The User-Defined Product Framework

by Ralph Johnson and Jeff Oakes
johnson@cs.uiuc.edu, joakes@itthartford.com

Introduction

This paper describes a generic framework for "attributed composite objects" that is called the User-Defined Product (UDP) framework. 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 purpose of the UDP framework is to let users 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 UDP framework does not solve all the problems of business object. For example, it does not automate workflow or accounting, though it must cooperate with them. Typical workflow problems are that a policy might have to be approved before it becomes official, and a new bicycle model might have the lifecycle "proposed", "accepted", "active", and "obsolete". Once a model becomes obsolete, you can't accept any backorders, but can only sell the inventory. Typical accounting problems are that some transactions involving a policy (like selling it, getting payments on, or paying off on the insurance) also involve either financial obligations or the transfer of money, and must be accounted for. The framework does not handle either of these problems, though you can write code that handles them. Thus, it only solves some of the problems of business objects.

The UDP framework was developed at The Hartford, where it was used to represent insurance policies. We believe it is much more general than insurance, so we also use bicycle examples, but only the insurance examples come from experience.

This paper describes the UDP framework as a sequence of patterns. Each pattern will progressively reveal the framework, making it more flexible and also more complicated. Sometimes the features added by one pattern are removed by another, so the framework

does not grow steadily, but sometimes seems to regress. The only complete view of the framework is the one at the end of the paper, when all the patterns have been revealed.

Pattern 1 - Composite/Interpreter

One way to define a complex composite object such as an insurance policy would be to multiply inherit from classes that make up its behavior, such as PropertyPolicy, AutoPolicy, and FloodRider. Unfortunately, this leads to a complex class hierarchy that is always changing. ITT Hartford estimated that it would take 10,000 classes to represent all the combinations that they used. Moreover, covering two cars might require inheriting twice from AutoPolicy, which few languages support.

Which is the best way to combine features, multiple inheritance or composition?

The answer is easy for Java and Smalltalk programmers, since their language doesn't support multiple inheritance. But even if a language does support it, multiple inheritance works best in simple cases. If there are a hundred possible components and tens of thousands of likely combinations, it is too complicated to specify the combinations in advance using multiple inheritance. Moreover, many combinations will need a special way to combine the features of the components. This is why CLOS has method combination rules. Unlike CLOS, C++ does not have method combination rules nor dynamic creation of classes. Thus, multiple inheritance should be used in C++ only for fairly simple cases.

Use object composition to combine features instead of multiple inheritance.

A policy should contain a set of components, some of which are PropertyComponents and some of which are AutoComponents. If there is flood insurance on a house then the PropertyComponent representing the house will contain a FloodComponent. This makes it easier to add a new kind of policy and to mix and match policies.

There will be a class hierarchy of PolicyComponents, and for each new policy feature there will be a subclass of PolicyComponent. A PolicyComponent like AutoComponent that contains other PolicyComponents will be a CompositePolicyComponent, following the Composite pattern. So, we will just call the classes Component and Composite.

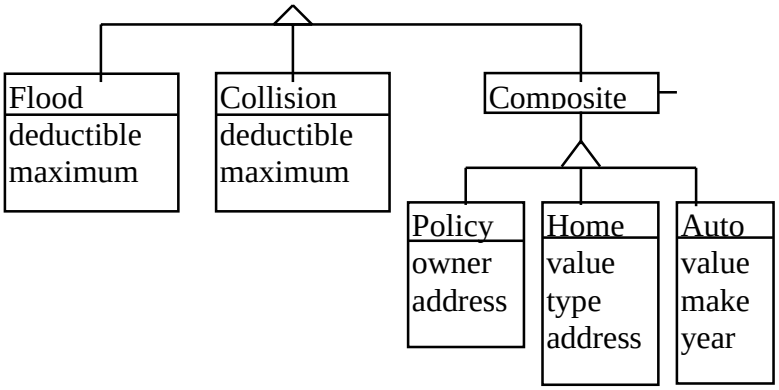

Figure 1: Component Class Hierarchy

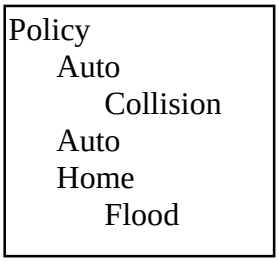

Figure 2: A Composite Object

Figure 1 shows a small part of a typical Component class hierarchy. Flood and Collision are leaf classes, while Policy, Home, and Auto are composites. An insurance policy would be represented by a composite object as shown in Figure 2, where the root is the composite Policy object and the interior nodes are the composites Auto and Home. This policy insures one house and two cars. The first car has collision insurance, but the second does not.

Each Component has attributes. The Policy component knows the owner of the policy and his address, while the Home component knows the value of the home and its address.

All Components must have the same interface. For an insurance system, this implies being able to compute the value of the policy, to print it, and to display fields for data entry. For a bicycle system, it implies being able to compute its cost and its weight for shipping. In general, a component implements part of this interface by using that same operation on its components. So, the value of an insurance component is some function of the values of its components, as is the value of a bicycle.

This can be looked at as an example of the Interpreter pattern. The Interpreter pattern has a class hierarchy (the Component hierarchy) that models a language, trees made from these classes that model programs in that language (the tree of Figure 2), and an interpreter implemented by distributing a method across the class hierarchy (evaluate the policy, or print it), and executed by traversing the tree of instances. It is not an ideal example, because most people do not naturally think of a description of an insurance policy or a bicycle as a program (though Lieberherr would think it quite natural [Lieberherr96]). Moreover, part of the Interpreter pattern is a “context”, which is an object that is an argument to the “interpret” method that is created at the start of interpretation and that participates in every aspect of it. The current design does not have a context. Nevertheless, we will see that the Interpreter pattern is a powerful metaphor that will be important later, and that the context will also turn out to be important.

The design is still complex and hard to use, because there will be a huge number of Component classes, and adding a feature means making a new one. But the next patterns will take care of that.

Pattern 2 - Variable State

Component has too many subclasses. How can we keep from having to subclass Component?

One reason to make a subclass of Component is to define instance variables to represent attributes. A Component's attributes are strings, numbers, dates, and other simple values. In contrast, its components will be other objects like itself. A program that edits a Component will do little more than display and modify its attributes.

The Variable State pattern[Beck97] represents the attributes of an object as a collection, rather than as its instance variables. Component will have a dictionary called "attributes" that maps the name of the attribute to its value. In this case, it is best to have a special Attribute class that not only holds the value of the attribute, but also its type. This way a Component editor can tell whether an attribute is supposed to be a number or a date, and can produce the appropriate user interface. This is similar to Fowler's Measurement pattern [Fowler97].

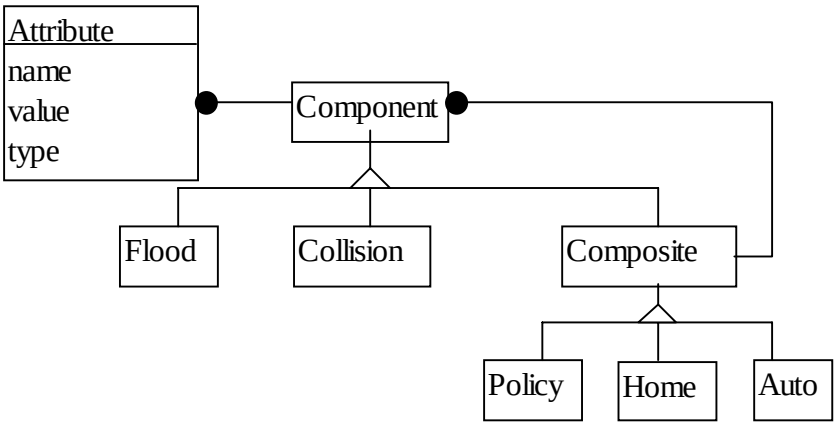

Figure 3: Design with Variable State

Eliminate the need to subclass to add instance variables by storing attributes in a dictionary instead of directly in an instance variable.

Unfortunately, the Variable State pattern alone won't eliminate many subclasses, because different Components have different behaviors.

Pattern 3 - Strategy

Component has too many subclasses. How can we keep from having to subclass Component?

A PropertyPolicy not only has different attributes than an AutoPolicy, it has a different algorithm to compute its value. The most obvious way to describe this difference is by making a subclass. However, another way to give different objects different algorithms is to use the Strategy pattern[Gamma95]. The Strategy pattern turns algorithms into objects and lets the algorithm used by an object be independent of the object's class. If we model the different algorithms with the Strategy pattern then there is no reason to make subclasses of Component. Instead, different behaviors would be different subclasses of Strategy.

The Strategy pattern requires certain conditions. First, a set of algorithms can be turned into a Strategy only if they have the same interface. An object can have more than one Strategy, but all the objects using a Strategy must support its interface. If all the subclasses of Component are going to be eliminated, then they all must have the same interface. Moreover, it would be best if the Component interface were small, because the Strategy pattern will replace each algorithm in Component with an instance variable, and make a new class hierarchy for each algorithm.

Fortunately, insurance components have a fixed (and small) interface. An insurance component has four responsibilities; to compute its value (rating), to edit itself, to print itself out in a format that the consumer can read (issue), and to print itself out in a format for government regulators (coding). We can make a separate Strategy for each responsibility. Unfortunately, the behaviors of a bicycle component are different from the behaviors of an insurance component. However, a bicycle component will also have a fixed and fairly small interface, and we can apply the same patterns to it.

Make a Strategy for each method of Component that varies in its subclasses.

One of the results of using Strategy is that it is no longer necessary to have a separate Component and Composite class. Composite is the only subclass of Component. Moreover, applications will use Composite, not Component, because even if a particular component is a leaf, it is possible that it will be given components of its own some day, since business rules are constantly changing. The following figure shows how Composite and Component have been merged, and shows two of the four strategies. Each Strategy hierarchy will be much larger than what is shown. Some of the strategies will be specific to a particular kind of component, while others (like the VSum and ESum) will be generic strategies. For example, these two strategies are parameterized with a set of attribute names, and they compute the sum of the values of the attributes.

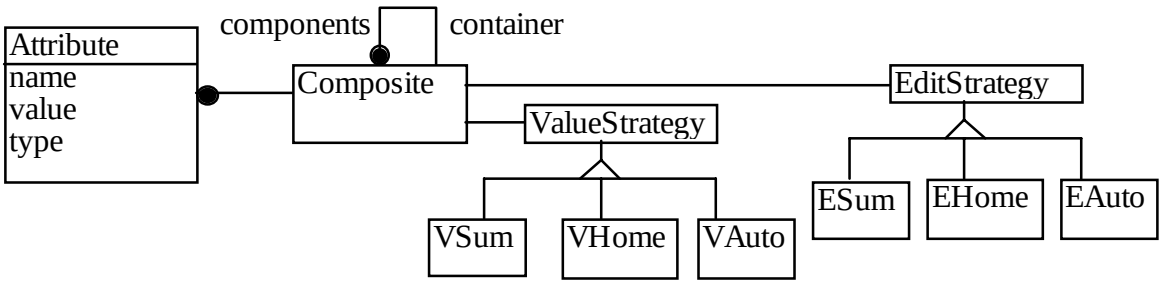

Figure 4: Design with Strategy

The Strategy pattern does not solve all our problems. In fact, it might not even reduce the number of classes. It replaces a constantly growing hierarchy of Components with a constantly growing hierarchy of strategies. Although the strategies are small and some of them are reusable, there will be a lot of them. We need a way to specify strategies with a fixed, and fairly small, set of classes.

Pattern 4 - Interpreter

Instead of making each strategy a monolithic class, we should compose them from smaller components. The way to learn what those components should be is to look carefully at the strategies.

The simplest strategies are those that compute the value of a policy. Each one returns a number that is a function of the values of the attributes of the Component and the values of its components. The function is sometimes an arithmetic expression, sometimes requires a table lookup (insurance rules vary by state, for example), and sometimes requires simple "if" statements. However, except for summing up the values of components, the functions never have to deal with iteration or recursion. Thus, they can be described by a fairly simple language, more at the level of spreadsheet rules than a real programming language.

How can we represent rules and functions in an object-oriented system?

The most obvious way to represent rules and functions is with a language, either the language used to build the framework or with a more specialized language. Using the language used to build the framework leads to a continuation of the Strategy pattern. Using a more specialized language makes it easier for non-programmers to customize the system, but requires implementing the language. There are lots of ways to implement a language; compile it to the underlying maching, define a simple virtual machine and compile to that, develop an interpreter. These implementation techniques trade off ease of implementation with speed of the final program.

In this case, speed of the final program is not that important. In general, interpreters are easier to implement than compilers. Moreover, object-oriented languages make one way of implementing an interpreter particularly easy. This is the Interpreter pattern, which

lets us represent a language as a class hierarchy, and a statement in that language as a tree of objects.

Implement rules and functions using the Interpreter pattern.

The Interpreter pattern has several parts[Gamma95]. First, there is a class hierarchy. In this case, it is the Rule hierarchy. This class hierarchy is used to provide components for a tree. Second, it supports an operation that evaluates the tree. In this case, the operation is valueUsing:. The argument to the operation is a context. In this case, the context contains all the attributes of the current component and its parents. This lets a collision component depend on the zip code of the driver of its car. <picture> Third, there is a way to build up "program" trees so that they can be interpreted later. This is done with a special purpose GUI.

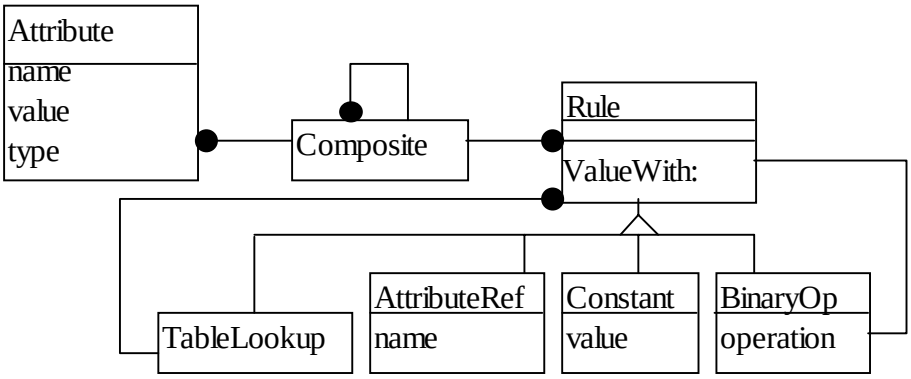

Figure 5: Using the Interpreter Pattern

The class hierarchy:

According to the Interpreter pattern, the Rule hierarchy corresponds to the grammar that is being interpreted. It has subclasses that look values up in tables, fetch the value of an attribute, represent constants, and perform arithmetic.

In theory, the Expression hierarchy does not change. Its classes are enough to describe any computation. In practice, a new application might need to make a new subclass because of a specialized need. For example, table lookup is not always needed, but is very common in insurance applications. If table lookup were not present, you could build an inefficient and awkward version from the arithmetic functions, but table lookup is important enough for insurance that it is worthwhile having a component that specialized for it.

<picture of "program">

Figure 6:

Figure 6 shows an example of how a set of objects can represent a Strategy. What we've done is to replace each Strategy object with a composite object that describes the same algorithm as the Strategy did, but does it with a fixed number of classes.

Contexts:

Each use of the Interpreter pattern has its own way of defining contexts. Sometimes the context is trivial, sometimes it is complex. One of the most common kinds of contexts is a name space, which is usually represented as a dictionary that maps names to objects, and that is what the UDP framework uses. In this case, the context maps names to attributes. If a rule needs to know the value of an attribute, it just looks it up in the context. Whenever a component is asked to evaluate one of its rules, it is given an initial context that contains all of the attributes of its ancestors. It will add its own attributes to that context before it uses it. But the original context shouldn't be changed, so it should be copied before attributes are added to it.

The standard Smalltalk dictionary is implemented as a hash table. If there are a lot of attributes then copying the context can take a lot of time. In that case, it is better to make a new context by building an object that points to both the old context and the new attributes. This context would have the interface of a dictionary, but be implemented differently. A context has as many segments as its component has ancestors, and if the tree of components were deep then it would be slow to access an attribute defined in the root of the tree. If the tree is deep, the roots attributes are accessed frequently, and each component has few attributes, then it is faster to represent contexts as standard Smalltalk dictionaries. But the odds are that contexts would be more efficient if they were represented as a sequence of dictionaries.

Evaluating the tree:

Rules can not only read attributes, they can write to attributes as well. Each rule consists of a set of formulas and tables whose values are computed and stored in the attributes. A rule can be divided into “pre-formulas” and “post-formulas”. A pre-formula is evaluated before a component’s children are evaluated, while the post-formula is evaluated afterwards. It is common for a pre-formula to initialize a total, for the children to accumulate a value in that total, and then for the post-formula to use the total.

This leads to very powerful rules. It is common for the root component (a Policy or a Bicycle) to initialize a lot of attributes, but for the non-root components to just update them. However, any component can compute a function of its descendents and use that as its own value. For example, a commercial policy may cover several buildings at a single location. However, there may be discounts and surcharges depending on the gross costs for each building. There are initial formulas which are evaluated for each building at a location. These values are inputs to the formula for the entire policy.

Building the tree:

The class definitions define a language and an interpreter for the language, but the actual programs are trees composed of instances of the Rule classes. So far we have not described how to build these trees.

Trees can be built either under direct program control or with a parser. Trees of Rule objects are really abstract syntax trees, so it is common to write a parser that constructs them, but it is not an essential part of the Interpreter pattern. The Interpreter pattern describes the classes of the nodes of the trees and how to interpret them, it doesn't prescribe how those trees were created.[Ralph, I'm unsure of your point here. And, with tables (trees) as well as formulas (Reverse Polish stacks) implementing rules, the direction is possibly confusing?]

The UDP framework uses a combination of techniques to build trees. There is a parser for arithmetic expressions, but special objects like tables have a special GUI for constructing them.
<Jeff, can you put a picture here showing one of the editors in action? Yes, but under separate cover to reduce size>

Rules are used for other purposes than just computing the values of attributes of a component. For example, they can be associated with an attribute and evaluated when it changes.

Pattern 5 - Type Object

The object model of Figure 1 had a subclass of Component for each kind of Component, but now all kinds of Components are implemented by a single, highly reusable, class. The differences between a collision rider on an automobile insurance policy and a derailure on a bicycle are expressed in the attributes and the strategies of a Component.

Although this design is flexible and reusable, it also can be inefficient and hard to understand. Part of the inefficiency comes from duplication. Most Components have the same set of strategies as some other Component. They should be sharing this set, not duplicating it.

Duplication also makes the design hard to understand. It is easier to understand a component as a "derailure" first, and then to look at its weight and price than it is to look at its attributes and strategies and figure out that it is a derailure. Humans naturally categorize objects, and class hierarchies are a natural way to categorize them. Eliminating subclasses might make our software more flexible, but it also makes it hard to understand.

How can you eliminate duplication in a component system and represent categories of similar components when all components have the same class?

Both problems can be solved with the Type Object pattern, which means creating objects to represent the types of Components[Johnson97]. Each Component knows its

ComponentType, and each ComponentType holds the strategies that all Components of that type have in common.

Use the Type Object pattern; i.e. make objects that represent the common features of a category of components, and let each component know its type and access those features by delegating to the type.

As is common with this pattern, a ComponentType will be responsible for creating new Components and initializing them. In addition to setting the type of the Component, the ComponentType will initialize its attributes. This means that ComponentType must know the attributes of a Component of that type. In fact, all Components with the same ComponentType will have attributes with the same name, though the values of the attributes differ. Each ComponentType will have a set of AttributeTypes, and it will create a component with one attribute for each AttributeType. An AttributeType will know the name of its attribute and can also be responsible for knowing whether the attribute is a number, a string, etc. Thus, we'll use the Type Object pattern twice.

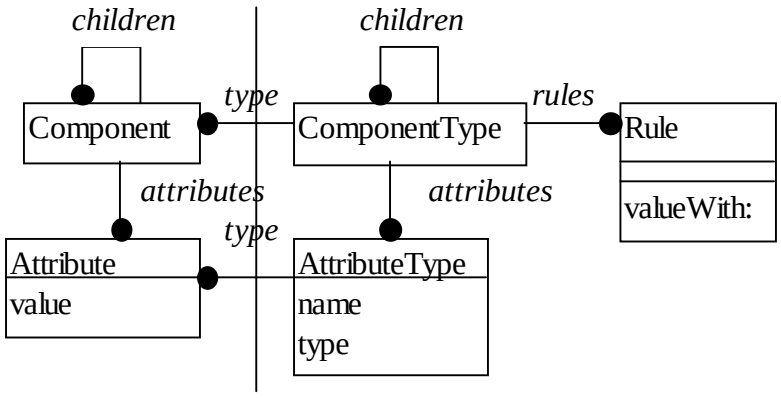

Figure 7: Using the Type Object pattern

The Type Object pattern divides the design into two sides, the instance side and the type side. ComponentType, AttributeType, and Rule (and its subclasses) are on the type side, while Component and Attribute are in the instance side. Although ComponentType and AttributeType both play the Type role in the Type Object pattern, Rule does not. It is just a composite strategy, the Element of the Interpreter pattern. Nevertheless, it is a type side class. This is typical of the Type Object pattern; not all classes on the type side are Types.

An important consequence of the Type Object pattern is that it makes building a generic editor easy. A single editor can edit any kind of composite object and will automatically work with a new kind of component whenever a new ComponentType is defined. Because insurance workers have different conventions than bicycle designers, they will probably need different editors. But a single editor can work for any insurance policy, or for any bicycle design.

A salesperson taking an order for an insurance policy or a bicycle will use the editor to fill out the order. The editor will present attributes whose values must be specified, and will provide a list of components that can be added. When a new component is added, such as adding an automobile to an insurance policy, then the set of possible components is changed so that the salesperson can add components to that new component.

To support this feature, each ComponentType must know the possible ComponentTypes of its own Components. This not only makes it possible for the editors to ensure that only correct composite objects are generated, but permits Components to check that their Components are of the right type. So, the Component editor can start at the root and build up a complex object.

The Type Object pattern usually has a database of types. This forms a new kind of reuse; people can build up a set of ComponentTypes for a particular domain and can reuse them to make new composite objects. In the UDP framework, the database of ComponentTypes is arranged in a tree, with the root of the tree being the type of the root of the product, and its children being the ComponentTypes of the components of its instance.

Pattern 6 – History (also "Historic Mapping"[Fowler97])
Many business applications have to deal with history. In a bicycle management system built from the UDP framework, part definitions change over time, and users need to look at old part definitions. The parts inventory changes over time, and we have to keep track of what was in the warehouse at a particular point in time so we can retroactively adjust individual retailers' pricing deals. So, we need to keep a history of both the part definitions and the parts. In fact, we might need to keep alternative histories, since parts designers might need to have provisional specs for what-if purposes.

This means that each attribute object has to keep a historical record of its values. An attribute must not only know its value, it must know its value at any point in time. It also means that there must be a historical record of the state of the part definitions. They do not use attribute object, so a slightly different solution is needed, though the same pattern.

How does an object keep track of how its value has changed over time?

There are lots of possible ways to keep track of old values. An object could convert every operation into a transaction, keeps its original value around, and replay those transactions whenever it needed the old value. This can be slow and might require finding the original values of other objects, as well. Alternatively, it can keep track of how each value changes at each point in time. This might take more space if each transaction can change several values.

There are also several ways to specify the version of an object that you want to read. One way is to use a global variable (or a Singleton[Gamma95]) to store the time. This

avoids changing the interface of the object. However, this makes multithreaded system difficult. The other way is to make the date an explicit argument.

Change the interface of methods to take a date. Instead of "children" it should be "childrenAt:", instead of "value" it should be "valueAt:[valueUsing:]". Change all variables to be a history, where a history is a sequence of associations whose key is a time and whose value is the value of the attribute at that time. An object can find the value at a particular point in time by scanning the history sequence

The history mechanism can be used for any kind of value. Histories are used to model component definitions and attribute definitions, as well as the value of attributes. When they are used to model component definitions, the values are collections of types and components. So, the history object also needs an interface for accessing collections.

One of the problems of using the History pattern is that programs must keep track of time. The easy way to do this in the UDP framework is to include it as part of the context. Thus, instead of using a date as an argument, the UDP framework uses a Context. The operations on History objects are thus valueUsing: aContext to read a value, value: aValue using: aContext to change it, and add:using: and remove:using: to change collections. Note that the design doesn't change much, except that the signatures of a few of the operations change. This is typical of the History pattern.

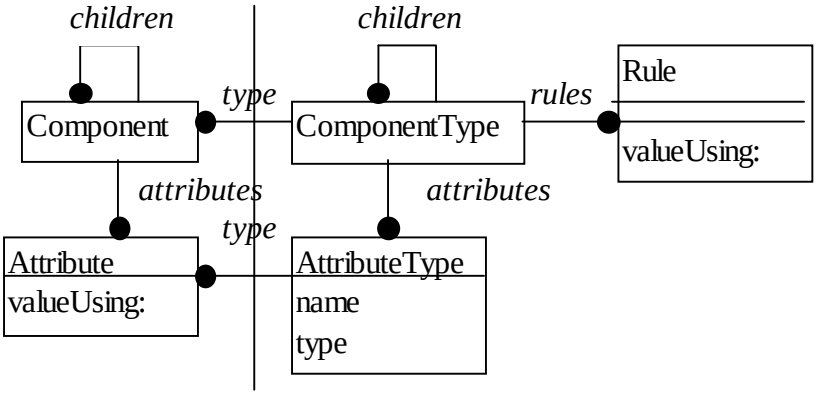

Figure 8: The Effect of History

Pattern 7 - Transactions

The simple version of History is not enough for the UDP framework. Not only does it need to keep track of when a legal value changes, it must keep track of the time that it changed in the computer. Suppose that we changed the price of a policy rider effective June 1, but we didn't enter that price until May 20. A price quote on May 15 for a July policy will be different from a price quote on June 5 for the same policy.

The UDP framework keeps track of both the effective time of a transaction and the time it was actually processed by representing transactions as objects. A transaction includes its effective date, the date it was processed, and its termination date. Transactions are important to the UDP framework because they are a link to other systems such as a workflow system and to an accounting system. In a workflow system, they signal a change of state. In an accounting system, they represent a transfer of funds between accounts.

The UDP framework uses transactions, not time, as the key to history. One particular transaction is always specified to be the "viewpoint". It is part of the context and is used whenever the system reads or changes an attribute, parent-child relationship, or rule.

A transaction can specify the new value for a set of attributes. Some transactions are temporary and have a termination date. In those cases, the attribute reverts back to its old value after the transaction.

The value of an attribute is given by the transaction with largest effective date, but whose effective date and process date are less than that of the viewpoint, and whose termination date is less than the effective date of the viewpoint. In general, all the transactions that change a value must be examined to compute the value. Usually there are no more than a few transactions that affect any particular value, so this is fairly fast.

Pattern 8 - Strategy

A component presents a different set of attributes in different contexts. Some attributes are optional and don't need to be displayed when the component is being edited, but might still be involved in rules when the component is being evaluated. For example, some of the attributes are only used to compute functions on parts of the tree, and are initialized only when the component is evaluated. So, the component needs to be able to provide different views of its attributes. Unfortunately, every type of component might need a different algorithm for filtering its attributes.

How can a component be parameterized with the different views of its attributes?

Make a Strategy for each view. Each component needs a set of strategies, one for editing the component, one for evaluating the component, and so on. These strategies are stored in the ComponentType, along with the rules for evaluating components. In fact, the strategies are built from the same sort of Rules as those rules.

Implement the view of the attributes as a Strategy.

Pattern 9 - Decorator

Sometimes attributes need to have rules. For example, if a life insurance policy is for over $1,000,000, the insurance company is required by law to reinsure the excess. The policy must identify the amount reinsured and the reinsuring company. These are

optional attributes that occur when the coverage amount is over $1,000,000. If we could give a rule to the attribute that held the coverage amount, then that rule could add the optional attributes when the coverage amount was over the limit. One solution is to give every attribute a "changeRule" variable, but most attributes don't have a change rule. It would be better if we could only give attributes rules if they needed them.

How do you add behavior to an existing object?

One way to simulate adding behavior to an existing object is to make a new object with the new behavior, initialize it with the state of the original object, and replace the original object with the new object. But if you want to add behavior without replacing the object, you can use the Decorator pattern[Gamma95].
Decorate the object with another object that adds the new behavior, but delegates most messages to the first object.

An AttributeDecorator would be an object that stands in for an attribute, that has the same interface as an attribute, and that references an attribute. It would delegate most operations to its attribute, but it would also contain a rule that it would evaluate whenever it delegated an operation that would change the value of the attribute.

Conclusion

If we put the models of Figure 5 and Figure 8 together, we see the core of the UDP framework (in Figure 9). The UDP framework uses a few little tricks when it combines these models. In particular, AttributeType is a Rule. Evaluating an AttributeType returns the value of the Attribute with the same type (i.e. same name) in the context.

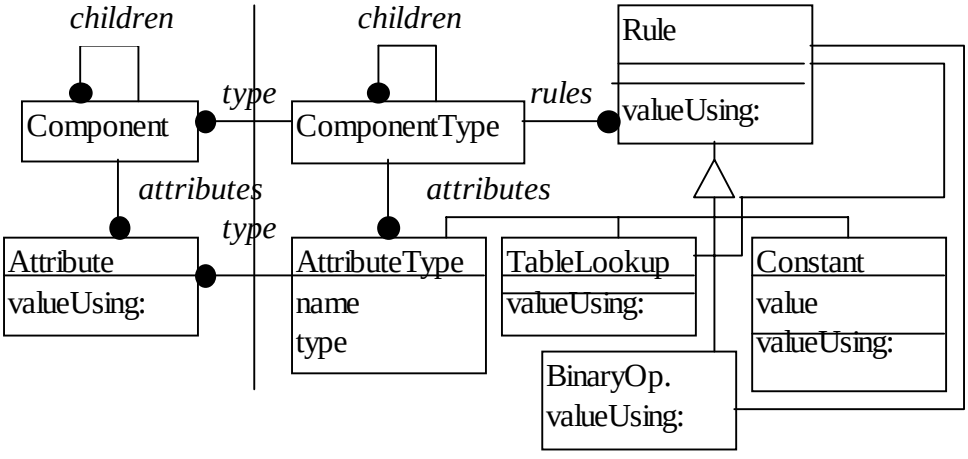

Figure 9: The UDP Framework

The UDP framework is more than just a framework, it is also a object modeling notation. In effect, we have created a new object-oriented language. Since it is implemented in Smalltalk, and since we are Smalltalk fans, it is reasonable to ask whether it is really a good idea to invent another language. What is wrong with just using Smalltalk?

The advantage of the UDP framework is that it is special purpose. It is not a general purpose modeling notation, but is specialized for modeling complex structures that not only change in time (and we want a complete history of how they change in time) but whose composition rules change over time. The framework automates persistence, user interface, and history, but it does so at the price of ignoring all behavior other than evaluating rules. As long as the behavior you want fits into the framework, the UDP framework can make it easy to develop an application. Although it is always possible to revert to Smalltalk to add new behavior, the UDP framework only provides a big advantage if it can be used for most of the functionality of your application. Fortunately, there are many applications for which the UDP framework seems to be useful.

References

[Johnson97] Ralph Johnson and Bobbie Woolf, Type Object, to appear in *Patttern Languages of Program Design* 3, Addison-Wesley, 1997.
[Lieberherr96] *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS, 1996.
[Beck97] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice-Hall PTR, 1997.
[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[Fowler97] Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.