The Giza framework: multiple views using patterns for local method binding

Marc-Antoine Parent
Human-Computer Interaction group
Centre de Recherche Informatique de Montréal (CRIM)
1801, McGill College Avenue, Suite 800
Montréal (Québec) Canada H3A 2N4
maparent@crim.ca

Abstract

Complex data can be analyzed, structured and represented in various manners. Visualization tools should support concurrent representations of many organizations of data. It is possible to do this using subject-oriented programming design environments, where objects offer different interfaces to different "subjects", or tools. In order to provide similar services within Java, we have remodeled a few key methods, especially accessors, of our model objects as strategy objects. The choice of strategy object from a message selector object is local to a method binding context object. Various views are assigned different such contexts, and hence execute different strategies to navigate or manipulate the data. This explicit manipulation of methods as objects allows dynamic building of new methods using composition or decoration patterns, thus providing some adaptive capability. We have used this approach to build an experimental framework that combines traditional and innovative tree representation tools, and allows analysis of derived attributes.

a Research paper
for the
OOPSLA '98 Conference

keywords: subject-oriented programming, object paradigms, design patterns

Contact info: Marc-Antoine Parent
e-mail: maparent@crim.ca
phone: (514) 840-1234
fax: (514) 840-1244

Introduction: tools to represent structured data

The work presented in this paper attempts to solve a fairly classical problem: to build reusable tools for structured data representation and analysis. We[1] are a small team that does research on data visualization within the Human-Computer Interaction group of the Centre de Recherche Informatique de Montréal. We have developed tools for representation and exploration of hierarchical structures, and use them to represent diverse data sets. Most of the data sets we are dealing with are partly, but not fully, amenable to strict hierarchical analysis. Class inheritance hierarchies, where multiple inheritance can be found, provide one example of such a data structure; it is a particularly rich one, because the inheritance hierarchy is intermingled with other structures, such as run-time aggregation hierarchies, call graphs, delegation chains, etc. Not all systems are equally complex; but most depart in some way from a simple hierarchical structure. For example, file systems are basically hierarchies with cross-hierarchical links. We have also worked on representing family trees, where marriage between cousins and remarriage add another layer of complexity to a basically simple hierarchical structure. Thesaurii and conceptual maps tend to be structured as lattices more than as hierarchies. Of course, we cannot fail to mention connection between pages in a web site, a popular area of representation recently, and rarely hierarchical at all.

An important array of data representation and exploration tools has been applied to this variety of data models. The layout of graphs is a topic of extensive ongoing research, and many strategies have also been developed to represent hierarchies, which are in some ways more amenable to display. Most commonly, hierarchies are viewed one neighborhood at a time, as in Yahoo, or as a partially collapsed linearization of the hierarchical structure (as in Microsoft Windows Explorer^{tm}.) Most hierarchies are too large to be viewed as a fully developed tree, but more advanced forms of representation exist, based on focus and content techniques, alternative layout algorithms (cone trees, perspective wall, tree-map^{tm}, etc.) or various forms of distortion (fish-eye, hyperbolic view, etc.)[2]

Our team has developed strategies (Cheops^{tm}, Millepede) based on the technique of visual elision that we have described in other papers [BPV96, Bea97, PV97]. One particularity of those approaches that is relevant to our software architecture is that many nodes of the logical structures are often mapped onto a single node of the visual structure. This means that visual objects that represent node elements

[1] Luc Beaudoin, <lubeaudo@crim.ca>, who designed Cheops, expert in visual communication; myself, <maparent@crim.ca>, who designed the Giza architecture, always yearning for abstraction; and Louis Vroomen, <vroomen@crim.ca>, who makes it all come alive on the screen, expert in instrumentation.

[2] Many references testify to the richness of that field. A selection of web-based references is maintained by one of the members of our team at http://www.crim.ca/~vroomen/mainPages/visual.html

are less in number than the nodes they represent. For that reason, the view cannot rely on the visual objects to store its view-specific state information.

The goals of the Giza architecture are to provide a visualization toolkit where various structure-representation tools, those of our team and others, could be used in various combinations to represent various aspects of structured data. As our first set of tools were directed towards representing hierarchies, we concentrated on hierarchical data structures (as opposed to more general structures like directed graphs) but we were keenly aware that few "pure" hierarchical structures existed, and that many seeming hierarchies contained the odd cross-link. Furthermore, it is sometimes informative to represent covering trees extracted from directed graphs. In either case, one wants to represent a subset of the links found in the structure. How does one specify whether to traverse all the links of a structure or only those of a covering tree? To compound the problem, as mentioned in the class hierarchy example, many complex models are traversed by more than one structure. Even seemingly strict hierarchies like that of animal taxonomy are criss-crossed with food chains, symbiotic-parasitic relationships, etc. One would like to apply visualization tools alternatively (or simultaneously) to the speciation or to the predation relationship in biology; to inheritance, delegation or aggregation in software engineering; to a strict matrilineal or patrilineal view of a family tree.

Model and view; which should shape the other?

Many traditional OO modeling techniques explain how to build an object model that fits a conceptual model adequately, and how the interface objects (views and controllers, to use the MVC vocabulary [KP88]) should then aim to reflect the internal organization of that object model. Clearly, this is a proper way to build well-integrated applications, although it has been argued that a danger of this approach is that the interface may get to reflect the computational model at the expense of the user's model of the data. However, interface components designed in this fashion are not necessarily reusable. They are designed for a specific model, and will not live beyond it.

To build reusable user interface components, on the other hand, it is customary to define a very narrow programmatic interface for the interaction between the model and those components. Then, a model that should be represented or manipulated by these interface components will have to conform to the programmatic interface so defined, either directly or through an intermediate object (using the Adapter design pattern. [GOF95]) We could say that the model is (indirectly) shaped by the view's structural expectations. Each different visualization tool expects some different features in the data model's structure, and satisfying each view's expectations adds more and more constraints on the model objects. Not all structures, for example, offer functions to follow links in the reverse direction; not all graphs are trees, but some are rooted, connected, and can be covered by a tree. We do not want to implant those graph-theoritic notions in any model that contains an object pointer, just in order to be able to represent that model as a graph, and that pointer as a link. Also, just as we do not want the user-interface objects to reflect the model too closely so we can reuse them for another model, we should also avoid tying the model to the views and controllers.

In the examples we have outlined, moreover, there is not one, but many structures. Now, if someone wants to display one of those structures as a tree, the objects in the model should comply to the tree programmatic interface and give access to that structure through that interface. However, doing that makes it impossible to access the other structures of that model through that interface.

But what if we want to represent and explore another of those structures? We'd have to alter the view to use a parallel tree-traversal dedicated to them. Hence the need for mediation, so we can bind one aspect of a model to a role it will play in one given view, while it can still exhibit another aspect[3] to fulfill the same role in another view.

Choosing attributes

The problem we have outlined with multiple structures may be more familiar in a simpler setting; we may have a collection of objects, each of which has various numerical attributes, and we want to build a histogram for each attribute. The solutions for that simpler problem are better known; one simple solution involves a universal "getter" on the objects, that takes an attribute selector as a parameter. While this is acceptable for a closed list of attributes, it is not appropriate if some attributes are computed *a posteriori*, or are imposed by the views.

We have referred a few times now to attributes derived from the views; we will now give concrete examples. Suppose we want a histogram of the cumulative value of a variable along branches of a hierarchy, but restricted to branches we have expanded. The notion of some branches being expanded or collapsed belongs properly to the view, and will not be part of the model. So the histogram will be based on a hybrid value, based on both the view and the model. Another example is found in the setting of focus and context techniques. If a given node or branch is the focus of interest, we will compute the value of a degree of interest function for the other nodes, based on their relationship to the focus. In the Cheops representation, during display and interaction, we frequently use the notion of "family relations" between a node and the currently selected node. Is this node an ancestor, the father, a descendant, a son, an uncle or a cousin of the selected node?

To add to the expressiveness of the system, one should allow the user to define complex derived attributes, based on the relationships between elements of the structure. In a file system, for example, we often want to know the cumulative size of folders; we may also want to know how many HTML documents they (recursively) contain; How many hyperlinks to external documents those HTML files contain, and whether any of those links are now broken... These characteristics are not by rights part

[3] Here, we use "aspect" and "role" informally, without referring to either aspect-oriented or role-oriented programming techniques

of the file system operation, and should not be part of the file system model; neither are they properly part of a web site maintenance system. However, they are valid user queries, and they show the inadequacy of building the views and controllers of a software system around the base attributes of a single model.

Forces

Before we describe the architecture we have designed in order to solve these constraints, let us step back and summarize the forces at play in the problem.

1. ***Reusable software components for many models***

   We are building software components for representing and manipulating an unspecified structured data model. We want to minimize assumptions and constraints made on that model.

2. ***Run-time choice of representing one of many structures or attributes of the model***

   Elements of the model have many attributes, and many structures can be found in the model itself. It must be possible to decide, at run-time, to display or manipulate any combination of these attributes, on any substructure of the model.

3. ***Some of the attributes are not part of the original model***

   Not all attributes to be represented are part of the model; some come from specific views, others are constructed by the user. Similarly, some of the structures manipulated may not come directly from the model, but have been derived from it. (eg. covering tree.)

4. ***Some of the attributes will be shared by many views; others will differ between views***

   We will want to construct views that share some of these derived properties of the model. This is useful, for example, to create manipulation tools. The effect of a manipulation should be reflected in at least one view. However, some manipulations are local to certain views and should not affect others. For example, the collapsing of a branch in a view should not necessarily be reflected in all other views.

5. ***Structures may have many elements: it is preferable to avoid multiple parallel structures***

   If a model lacks a property defined in a view, but which may be shared by some views, it is tempting to create a parallel structure, that will reflect the original model but will add the required property. For small models, this is acceptable; but for models of thousands of nodes, the strain on memory is important. Also, parallel structures have to be kept in synchrony with the “master” model, which would require more code, and more time.

A solution: Subject-Oriented Programming

The above discussion supports the argument that the definition of the model is shared between the model and other software components that interact with it. This point of view has already been argued convincingly by Harrison and Ossher [HO93] and others [ShS89, HO90, etc.]. They distinguish three functions of the object concept: traditionally, the object is used as

- a unit of role, playing a specific part in the model, akin to the notion of interface;
- a unit of behavior, being the basis of message dispatching;
- a unit of storage, the objects having full knowledge of all their attributes

They suggest decoupling those three functions. In particular, they suggest that the behavior (the set of methods) of an object would be specific to the “subject” that manipulates it, i.e. to a software module that expects some specific behavior for the object. The object would have to behave in a way that is a meaningful combination of the behavior expected from all the subjects that deal with it. As for the attributes of an object, they would also be a combination of all attributes expected by all the subjects that deal with the model. Shilling and Sweeney [ShS89] have gone one step further and proposed that attribute sets of an object be indexed, not only by view type, but even by view instance (which they call an activation of a view.) This allows each view to set its own addressing space for derived attributes. These techniques are supported by special development environments and custom-made object-oriented databases.[Har87]

This paper demonstrates how similar functionality can be emulated within a static object-oriented language, using programming conventions and design patterns. It does not claim to replace the work of the subject oriented programming teams, but to make some of the functionality of their advanced systems accessible to the C++ or Java programmer. Of course, such can only be done at a cost, in terms of performance and security, and also requires programming discipline. Yet, the solution that is outlined here opens the door to interesting new programming techniques.

The path from entity to attribute: Accessor objects

The first brick of our architecture is a fairly innocent one, an application of the Strategy design pattern to the model’s accessor methods (what Smalltalk programmers refer to as getter/setters.) Different views access different attributes or structures in the model, because they use different Accessor objects. This directly answers the second of the forces listed above.

The basic Accessor object encapsulates a recipe to obtain (and eventually manipulate) an attribute from an object of the model. How that attribute is actually linked to that object is known only to the Accessor, and maybe to the model object. So an object that needs to obtain some property X of an object Ob of the model would have to first obtain an XAccessor object (a member of a subclass of Accessor specialized in obtaining that property for those objects.) Then, when it actually needed the

X datum, it would call the getData method of the XAccessor object. Note that to define the XAccessor's getData method, we have to create the XAccessor class; this will happen for all Accessor objets that deal directly with properties of the model. Those Accessors will be loosely referred to as classes or objects, according to emphasis. Application of the Singleton design pattern is here appropriate, though not necessary.

(Examples are written in Java, which is Giza's current implementation language)

```
class someModelObject extends Object {
   someClass x; // an attribute
   ...
}

class XAccessor implements Accessor {
   // specialized for the class someModelObject
   // Could be an inner class of it
   // Could also be an anonymous class
   public Object getData(Object anObject) {
      try {
         return ((someModelObject)anObject).x;
      } catch (ClassCastException e) {
         return null;
      }
   }
   public Object setData(Object anObject, Object datum) {
      try {
          ((someModelObject)anObject).x = (someClass)datum;
      } catch (ClassCastException e) {
            throw new RuntimeException("some message");
      }
   }
   ...
}

XAccessor myXAccessor = ...;
Object modelObject = ...;
Object anAttribute = myXAccessor.getData(modelObject);
```

In this example, the XAccessor simply encapsulates a roundabout way to obtain a traditional instance variable; but many other types of Accessor objects exist in Giza. They can obtain attributes from an independently maintained dictionary, thus:

```
class HashAccessor implements Accessor {
   // Generic hash-based Accessor
   Dictionary d = new Hashtable();
   public Object getData(Object anObject) {
      return d.get(anObject);
   }
   public void setData(Object anObject, Object datum) {
      d.put(anObject, datum);
   }
   ...
}
```

In contrast to the model-bound Accessors, each instance of the HashAccessor class will deal with a distinct attribute of a distinct class of the model. As a variant of this, an object-oriented database facility, such as that used by Harrison and Ossher's team, would lend itself quite easily to mediation by Accessor objects. This would probably enable Giza to interact with existing subject-oriented code from its own standard Java VM environment.

Accessor objects, simply put, can encapsulate any of the numerous ways through which it is possible to associate a data structure to another. As a final example, some model objects in Giza host a linked list of tagged attributes which allows the use of the notion of Facet, as defined in [Gam95], Obviously, it has proven simple to write accessors to fetch attributes defined as Facet objects. We see how these various structures simply answer some requirements of force 3.

This simple application of a known pattern pervades the Giza programming methodology, and marks an subtle yet deep change in the approach to data. This paper shows how Accessor objects can be used, selected, assembled, and built; and how they can be building blocks for an implementation, within a classical static object-oriented language, of a subset of subject-oriented programming.

Between the model and user interface: Context objects

How are those Accessor objects obtained and selected? Having established that many attributes do not naturally belong either to model objects or to view objects, it remains to be stated where exactly they should be defined. In force 4, we have stated that some attributes must be shared between views, to ensure the coherence of the application; and yet should not be shared by all the views, to allow the construction of many simultaneous divergent representations of the same underlying model.

To achieve this, the approach developed in Giza does not exactly follow the approach of the subject-oriented programming community, in that coherent attributes sets are not defined at the level of the view class or view instance. This is where we introduce the keystone of our architecture, and define an explicit intermediate level where a point of view on the model is represented by a specialized object, called a Context. Context objects are themselves linked in a strict hierarchy; some contexts are said to be more specific than some others. The parent Context is said to be wider than its sub-Contexts. Any software component that interacts with the model or models should first bind itself to a single Context object, and that object will act as a first mediator for interactions with the model.

Any Accessor object, linking to an attribute of the model, will be known to the view through its primary Context object, or a wider Context. Two views that share the same Context object will be effectively working on the same model; any Context can locally override any of the attribute definitions for the model, and so two models bound to different Context objects will have “a different point of view” on the model, will effectively work on a different structure through their different Context. This hierarchical organization of Contexts, on the other hand, should be adequate to provide coordinated views.

Choosing an Accessor in Context

Once the view is bound to a Context object, it will ask it to provide an Accessor for each attribute it expects to find in model objects. To allow this, the view must have a list of Selector objects, so named after method selectors in Objective-C, that describe what attribute the view expects to obtain with a given Accessor. The Context itself has a collection of Accessor objects that it uses, and that effectively define it; it will first attempt to provide an Accessor that fits the requirements of the Selector from that collection. If it does not find any appropriate Accessor, it will try to create one (this process will be described in another section) or get one from its parent Context. Some Accessor objects come with the model, and are directly accessible to all Context objects.

In general, it is assumed that a Context object is primed with a set of Accessor objects that collectively mediate access to the model in a manner specific to that context. Setting up a coherent Context is a crucial step in using Giza, though it is possible to provide enough Accessor objects with the model and views to ensure complete functionality in a blank Context.

Picking an Accessor from the crowd

Each Accessor has a Signature, specifying its name, the classes it expects to link (parameter type and data type) and a few other parameters. The Signature objects are obviously related to Selector objects, and are indeed a subclass. In the simplest case, a Context that is asked to find an Accessor related to a Selector will simply search its collection of Accessors for one whose Signature matches the Selector. However, in general, a model and a view need not share the same terminology for attributes. This is why the Context object actually knows its Accessors through a list of Selector-Accessor pairs. These Selector-Accessor pairs are objects in their own right, and are called FunctionalBinding. When asked to find an Accessor for a Selector, the Context will actually look through these FunctionalBinding and attempt to find one with an identical or compatible Selector. A Selector can be compatible to another if the class expected by the first is a super-class of the second, and the class returned by the first is a subclass of the second.

Accessors are always known at least through their Signature, so any Accessor in the list is at least represented once in a pair with its own Signature. The list is indexed by the Selector’s name, for convenience and efficiency. Thus, setting up a Context consists primarily in providing for it a set of FunctionalBindings.

Of course, in some cases, it is possible that many FunctionalBindings provide compatible Selectors for a given Selector of the view, all with the same name. A mechanism has to be provided to arbitrate such conflicts, by minimizing “Class distance” between Selectors. Such a mechanism will clearly replicate some of the algorithms of the object-oriented message dispatcher of the language. In a way, that is needless; but it is also an opportunity for flexibility. This binding of Selectors to Accessors is very much what a message dispatcher does; and it is where one must act to allow a Context-local interpretation of the model. In theory, each Context object could be built with its own dispatching rules and algorithms. In Giza, only the simplest Selector-matching algorithm has already been implemented, and it has been adequate for many applications.

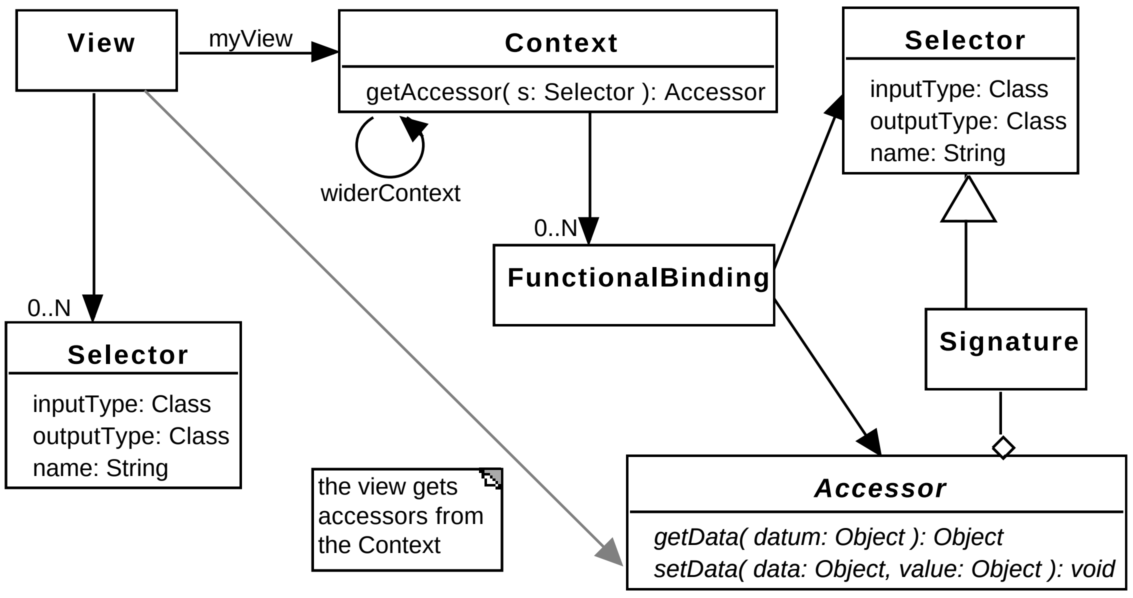

Fig 1: relationship between View, Context and Accessors

Accessors within Roles

More precisely, it has been quite adequate because the models Giza currently handles have complex structures, but simple class structure. An Accessor that is found for one object of the model will handle all similar objects, because they can be expected to be of the same class. In a complex model, however, one can expect many interrelated classes and subclasses. In many cases, the view will expect those different classes. However, we know that we should program the view without making explicit references to the classes of the model, which should be changed easily, fulfilling the requirements of force 1.

So the view needs more than a list of Selectors to define the Accessors it expects of the model; It should specify a list of Selectors for each different class it expects to find in the model. But the classes of the model are still unknown; so the view should have an object that embodies its expectation of various entities it will manipulate, that will play various roles in the view's representation. These Role objects own a list of Selectors, and will be matched to classes of the model in RoleBindings, just as Selectors are matched to Accessors in FunctionalBindings. This, however, means that the Selectors in the view's Roles will themselves be expressed in terms of Roles. The algorithm to match these Role-based Selectors to class-based Signatures is somewhat complex, as it depends on the RoleCasting, i.e. the entire set of RoleBindings. Again, providing the Context objects with an adequate RoleCasting becomes a fundamental condition to use Giza adequately. This part of the system, however, is undergoing further refinement.

Dynamic Accessors

So far, we have described a complex system to obtain one datum related to an object, through the mediation of a Context and an Accessor object. At this point, the performance costs of the flexibility we introduced may seem prohibitive. The Context's search for the proper accessor is turning into a formidable enterprise. If this has to be performed every time we query an attribute of an object, the performance cost will be overwhelming.

An obvious solution is to cache Accessors. The view can obtain all the Accessors from the Context as it needs them to treat objects of the model; it should then store those Accessors in private instance variables and use the same Accessors whenever it has to obtain the same attribute from an object playing the same role. The cost of going through an Accessor object is not considerably more than a direct access; We have to fetch the reference to the Accessor itself, and add one message dispatch to call the getData method. This more or less doubles the time for any access, not a serious consideration. The real cost is at initialization, when we obtain the Accessor objects from the Context.

What if the model object we hand over to the Accessor is actually of a different class? In most cases, it will not matter. The HashAccessors only deal with Objects; the FacetAccessors all share a common subclass; and in the case of instance variable Accessors as in our first example, it will work for all subclasses of the someModelObject class. We can be more careful and use a conventional accessor method and the language's own dispatch capacity. Let us rewrite the Accessor from our first example:

```
class someModelObject extends Object {
    someClass x; // an attribute
    ...
    someClass getX() {
        return x;
    }
}

class anotherModelClass extends someModelObject {
    someClass getX() {
        ... // some other behavior
    }
}

class XAccessor implements Accessor {
    // specialized for the class someModelObject
    // Could be an inner class of it
    public Object getData(Object anObject) {
        try {
            return ((someModelObject)anObject).getX();
        } catch (ClassCastException e) {
            return null;
        }
    }
    ...
}
```

This Accessor can handle objects of either the someModelObject or anotherModelClass class gracefully.

That handles *most* cases, but cannot be considered safe practice. What if we have reasons to expect many unrelated classes being used for the same Role? We have enclosed all our typecasting in try clauses, and do not yet offer adequate exception handlers in the catch clause.

Actually, if another model object with a different class was introduced, the proper action to take would be to re-fetch a new Accessor from the Context and the original Selector. It is possible for the exception handler to do this. However, such a handler would burden all Accessors. A simpler way to achieve the same effect is to wrap the Accessors in a Decorator that handles this case.

```
class XAccessor implements Accessor {
  // specialized for the class someModelObject
  // Could be an inner class of it
  public Object getData(Object anObject) {
    try {
      return ((someModelObject)anObject).getX();
    } catch (ClassCastException e) {
      throw new WrongAccessorException();
    }
  }
  ...
}

public class AccessorCache implements Accessor {
  // a generic decorator
  private Accessor cache = null;
  public Selector definition;
  public Context myContext;

  public AccessorCache(Selector d, Context c) {
    definition = d;
    myContext = c;
    cache = myContext.getAccessor(definition);
  }
  public Object getData(Object anObject) {
    try {
      return cache.getData(anObject);
    } catch (WrongAccessorException e) {
      cache = myContext.getAccessorForClass(definition, anObject);
      return cache.getData(anObject);
    }
  }
  ...
}
```

This Accessor can be used instead of one the Context directly provides, and will always be valid. The cost is another method dispatch.

A variant of this AccessorCache offers important possibilities: though it has not been stated up to now, it is quite possible for the bindings in a Context to vary at run-time. If a view caches Accessors directly, it might not be aware of such a change of Context. However, we could make Contexts observable, and AccessorCaches could register interest in the Context they are based on, and update the Accessor in their cache as needed. Thus, not only do Contexts offer multiple local views of a model, but the configuration of those views can be deeply changed at run-time, and the views will indirectly adapt to that change.

Observing accessors

Using Accessor objects involves some cost, but we are beginning to see the benefits. We have seen how they can be decorated with observers, to become dynamically tied to the context. They can also be decorated with observables, and give an entirely new twist to the Observer design pattern.

In the original Observer design pattern, an observer object subscribes to changes in an observed object. Often, it is interested in changes in a specific attribute of the observed object. For that reason, the notification of change message may contain some clue as to which attribute was affected.

Now, suppose that we replace, in all Contexts, some Accessor object by a version of the same Accessor decorated with the following class:

```
public class ObservableAccessor extends Observable, implements Accessor {
  private Accessor myAccessor;
  public ObservableAccessor(Accessor anAcc) {
    myAccessor = anAcc;
  }
  public Object getData(Object modelObject) {
    return myAccessor.getData(modelObject);
  }
  public void setData(Object modelObject, Object datum) {
    myAccessor.setData(modelObject, datum);
    setChanged(modelObject);
  }
  ...
}
```

In that case, we would not register interest in an object, but in an attribute, and the object would be given as a parameter in the notification.

As stated in force 5, we are dealing with important collections of objects; If we are observing one attribute for all the objects in a graph, it means we have only one object to observe instead of many thousands. Of course, if there are many observers, and each needed only to observe a subset of the observable objects, there will be many spurious notifications. But there is at least one situation where such a system will be highly beneficial: when using the Observer design pattern to maintain a constraint between attributes.

Let us take one example: suppose we are dealing with a tree, and the tree nodes have a certain numeric attribute n. If we want to maintain a running sum N of that attribute at all nodes of the trees, it would be simple to set an Observer for the nAccessor, and that accessor only; whenever any other process altered the value of n for one node, it would do so through the ObservableAccessor's setData method. The Observer would be notified and could easily update the N value for the node and all its ancestors.

Accessors as building blocks

We have seen how Accessor objects reify dynamic links between model objects and attributes. Being themselves objects, they lend themselves to manipulation, and are easily integrated in various design patterns. Most notably, Accessors can be involved in Composition. If an Accessor A allows to obtain an attribute X of model objects of class C, and those X attributes themselves are model objects of class D, for which we have a an Accessor B giving attribute Y, it is easy to obtain by composition an accessor [BoA] that will yield the Y attribute from an object of class C.

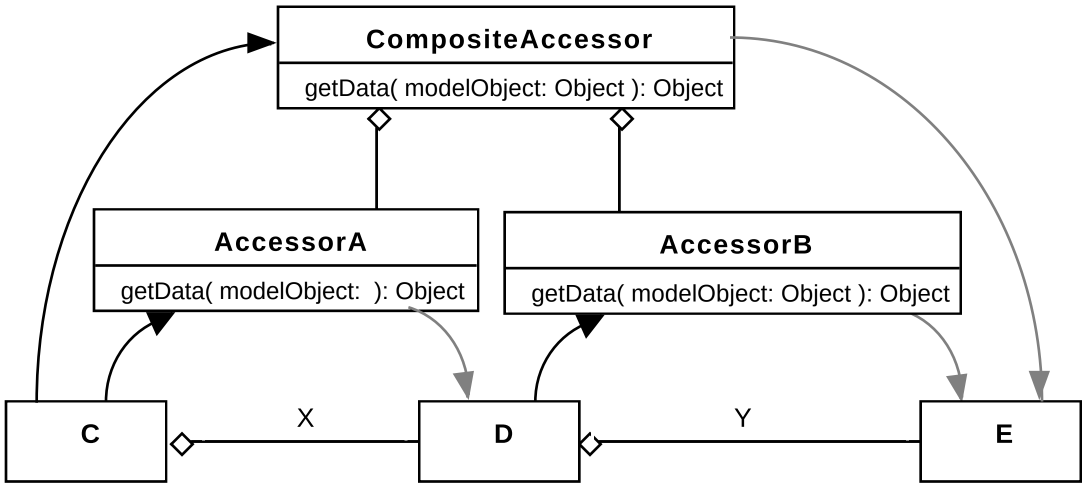
CompositeAccessor
getData(modelObject: Object): Object
AccessorA
getData(modelObject:): Object
AccessorB
getData(modelObject: Object): Object
C
X
D
Y
E

Fig 2: Accessor Composition (curved arrows represent parameter passing)

This is the most simple case; because of its rooting in graph-theoritic concerns, Giza defines primitive concepts for neighborhoods and operations. Each object in a structure has Accessor objects that return various neighborhoods around an object. These objects, called TraversalStrategies, combine the Smalltalk concepts of Collection and (ordered) Iterators. Example neighborhoods of a tree structure include the node's Ancestry, Sons (first-generation descendants), DepthFirstDescendants, BreadthFirstDescendants, etc. Graph structures add the notion of multiple, Parents, with the derived notions of DepthFirst- or BreadthFirstAscendants.

The advantage of returning Iterators directly instead of a Collection is that it simplifies construction of operations. Visitor objects can be thrown along any TraversalStrategy (objects that contain a routine that will be called for each object in the iteration, as per the Visitor design pattern.) Some of those Visitors are Operations, based on another Accessor object. We can define a complex composite Accessor from a TraversalStrategy-yielding Accessor, an Operation, and some attribute Accessor. This allows to define at run-time, by object composition, an Accessor that would, e.g. calculate the average age of all children of someone in a family tree, or accumulate a histogram of the number of children in its entire descendance.

Those complex operations allow one further variant: TraversalStrategies can be restricted (by decoration) with a BooleanFilter, which can itself be constructed from a condition applied on the return value of some Accessor. Thus, it would be easy to restrict the data-gatherers above to female members of the family, or even female lines. All this, again, using some complex composition on a limited vocabulary of Operations, Conditions, and Context-chosen Accessors. This is how we fully answer force 3.

Constructing Accessors

Of course, it is not enough to say that such composite Accessors *can* be constructed at run-time; we must now describe how to do this. First, the task of creating new Accessors lies, again, with the Context objects. When a view requires an Accessor for a certain Selector (within a certain Role), the following happens:

- the Context would first try to find an existing Accessor in its list of FunctionalBindings, according to the RoleCasting.
  - In case of failure, the Accessor is sought with the parent (wider) Context objects (and recursively.)
- In case of failure, the Accessor is sought in the model's base Accessors.
- Finally, if that also fails, it will try to create the missing Accessor.
  - In case of success, the new Selector-Accessor pair will be added to the Context's functional binding.
  - In case of failure, the parent Context object (and, recursively, ancestors) will also try to create the Accessor according to their own creation rules.

The Context is thus involved in something akin to the Abstract Constructor design pattern [Lan96]. It attempts to create an Accessor object from an abstract specification of the product's functionality, the Selector. However, the Selector as defined does not contain enough information to specify the structure of a complex composite Accessor. Also, in many cases, a very general rule can be applied; for example, many properties that are not otherwise specified are to be specified using HashAccessors. What we need is, not one object, but many objects to keep track of the ways in which missing Accessors can be constructed. So we finally install the final stones of the Giza architecture: CompositeSelectors and Scripts.

First, we will define a subclass of Selectors, that can carry composite definitions. Their structure will essentially reflect that of the composite they describe, and depends on a well-defined language of Accessors, Operations, TraversalStrategies, BooleanFilters, etc. such as has been hinted at in the previous section. We are currently developing this language, and a syntax to describe it. This allows to build complex Accessors or Selectors from a configuration file that uses that syntax.

In fact, once we can create composite Selectors or Accessors using a specification grammar, we hope to extend that mechanism to alternate grammars, using the Interpreter design pattern, so we can build Accessor objects directly from an OQL request. Conversely, we have already hinted that some Accessors might obtain their data through a database systems, and can be considered equivalent to a SQL or OQL request. Though it is possible to use those Accessors as building blocks within a composite Accessor, it would in general be wasteful. Instead, we hope to develop a translator that can transform a CompositeSelector back into the appropriate SQL or OQL statement, which can be encapsulated into a simple Accessor. But all this area remains to be explored.

However, in general, these CompositeSelectors are not used directly by the view. They are hints from the Role, specifying that if a simple Selector cannot be satisfied directly, it can be satisfied through composing other Accessors in a certain way. Why do we not use those CompositeSelectors? Because in some cases, it is possible that the composition described by the CompositeSelector corresponds to a method which exists directly in the model currently being used, for which the Context already has an Accessor object, currently identified by a simple Selector object. It is possible that those simple Selectors do not match, in the which case we have to make their equivalence explicit in the Context's setup. This is not desirable, but the alternative is to identify the model's original Accessors with CompositeSelectors, and compute equivalence relationships between CompositeSelectors when we want to find an Accessor, or equivalently to provide a normal form for the CompositeSelectors. Given the generality of these selectors, that latter problem is likely to be intractable.

Creating families of Accessors with Scripts

Specifying individual FunctionalBindings between Selectors and Accessors is often unnecessarily tedious. In many cases, one has heuristics to propose what Accessor will be used for what Selector in what Role. These heuristics can be encapsulated in a Strategy object, installed in the Context, called a Script. It is the Script that actually tries to build an Accessor from a Selector. The Context will not itself try to build new Accessor objects, but will rely on a series of Scripts that it has been instructed to use. Instead of priming a Context with a long list of FunctionalBindings, the Giza developer should prime it with a few well-written Scripts. Some of these Scripts use simple heuristics, as outlined above. For example, "anything unknown can be handled by a HashAccessor". Of course, in general, a Script should be much more specific, and only act for some well-defined problems.

Let us analyze Scripts further. A Script carries an algorithm for building Accessor objects from a Selector (simple or composite.) The Script will use other Accessors as building pieces, and needs to have a global understanding of the model before it can build the missing Accessors. For that reason, it should have access to the Context's FunctionalBinding and RoleCasting. Some of the Accessors it will use as building blocks may not originally be part of the model, but may be easy to construct (possibly through other Scripts.) We will illustrate this through an elaborate example.

Suppose we have a view that expects to display a tree. It has a tree Role, with Selectors for access to Parent, Sons, and DepthFirstDescendants of a node. Let us suppose our base model is a graph, and offers Accessor objects for Sons and Parents. The view will ask the Context to satisfy the tree Role. One of the Context's Scripts may know that it can create a ParentAccessor by priming a HashAccessor through a recursive BreadthFirstDescent from a root node in a graph. (This is a classical algorithm to create a partial covering tree.)

So the Script will initiate a request in the Context for an Accessor object that yields the BreadthFirst-Descent attribute, and another Accessor that yields the RootNode attribute. Recursively, the first request will trigger another Script, which is able to create a BreadthFirstDescentAccessor from the SonsAccessor, which is already available from the graph model. That Accessor is created, and provisionally added to the list of FunctionalBindings. If the Giza developer has had the foresight to provide an Accessor that can lead to a choice of root node for the graph, the original Script will be satisfied, and will terminate. (It is easy to create such an Accessor, if only by defining an Accessor that returns a constant value.) The successful termination of the Script will confirm the validity of the graph BreadthFirstDescentAccessor that was created as an intermediate structure. If no Accessor can be found for the RootNode, and no Script can create it, the ParentAccessor-making Script will fail. In that case, we should prompt the user to add Scripts or FunctionalBindings to the Context.

We see that we have a rather complex problem of graph traversal; each Script will require some Accessors that will trigger other Scripts that will require Accessors, etc. If we applied brute recursion, there would be potential for loops and deadlocks. Fortunately, graph traversal is a well known, tractable problem. Note that it has not been implanted in Giza yet. Currently, we know we can rely on parametrized FunctionalBindings, or very simple Scripts. The remainder of this section describes how we plan to expand Giza.

This automated creation of Accessors by Scripts may seem able to solve all our problems, except that it does not. The Context still contains a SonsAccessor taken from the graph model, that will not be compatible with the SonsAccessor expected by the tree Role. Should the Script that created the ParentAccessor handle the SonsAccessor as well? Even if we trusted it to do so, could we trust it to also take care of the DepthFirstDescent attribute, that can easily be constructed from a SonsAccessor... provided we use the right one?

Clearly, what we are dealing with is the coherence of the Role, and it cannot be handled by something outside the Role. Thus, the Role will contain Scripts, that will be applied after the Context's Scripts and take precedence over them. However, this does not mean that all the Scripts should be handled by the Role, for the Role's set of Scripts is fixed by its nature, whereas the initial setup of the Context's Scripts is the variable that allows the user to use Giza's flexibility. This is still very much an area of open research. We hope that investigation in the field of adaptive programming will provide new insights.

Using Scripts has another drawback, namely that it becomes too easy to create an Accessor object. Many Scripts can potentially create an Accessor, and many may create a woefully inadequate one. (The universal HashAccessor-creating script is especially dangerous in this regard.) Another step in which we plan to extend Giza is to develop a measure of the qualityof an Accessor. Performance considerations will be paramount, and fortunately easy to estimate for various types of Accessors, simple or composites.

On the other hand, once they have been controlled by measure, generic Scripts can be extremely useful. For example, in Java, when we introduce a new object model, we could ideally define only one Accessor for it: the Java (meta-)Class. All others Accessors could then be deduced from the model by a generic Script using introspection. Then, Giza would really adapt to the model with minimal effort.

Related work

The goal of writing reusable tools runs deep in the object-oriented programming community, and many approaches have been designed with this aim in mind. Design patterns themselves, which are very much the language with Giza has been designed, are a step in this struggle to write "universal" software. I have found echoes of similarity in many architectures including Amulet [MMM97], ValueModels [Par97], etc. However, few approaches share with Giza its most basic assumption: if the view is to be really independent from the model, its perspective on the model must be private, and may differ from that of other views. The model is generally thought to have intrinsic properties, that other software components should strive to conform to.

Subject-Oriented Programming is the notable exception. The SOP paradigms goes further than Giza in dispossessing the model of intrinsic reality, for the model is almost defined by the sum of the subjective views on it. Giza, in contrast, admits that there is a model with its own intrinsic behavior, but anything that attempts to interact with that model will get a personal diffracted view. The introduction of explicit point-of-view objects in the model is an innovation of Giza that has an important impact on programming styles.

Another specificity of Giza is that attributes are processes, and those processes are objects. Neither part of that equation is unique to Giza; Code is a first-class object and can be manipulated in many languages including Lisp, Smalltalk, etc. In a simpler way, even the message selectors of Objective-C allow for code assembly. Also, attributes are processes in any architecture that pushes accessor methods seriously, as Smalltalk does. Highly original architectures like access-oriented programming [SBK86] have been built on an appropriate use of accessor methods. But the systematic reification of accessor methods, with the run-time construction of complex accessors it allows, may be unique to Giza.

One weakness of the Giza architecture is that we are still looking for ways to analyze how to assemble Accessor objects into optimal constructs. This is probably a problem familiar to designers of object-oriented database systems, and it is probably in that direction that Giza will evolve. In that, it will still follow the path that SOP has taken. But Giza will never need databases; indeed, its promise is in an extremely versatile way to access any data, independently of how it has been stored.

Conclusions

What is the ontology behind that architecture? We have found that the model, though it has a structure of its own, is mostly interesting for what structures you can read from it, or maybe even into it. One could say that structure is in the eye of the beholder. Accordingly, we have removed behavioral autonomy from the objects, and also storage unity. The object does not know alone how to act; the choice of course of action must be context-driven. The object does not know how much information has been associated with it, deduced from its basic properties; and different values may have been assigned to the same properties according to different point of views. So control of the storage of the facets of the object is also taken away from the object.

Also important, the distinction between object and attribute has been blurred. We have objects linked to other objects; they form a huge directed graph, and there are paths along that graph that we can traverse to go from one object to another object that is directly or indirectly associated with it. It is not only that attributes have become objects, or, to use a linguistic metaphor, that the adjectives have become nouns; one would rather say that they have become verbs, that the attributes have become actions. The Accessor objects sometimes do no more than a traditional accessor method, peeking in the object's structure; but in some cases, they construct the information they are asked to give. Their creation is itself an important act in the life of the context. By creating a given Accessor object, or by querying it, we are altering how we view the model, the model itself if not the model objects. Again, the structure is in the act of seeing, and seeing is always performed in context.

What is also original, by reifying these fundamental actions, we have made them amenable to manipulation, just like objects; processes can now be aggregated, observed, decorated, and the binding of "methods" to messages can itself be manipulated. Programmers of dynamic languages such as Lisp, where code is naturally a first-class object, will be familiar with the benefits of code manipulation, but it is welcome news to be able to do so from within a static language, without hampering efficiency unreasonably.

Finally, by subordinating all these to locally defined rules that bind Accessors to Roles in Context, we allow a totally contextual, relative and dynamic definition of data structures.

Acknowledgements

The author would like to first thank Luc Beaudoin for creating the need for Giza; many thanks also to everybody who listened to the various stages of the elaboration of this architecture, read parts of this paper, or even had the courage to use the software. The last mostly applies to Louis Vroomen, but I have also benefited from comments by Nathalie Barthe, Anna-Maria DiSciullo, Patrick Fournier, Cameron Hayne, and Ruedi Keller. Finally, thanks to Frances de Verteuil, director of the HCI group at CRIM, for believing this research was worth the time and investment.

References

[Bea97] Luc Beadoin, “Visual elision: A technique to represent and explore complex hierarchies.” *presented at* Graphical User Interfaces for Hierarchies: A Workshop, University of Maryland, College Park, Maryland, Dec.97, http://www.crim.ca/~vroomen/workshop/slideshows/elision.htm

[BPV96] Luc Beadoin, Marc-Antoine Parent and Louis Vroomen, “Cheops: A Compact Explorer for Complex Hierarchies.” *in* Visualization 96, San Francisco, USA, Oct 1996, pp. 87-92

[Gam97] Erich Gamma. “Facet.” *in* Pattern Languages of Program Design 3, Reading, Massachussets: Addison-Wesley Longman, 1997.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Design. Reading, Massachussets: Addison-Wesley, 1995.

[Har87] William Harrison, “The RPDE^3 Environment - A Framework for Integrating Tool Fragments.” IEEE Software, November 1987, pp. 46-56

[HO90] Brent Hailpern and Harold Ossher, “Extending Objects to Support Multiple Interfaces and Access Control.” *in* IEEE Transactions on Software Engineering, Vol 16, No 11, Nov. 1990, pp.1247-1257

[HO93] William Harrison and Harold Ossher, “Subject-Oriented Programming (A Critique of Pure Objects).” in Conf. Object-Oriented Programming Systems, Languages, and Applications, Washington, DC, ACM, 1993, pp. 411-428

[KP88] Glenn E. Krasner and Stephen T. Pope. “A cookbook for using the model-view controller user interface paradigm in Smalltalk-80.” *in* Journal of Object-Oriented Programming, Vol 1. No 3, Aug/Sep 1988, pp.26-49

[Lan96] Manfred Lange, “Abstract Constructor.” *presented at* writer’s workshop of EuropPLoP-96, http://www.cs.wustl.edu/~schmidt/europlop-96/papers/paper26.ps

[MMM97] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski and Patrick Doane, “The Amulet Environment: New Models for Effective User Interface Software.” IEEE Transactions on Software Engineering, Vol. 23, no. 6. June, 1997. pp. 347-365.

[Par93] “The discrete charm of ValueModels”, *in* ParcNotices, ParcPlace Systems, Vol 4, No 2, Summer 1993, Sunnyvale, CA, pp.1, 8-9.

[PV97] Marc-Antoine Parent and Louis Vroomen, “Giza: A framework for visualization.” *presented at* Graphical User Interfaces for Hierarchies: A Workshop, University of Maryland, College Park, Maryland, Dec.97, http://www.crim.ca/~vroomen/workshop/slideshows/giza.htm

[SBK86] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kanh, “Integrating Access-Oriented Programming into a Multiparadigm Environment.”, IEEE Software, January 1986, pp.10-20

[ShS89] J.J. Shilling and P.F. Sweeney, “Three steps to logical views: Extending the object-oriented paradigm,” *in* Conf. Object-Oriented Programming Systems, Languages, and Applications, New Orleans, LA, ACM, 1989, pp. 353-361.