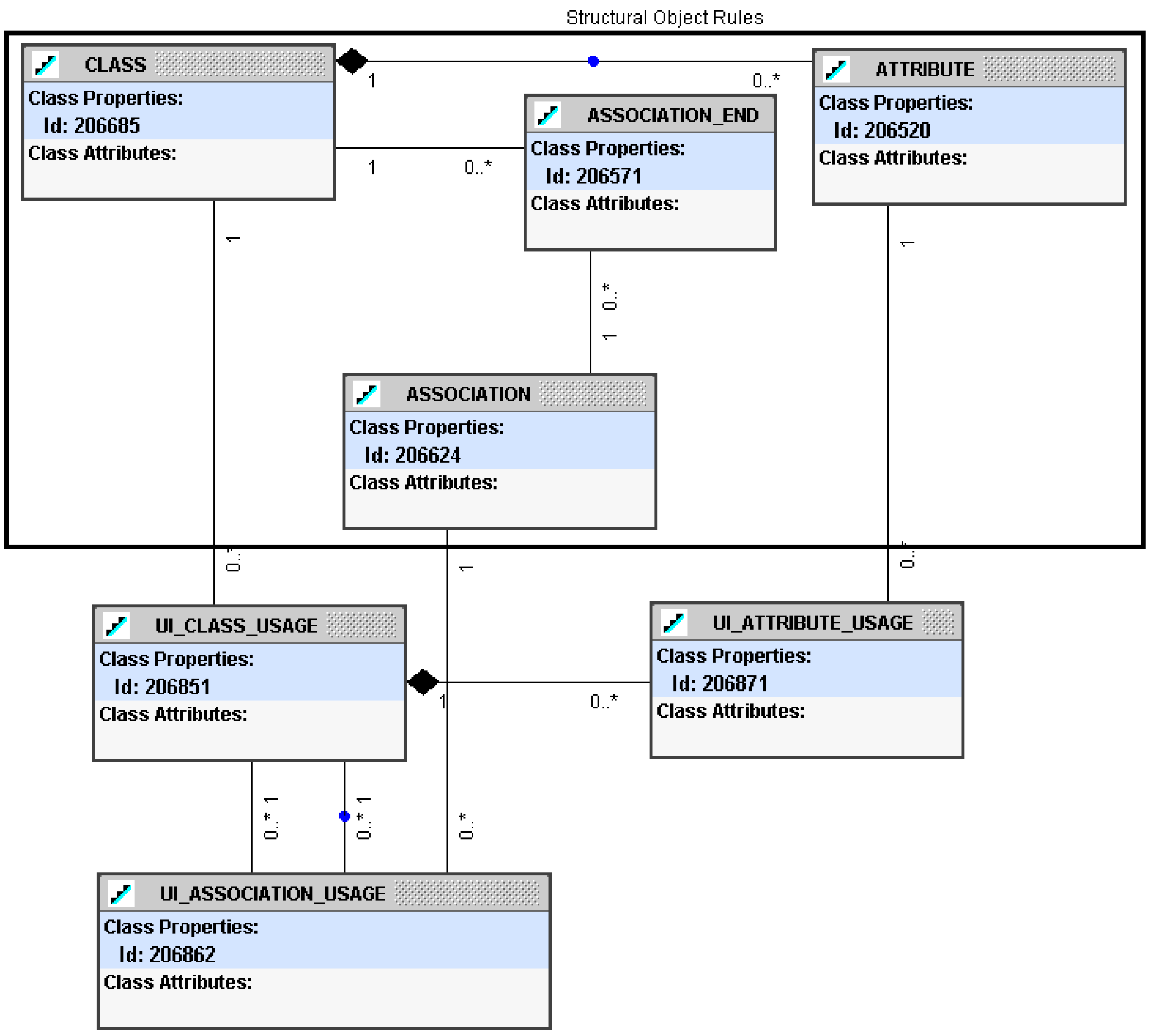

Figure 4: UI Model Specification

2. View Layer

The rules in the view layer of the logical UI are themselves divided into structural (elements and how are they grouped), logical (what happens when a screen opens, or a button is pressed), and presentation (how and where the elements are displayed).

The view layer structural rules are very simple. They define the UI elements (fields, buttons, etc.) and how they are grouped and bound to the UI model.

On the other hand, the view layer logical rules are quite complex. A full Event-Condition-Action (ECA) architecture is needed to define what happens when events (button press, open an application, etc.) occur. Conditions, actions, and events are defined as reusable objects. The model to support this ECA structure is shown in Figure 5.

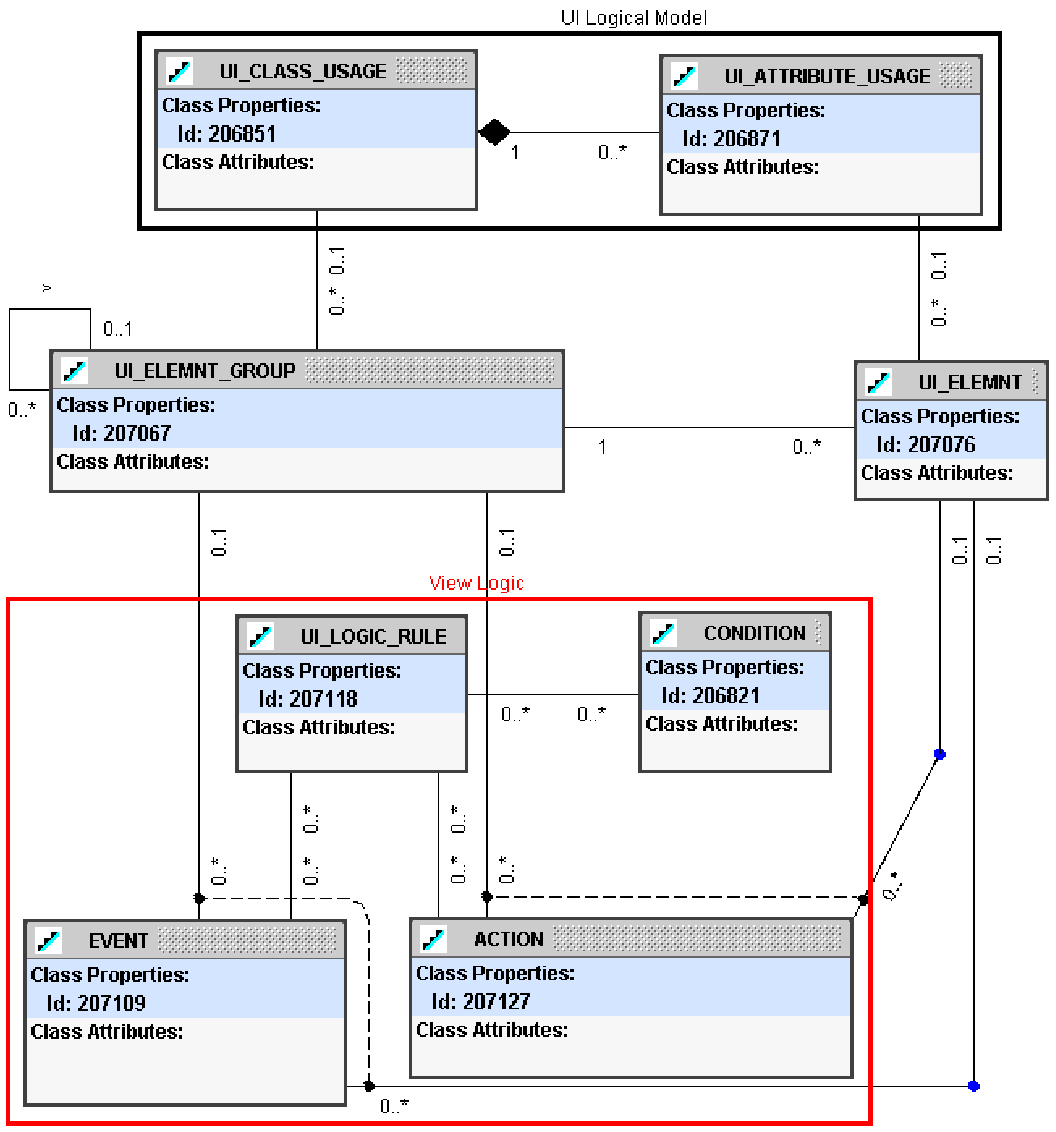

Figure 5: ECA Architecture model

Presentation rules are simply attributes of UI_ELEMNT_GROUP and UI_ELEMNT. The tricky part here is that different products use very different methods to describe the layout presentation. The Oracle Developer tool uses X and Y position, whereas HTML uses tags that automatically position themselves on the screen. Trying to come up with a technology-independent way to describe layout is not a problem that is currently well handled in any tool.

3. Controller Layer

The real strength of the Model-View-Controller (MVC) design pattern is the Controller layer. This layer can be partitioned into rules controlling screen navigation and what happens during screen transitions. The Struts controller should not be used to logically define rules. It is too restrictive an implementation and would force a redesign if/when JavaServer Faces (JSF) become the standard. The Controller layer is a natural place for using the same type of State Transition Engine (STE) design employed for process flows at the object level. However, rather than the states representing object states, they will represent UI element groups.

By using a technology-independent way of describing page flow, it is possible to generate a system to Struts, JSFs, or any other platform.

Object Interaction Rules

Rules, thus far have been concerned with the objects in the system. There are many places where it is important to describe how one set of objects relates to a different set of objects. This is obviously important for data migration, but is also important for defining interfaces between systems. Object interaction rules can also be useful for defining how data objects relate to user interface objects.

Mapping is typically supported by taking one source object class at a time and describing how it can be transformed to a target class using a process flow metaphor. This is how Informatica and Oracle Warehouse Builder approach the problem.

The other way is to describe complex sets of information in a master-detail structure and then describe how to map that complex structure. This way of thinking does not require a process flow and usually results in code generation that runs more efficiently.

System interaction should be defined on the generated objects (but not necessarily database tables), and not on the logical objects. The mapping is actually done to elements in the repository so that if the element names change, the mapping code can be automatically regenerated.

The basic mapping model is shown in Figure 6.

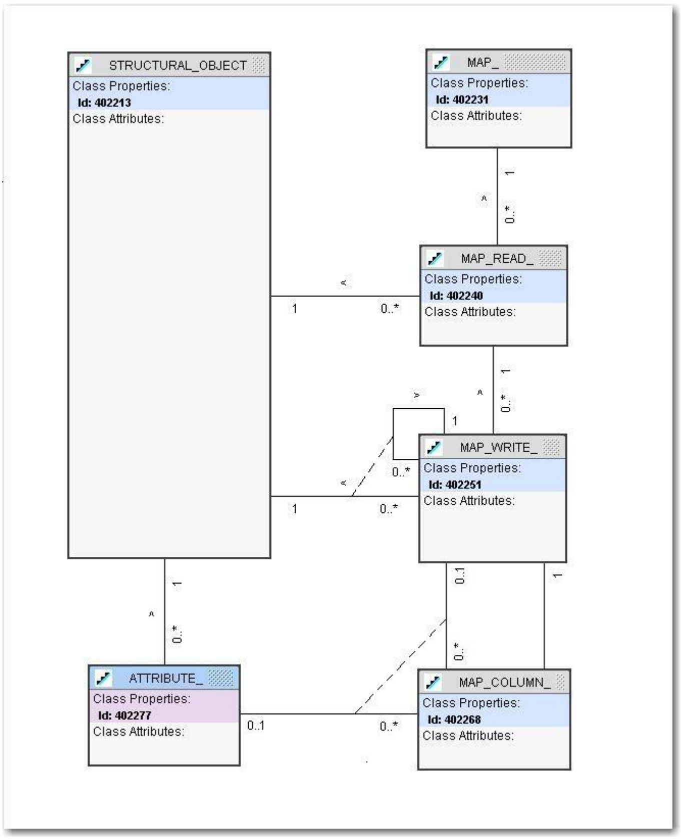

Figure 6: Mapping Model

VI. Conclusions

The success of this framework can be measured by the impact it has had on the thinking and architectural discussions of all of the technical staff at Dulcian. Deciding whether a rule is at the object or UI level is always the first part of these discussions.

This is the framework upon which Dulcian's BRIM® development environment is based. Each area of the framework has spawned a tool in the BRIM® architecture. Even in non-BRIM® projects, the framework described here has had a huge impact on our thinking. ADF projects typically use a thick database approach, implementing all object rules in the database and a very thin ADF BC model layer, just as the framework would suggest.

This framework has several important advantages:

1) Separation of object rules and UI rules means thinking about database objects independent of their UI presentation, which encourages object encapsulation.
2) The addition of the “state event” to a process flow state changes process flows from unmanageable flowcharts with thousands of states to easy to understand flows with no more than 100-200 states.
3) A separate object interaction part of the framework is used for everything from ETL to web services.
4) Recognition of data validation rules (especially validations that span records in multiple tables) as an independent framework component reduces the complexity of this long-standing challenge to building successful systems.

The framework has remained stable with no changes for over 3 years. It continues to help us build better systems in any technical architecture. I hope you will find it useful.

About the Author

Dr. Paul Dorsey is the founder and president of Dulcian, Inc. an Oracle consulting firm specializing in business rules and web based application development. He is the chief architect of Dulcian's Business Rules Information Manager (BRIM®) tool. Paul is the co-author of seven Oracle Press books on Designer, Database Design, Developer, and JDeveloper, which have been translated into nine languages as well as the Wiley Press book *PL/SQL for Dummies*. Paul is an Oracle ACE Director. He is President Emeritus of NYOUG and the Associate Editor of the International Oracle User Group’s SELECT Journal. In 2003, Dr. Dorsey was honored by ODTUG as volunteer of the year, in 2001 by IOUG as volunteer of the year and by Oracle as one of the six initial honorary Oracle 9*i* Certified Masters. Paul is also the founder and Chairperson of the ODTUG Symposium, currently in its eleventh year. Dr. Dorsey's submission of a Survey Generator built to collect data for The Preeclampsia Foundation was the winner of the 2007 Oracle Fusion Middleware Developer Challenge and Oracle selected him as the 2007 PL/SQL Developer of the Year. Paul can be contacted at paul_dorsey@dulcian.com