Dynamic Attributes, Code Generation and the IUE

Terrance E. Boult, Samuel D. Fenster and Jason W. Kim*
Electrical Engineering and Computer Science Department, Packard Laboratory,
19 Memorial Drive West, Lehigh University, Bethlehem, PA 18015.
and
Center for Research in Intelligent Systems, Department of Computer Science,
Columbia University, New York, N.Y. 10027

Abstract

This papers reviews two software aspects of the IUE project: dynamic attributes and code-generation. Dynamic attributes separates inheritance of object state from inheritance of storage and provide more flexible object derivations. The primary motivation for dynamic attributes difficulties with using in existing C++ for what the IUE specification committee considered a natural design. After presenting basic background and motivation, the paper overviews the dynamic attributes mechanisms and the current C++ proto-type. It discusses how dynamic attributes provide more flexible access, cleaner class structure, and greatly reduced storage constraints, while still maintaining time efficient data access. It also discusses other benefits of the dynamic attributes mechanism including derivations with subtyping, run-time attributes and properly scoped polymorphic variables. This is followed by a summary of an initial performance study of the second proto-type which is now incorporated into the IUE development process.

A TEX-based code generator is then described. This generator supports the dynamic attributes mechanism, provides the needed support for the IUE metaclass objects and ensures (syntactic) consistency between the specification documentation and the developed code. This facility mechanism has produced over 100K of working IUE code directly from the IUE specification documents.

*This work supported in part by ARPA contract DACA-76-92-C-007 and by NSF PYI award #IRI-90-57951. The work reported herein was performed at Columbia Univ. T. Boult and J.W. Kim are full time at Lehigh. S. Fenster is at Columbia. The views presented herein are those of the the authors and may not reflect those of the IUE design committee, AAI, ARPA or the government.

1 Introduction and Background

The image understanding domain places strong demands on the underlying software, and it was determined, *a priori* that the IUE should handle:

- Large images: 10–30 images of size 4K$\times$ 4K
- Sequences: 200–500 512$\times$512 images
- Many features: 10,000–30,000 edgels per image

Because the system was intended for research and development in various domains attaining reasonable efficiency by requiring pre-filtering or using only small windows of data was considered unacceptable. It was prescribed that the system would use of C++ and CLOS[1] so solutions requiring other languages, such as Smalltalk, Eiffel or Self, were also not acceptable.

The committee started off with a mathematical orientation to its hierarchy design. Objects were endowed with properties/behaviors important to various categories of IU algorithms; and a conceptually "clean" but deep hierarchy of well endowed objects began to emerge. When faced with choices between flexibility/extensibility and performance the committee generally, though not always, side-stepped performance issues. Additionally, the natural clash between CLOS and C++ kept the committee from getting into low-level details too early. The committees design philosophy was leading, however, toward a design that would be highly inefficient. E.g. if the original design were directly implementable in C++ or CLOS a 2D-line-segment would have required more than ten thousand percent (1000+%) more storage than for a simple representation! With a requirement to handle 10,000-20,000 lines that was

[1] The status of CLOS development has subsequently been put on hold

unacceptable. So too was treating a handful of "common" object in a very special manner. The committee was faced with a difficult choice: greatly change its view of the "natural" hierarchy of IUE objects, or suffer staggering performance penalties. Both alternatives seemed unacceptable and we proposed and developed dynamic attributes as a potential solution. After a first proto-type proved the concept and performance, the solution was eventually approved by the committee and is now an integral part of the IUE design.

With dynamic attributes a general 2d-line-segment only takes only as much space as you would expect, even though it is conceptually well endowed. Special forms of 2d-parametric-line-segments, say with coordinates restricted to pixels in an 8K×8K image, could be defined using as little as twelve (12) bytes even though the user can access/change the four coordinates of its endpoints, the "name" of the segment, any annotations associated with it, the coordinate system in which the line-segment exists, the line-segment's drawing color, its drawing style, the name of each end-point, the drawing color of each end-point, the drawing style of the each endpoint, etc.. In addition a user can access (but not change) the line segment's slope, its bounding-box, its length, its tangent, its normal... (As will be explained later, modifying anything other than the endpoints, however, will consume additional space.)

To summarize the premise of of dynamic attributes is **inherit state, not structure!**. The design goals are to:

Goal #1 Provide access to an object's state without fixing its structure and delay the space/time tradeoffs until *after* the core IUE system is finished. If an IUE user defines a new subclass with more fixed state or storage, the existing IUE code should work and should have faster state access!

Goal #2 Minimize performance penalty with respect to direct data member access.

Goal #3 Increase *flexibility* of state access, e.g. provide run-time attributes and compute-on-demand attributes. (Sensors need them!)

The next section discusses background on object-oriented programming and why storage problems arise. Section 3 discusses the constraint-based view of inheritance and how it relates to the IUE. In section 4 attributes are defined and issues of mutation, state access and the various forms of dynamic attributes are discussed. Section 5 presents performance analysis of the existing system and is followed by a section on implementation details. Section 7 discusses the TEX-based code generator.

2 Object, inheritance, and types

This section discusses background on objects and object-oriented programming and how they related to dynamic attributes. We will use IUE-like examples (see figure 1) to help keep things concrete. We are defining terminology to be used in later sections and setting up some motivation for the dynamic attributes systems.

Before we discuss object technology, we need to define a few commonly used software terms. (In doing so, we will use the term *object*, which we will define momentarily.) These definitions, however, transcend object-oriented programming and apply to other paradigms such as functional or logic programming.

Abstraction: The essential characteristics of an object that distinguish it from all other kinds of object. (Note that this *must* be relative to the perspective of the viewer!)

Hierarchy: A rank or partial ordering of abstractions. The abstraction of a related set of things define a *class*; subclasses are elements of the abstraction hierarchy that refine the abstraction. The IUE objects form proper abstraction hierarchies, e.g. a circle *is-a* conic *is-a* curve *is-a* spatial-object *is-a* set (actually a point-set).

Typing: Preventing non-equivalent software objects from being exchanged. *Type hierarchies* provide a restricted form of exchange within the partial order. A *subtype* of a type satisfies the "Liskov substitution principle"[Coplien-1992]: It can be used anywhere the supertype could be used.

Inheritance: The use of an abstraction hierarchy to define shared aspects of classes of objects. While abstraction hierarchy provides semantic sharing, inheritance can provide "interface" sharing, code sharing and/or representation sharing. A class of objects A that inherits from another class B is said to be *derived* from it. A is called a *base class* of B.

Encapsulation: Hiding all non-essential details. Of course, this definition depends strongly on what is considered non-essential. Some software experts take a strong view, e.g. Ingalls [1978] suggests that "No part of a complex system should

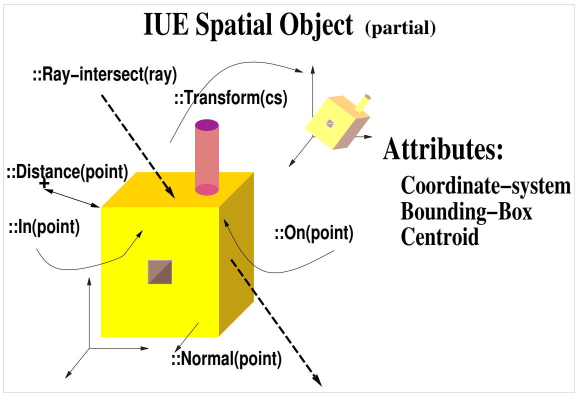


Figure 1: *A spatial object with its state attributes and some behaviors.*

> depend on the internal details of any other part." By using only the "interface" to objects, rather than representation details, we obtain encapsulation. If a language does not allow access to encapsulated details, they are called *private*, in which case things visible in the interface are *public*.

As we shall see, there are often tradeoffs to be made between encapsulation and inheritance. Inheritance can allow the sharing of internal details for efficiency (and code sharing) but this violates strict encapsulation.

Uses of inheritance in object oriented programming include: *sharing interfaces*, *subtyping*, *code sharing*, and *data sharing*.

Polymorphism is a concept from type theory in which a name (variable) (or a pointer/reference to a variable, as in C++) can refer to an instance of any type derived from a particular type (or that satisfies a particular type predicate). E.g. a polymorphic image addition method would accept any image subtype as input and do the proper operation. In conventional languages, such as C, is accomplished via a switch/case statement. In object oriented languages it can be accomplished using operator overloading and virtual dispatching, wherein multiple versions of a routine such as "+" exist, and the run-time type of the instance is used to determine which version is actually called. Polymorphism allows very general operations by deciding which particular implementation of an interface behavior ends up being used. In addition, a polymorphic operator need not be modified when a new subtype is added to the system.

Recall the basic principle that an object's class is embodied as *interface and behavior*. Since polymorphism requires *consistent syntax and semantics* in the interface/behavior of a subclass, then subclasses should be subtypes. Thus, our object hierarchy has very general objects at the root, and more specialized objects as we progress deeper. As we move deeper, the we can add to the set of states and behaviors associated with a class but we cannot remove them. Note here that to understand the semantics of a derived class, one must understand the semantics of all the things from which it was derived.

Let us now formally define objects and classes. There are many definitions with slight technical variation, but a typical one, given in [Booch-1991, 77-84], is: *"An* **object** *has state, behavior and identity."*

Where the terms used in the definition are defined as:

"**State** *encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties."* For an IUE-spatial-object, say house1, its state includes its coordinate-system, bounding-box and centroid. We will use the term *attribute* to refer to a property of an object without specifying how that property is stored.

"**Behavior** *is how an object acts and reacts, in terms of its state changes and message passing [method or procedure calls].* For example, in the IUE-spatial-object these include the "member functions" (methods) house1.in(spatial-object), house1.distance(point), house.transform(cs2), ... Behaviors are invoked through an interface, which has both public parts (behaviors anyone can invoke) and private parts (behaviors restricted to use by instances themselves).

"**Identity** *is the property of an object which distinguishes it from all other objects."* (It is *not* equivalent to a variable or variable name!). Generally associated with identity is an operation to test if two object have the same identity, say &house1 == &house2 or (eq house1 house2). Note that identity is different from object equality, which can be defined in terms of equal states. The latter is generally written (house1 == house2) or (equal house1 house2).

"An object **[instance]** *is a concrete entity that exists in space and time; a* class *represents only an abstraction, the* 'essence' *of an object as it were."*

Finally Booch defines a class, saying [Booch-1991, p93]: *"A* **class** *is a set of objects that share a common* *structure and a common behavior."*

Do you notice anything odd about this definition? Look back at the definition of an object: state, behavior, identity. Why then does the definition of a class say they share a common "structure" rather than a common "abstract" state? The answer is one of convenience. Booch goes on to say the following:

"The state of an object must have some representation, which is typically expressed as constant and variable declarations placed in (the private part of) a class' interface...."

"The careful reader may be wondering why the representation of an object is part of the interface of a class, not of its implementation. **The reason is one of practicality***; to do otherwise requires either object-oriented hardware or* **very sophisticated complier technology***."* [Booch-1991, p95, emphasis added].

As we shall see, the Dynamic Attributes project is a first step to providing the needed compiler technology to allow the separation of the interface to an object's state from the internal representation of that state. We have not (yet) actually changed the compiler. Rather the current DA mechanism uses existing language features, supported by our code-generator, to provide new semantics, which allows efficient separation of the access to state "attributes" from their internal structure and storage.

2.1 State $\neq$ Storage Layout

There are really many issues that underly the difficulty of separating the implementation from the abstract state. The most demanding of these is the interplay of inheritance and polymorphic operators.

The code sharing aspects of inheritance in existing systems dictate that a behavior defined in one class is applied to objects of subclasses unless a more specialized version is supplied. At first inspection, this seemingly implies that as we move deeper, the internal representation of a class must, at the very least, replicate the storage of its superclasses. After all, we want (need) existing code to operate correctly, even if the object to which it is applied is a derived class.

But if we *efficiently encapsulate* state access, neither the class implementation nor the rest of the world cares where, or even if, things are stored!

This leads us to a very important question: *Should a behavior (method) be allowed to violate the encapsulation of state in its associated class?* Should the right hand know how the left hand represents things? To which we (and many texts on object-oriented programming/style) answer a resounding "NO!".

It is becoming common for "state variables" to be private and for all functions, including those of the associated class, to access them via (inline) functions (*accessors*). Then changes to the representation of an object, e.g. changing an array from storing size and initial index (and hence computing final index) to storing initial and final index (and computing size when needed) can be completely localized. With inline functions, as in C++, using accessors increase encapsulation but *does not compromise program speed*. They may, however, slightly slow down compilation.

It might seem that for efficiency, violation of encapsulation is needed when a single "access" to the state is actually a sequence of accesses. However, such a sequence can be efficiently encapsulated via an *iterator*, an interface function designed for sequential access to objects with a large internal state. In addition to the protection offered via encapsulation, the iterator interface provides an abstraction, often making it easier to read the program.

3 The Constraint-Based View of Inheritance

There are many "views" one could take when building an object hierarchy. In this section we start with a classic example which demonstrates some of the problems to be faced. Then we discuss the issues which arise in the IUE design and its use of a constraint-based view of inheritance.

3.1 The Circle and Ellipse Problem

To exemplify some of the issues that arise in hierarchy design, let us consider how one would design a hierarchy with two shapes, a circle and an ellipse (see figure 2). A circle can be specified with a center and a radius. The ellipse has many representations, one of which is a centroid, a major radius, an eccentricity and orientation defined as the angle between the y axis and the axis of the major radius. Both classes might include attributes such as drawing color, name, etc..

We might start with the mathematical point of view that every circle is an special case of an ellipse, and

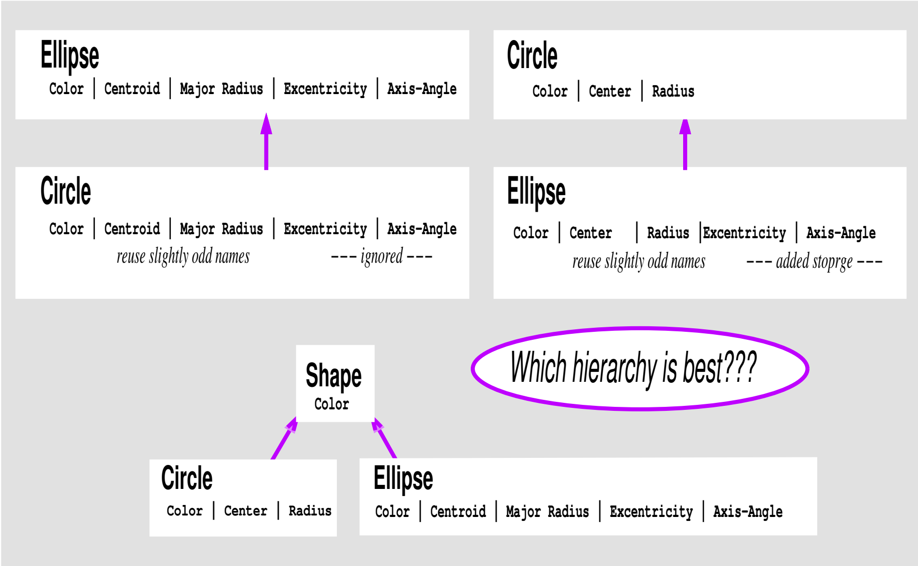


Figure 2: *Three hierarchy choices for the "classic" circle and ellipse problem.*

make circles subclasses of ellipses. Code which used the ellipse's properties in a read-only manner would naturally apply to the circle, e.g. a drawing routine need only be defined only for ellipses. Thus we could benefit from code reuse. However, the representation of a circle would require twice as much space as is really necessary. Even worse, someone could modify the "eccentricity" thus violating the definition of a circle! Obviously this design cannot be used without further hackery.

Still we would like to share code between circles and ellipses, hence one might think choose to view the ellipse as an extended circle? In this way the "constraints" on the circle could not be violated (they are not represented) and the objects are represented in a space efficient manner. Unfortunately, this give center and especially radius a slightly different "semantic" interpretation. Furthermore the shared state is not sufficient for many routines, e.g. for drawing, so code sharing would be limited. More problematic would be behaviors, either local to circle or of other objects, which make use of mathematical properties of circles such as rotational invariance, constant curvature, etc.. The problem arises because this hierarchy violates the Liskov substitution principle—an ellipse is not a subtype of circle. While we are unaware of the use of a design with ellipse derived from circle, similar construction are not uncommon, e.g. in [Pokorny-1994], a graphics text using C++, 4th order B-splines (ub4curve) are derived from third order B-splines (uc3curve), and quadrangle is-a triangle?

Thus is often drawn to the "sibling" view, wherein a superclass shape os defined with minimal semantics and only ancillary attributes such as color. Then and both ellipse and circle derived, as siblings, from the shape class. Definition of interface functions, such as drawing, would be specified at the shape class (as pure virtual functions), and a separate implementation supplied for both circle and ellipse. This design provides consistent interface semantics, but little or no code sharing. It also permits code to make proper use of mathematical properties and invariants at the circle and ellipse level.

We have discussed three possible hierarchies for this classic problem. Which one is the "best" hierarchy? This has persisted as a classic problem because the answer, of course, depends on your point of view. Remember the hierarchy simply codifies someone's view of a sequence of abstractions. However, one person's abstraction can be another's abomination. (We prefer a mathematical view, but delay our proposed solution until section 4.5.)

The issues are thus tradeoffs between semantics, space and code reuse. The ideal would a system with clean semantics, efficient space utilization and extensive opportunities for code reuse. Before we see how dynamic attributes will provide these benefits, let us discuss the constraint-based class derivations used in the IUE and their ramifications.

3.2 Constraint-based derivations

When the committee was designing the IUE hierarchy, we tried to maintain a mathematical orientation in the design. Some of the specializations were obtained by adding additional state to an existing object, e.g. a parametric line is a specialized line where we add a parametric representation. This style, specialization through extension, is well supported in C++ and other object oriented languages; a derived class inherits all the state of the base class and can extend the state representation. Let us therefore refer to this specialization through extension as the traditional view of inheritance. In many instances this is a natural view of inheritance.

However, in the IUE design, as if often the case in mathematics, the sequence of going from a general "type" to a more specialized type was often viewed as constraining some aspect of an objects state, e.g. a 2D line is a line where the coordinate space in which it is embedded is restricted to be a 2 dimensional. Here the constraint is to restrict a state variable to a subtype of its original formal type. While such derivations are mathematically quite natural, C++ does not permit such refinements on the representation of an objects state. In fact, very few strongly typed object oriented languages do.

A second class of constraints is to constrain a parameter to have a particular value, e.g. the eccentricity of a circle is always 1. In this case the parameter of interest need not even be represented, though of course the object could, if asked, report the value of that parameter. But any operation on the superclass which tried to change the parameter would fail on the subclass. Thus, the subclass would fail to be a subtype; it would violate the substitution principle, which is perhaps the most useful aspect of inheritance: A routine expecting an ellipse could not be given a circle, lest the routine try to change a constrained, or nonexistent, part of the circle's state. Thus while the language allows such derivation by constraint, it does not really support it.

A third class of constraints is constraints between parameters, e.g. an isosceles triangle is restricted to having two sides of the same length. Once again using the constrained parameters is easy, but modifying them can be problematic. If these parameters are accessed directly (as C++ data-members), there is no way to insure the constraint. Again C++ allows, but does not provide significant support for, this constraint-based view of inheritance. And again, a derived class that could often use less space must retain all of its superclass' storage, and hence wastes space.

One may may even consider the specialization through extension as a form of constraint-based derivation. Let us consider a class extending the state representation with a variable $s1$ of type $t1$. In this case the base class does not restrict $s1$ in any way, not even its restricting its type. The derived class has constrained the state to require $s1$ to be of type $t1$. If one consider the be a root type, such as `T` is in CLOS, from which all types derive, then we can consider the specialization through extension as a form of constraint by subtyping. As we shall see, this view is supported by the dynamic attributes mechanism.

What is needed to support the constraint-based view of inheritance is the ability to "subtype" a state variable during subclassing, the ability to efficiently separate state storage from state accessibility and the ability to enforce constraints between state variables. Dynamic attributes will provide support for these.

4 Attributes

In section 2 we said an object was defined by its state, behavior and identity. The definition of state included the static "properties" plus the value of each of these properties; it did not not discriminate between these two important, but different, aspects of the object. The issue becomes important when we define behaviors in terms of their state changes. Do we want the behaviors to change which properties are associated with an object? If so what are the semantics of the added properties, and what are their interfaces? What does it mean to remove a property?

We define an *attribute* of an object to be the name and semantics associated with an *abstraction of a particular property of an object*; an attribute thus gives a name and semantics to a property of a object. An object has a set of attributes, and we reserve the word state to refer to set of values these attributes assume at a given point in time. This definition allows state-changes to be restricted to changes in the values rather than potentially changing what is in the set of values. The state-space of an object is the set of combinations of values that its (fixed) set of attributes may assume. We point out that the "attributes" of an object are not tied to its representation in storage. Rather, attributes are "properties" of the object's state—independent of its representation. For example, we might say that an array has the properties of initial-index, final-index, size, indices-used, and data, or that a circle has attributes of center, radius, diameter, size, curvature, eccentricity, etc. Note that from a representational point of view this set of attributes may be overcomplete—one is not free to independently specify a value for each of these properties. However, they can all be queried and some can be modified (though the semantics of updating, say size, would need to specified). We will come back to the issue of modifying attributes in the next section. Let us now consider they the distinction between static attributes and dynamic attributes.

Static attributes are attributes of an object that were declared *a priori*; in the IUE, static attributes are declared in the IUE documentation, and hence known to the compiler. The semantics and type information for such attributes are known *a priori* and allows automatic generation of more efficient access functions and allows constraints to be enforced between static attributes.

However, it is common in a research system to want to add additional attributes to an object while processing data. In CLOS, this a common use of the property list. While it would not be difficult to implement a property list for the IUE, it has the drawback that a user/programmer must know if the attribute they wish to use is a static attribute or one on the property list. While this might seem only a minor issue, in systems that support polymorphic operations it becomes more subtle; an attribute not declared in the base class would be stored on a property list, but it might be directly supported in a derived class. Developing modules that test the object type to determine where the "attribute" is stored (property list or object) violates the ideas of encapsulation.

The idea of dynamic attributes is to define a single space of attributes with a consistent access interface. There are general access methods that can access attributes whether or not they were declared as part of a class' definition; predeclared attributes provide specialized accessors. We allow the same attribute name to be used independently in unrelated classes. Within a class, name clashes for predeclared attributes are detected at compile time. Run-time attributes that are "defined" on first use; the first access causes, on a per instance basis, the creation of the attribute. Subsequent access refers to the existing attribute. Thus we effectively define, on a per instance basis, a unique name space of attributes. Since definitions in this name-space can be made by different programmers/users, we provide a means for testing if a name is already in use.

Dynamic attributes provide a uniform mechanism for accessing object state. By now the C++-oriented reader will probably be wondering about the type safety of such a uniform access mechanism. The implementation details of this will be discussed in section 6, but the short answer is that the mechanism is run-time type safe and all accesses to statically defined attributes via code-generated access functions are compile-time type safe. In fact, the system provides additional safety in that we separate read access from modification access, hence preventing certain classes of errors.

Given our view of attributes, an interesting (and non-trivial) question is, "What is the difference between attribute 'accessors' and other behaviors?" In the pedantic view, all attribute accessors (even regular data member access in C++) can be viewed as behaviors. But which behaviors can be viewed as attributes? Attribute read access does not need auxiliary information (i.e., it can be a member function of 0 arguments and no global program data); write access requires only one argument (the new value), and must affect the state such that the value persists. Any argument-free behavior with no side effects can be made into a readable attribute

The issue of whether to compute redundant information vs. store it is a well known problem for designers. The use of dynamic attributes allows this decision to be easily delayed during the design. Because of the uniform interface, the choice can be easily reversed later, or various combinations supported in specialized subclass. As we shall see, dynamic attributes even allow the end users of the class-hierarchy to derive a new subclass with different choices and still have existing code function properly. Furthermore, the existing code will have performance almost equivalent to that obtainable if the user's compute-vs.-store choices had been made by the system designers.

Thus the second distinction between an attribute and behavior is that, in the opinion of the system designers, a derived class may be defined for which would be effective to store the associated property. If a writable (mutable) attribute is part of a object constraint, it is critical to define the semantics of side-effects of updates to any dependent parameter.

The decision of attribute vs. behavior requires foresight on the part of the designer. The only down sides to making an argument-free behavior into an attribute is potential name collisions with other attributes and the need to define the semantics of updates to mutable attributes. Therefore, the designer should err on the side of flexibility, especially if the behavior is free of side-effects.

4.1 Mutability vs. Utility

One of the issues that continually causes difficulty in the design of an inheritance hierarchy is an inherited attribute's utility vs. its mutability; i.e., the distinction between the subclass' ability to meaningfully use the superclass' attribute and its ability to change the value. This section discusses this issue and our solution to it. We refer to attributes which are used by a class, but never modified (even indirectly), as *read-only* attributes. We note that this is a view of the attribute from a particular class' point of view; if we have a pointer to a polymorphic object the actual instance may or may not support modification.

The real issue underlying utility vs. mutability is how much is being revealed about the attribute through the interface that supports its use or change. Utility is providing read access to a state attribute and hence exports little knowledge about the underlying object—just that it can compute the associated state variable. Changing state, however, is a considerable more complex behavior, requiring knowledge that the representation of the attribute is sufficiently powerful to accept the change. Such mutability also requires significantly more semantics, especially when the state of the object is over-parameterized. For example, if one changes the size of an array how does that change effect the index set?

A major reason for the dichotomy between utility and mutability has to do with the general nature of subtyping (or *is-a* inheritance). A constrained form of a class is still an element of that class so, from a utility point of view, it satisfies the type (or is-a) constraints. However, the constrained form of the object cannot support direct modification of the constrained aspects of its state, and hence does not satisfy the mutability aspect of is-a.

A mathematical object cannot change; rather we are asking for a new object just like the old object but where one (or more) attributes have new values. The mutation operation on is really a shorthand for requesting a new object, deleting the old object and assigning the new object to the "name" associated with the old object.

Taking this view, we can change the eccentricity of a circle (given semantics for that change), but the result will be, of course, an ellipse. Thus the change cannot be done in place, nor through a polymorphic reference. For the general ellipse class, we can define behaviors that return a "new" ellipse with the requested modification. We can also provide a subclass of ellipse, mutable-ellipse, which provide in-place modification to important "ellipse" parameters. If there are multiple important mutable forms, e.g. fixed eccentricity but mutable scale, they can be implemented as difference mutable subclasses of ellipse.

One might note that the above idea, splitting the hierarchy based on mutability, is really the same as splitting the hierarchy any time there is a significantly complex behavior that discriminates between important categories of instances. In our view of abstraction, the ability to modify an object in place is often such an operation. While the combinatoric possibilities of subclasses with all combinations of mutable/immutable attributes would completely swamp the hierarchy, most of these are not meaningful sets. Determining, however, what parameters to label mutable vs. immutable depends on the abstractions being used to build the hierarchy, in particular the abstractions to be used *deeper* in the hierarchy. For example, object color is probably not important deeper in the spatial object hierarchy so it can be left mutable at the highest levels; dimension of the coordinate system is expected to be constrained and hence should be read-only at the highest levels. As in the choice between attribute and behavior we rely on the foresight of the designer to anticipate the needs of users and make appropriate utility vs. mutability design choices.

4.2 Access functions

In the dynamic attribute mechanism, we have actually separated utility/mutability into three forms of access: read (get), write (put) and reference (ref) access. The first two should be clear. The last case, ref access, returns to the user a memory reference where the user can directly access the object which is the attribute's value. Note that providing ref access export far more information about the underlying attribute—now its "storage" format is publicly known. We note that a secondary advantage to this decomposition, which we are just beginning to explore, is that in a multi-processor or multi-threaded environment we have now decomposed read and write operations to provide greater opportunities for concurrent access.

Let us finish this section with an example of each accessor. Assume an object ob with an integer attribute slot1 and a line_t attribute, line1, which itself has attributes point1 and point2, each of which in turn have attributes x and y of type double. The syntax for the attribute accessors of object ob look like:

```
int i; double u; line_t myline;
i = ob.slot1();
ob.ref_slot1() = i;
ob.put_slot1(i);
myline = ob.line1();
u = ob.line1().point2().x();
ob.ref_line1() = myline;
ob.ref_line1().ref_point2().ref_x() = u;
ob.line1().point2().put_x(u);
```

And the "generic" accessors:

```
IUE_value myval;
myval = ob.get("line1");
ob.ref("line1",DAtype<line_t>::typeof()) = myval;
ob.put("line1", myval);
```

Note that the generic accessors read and write polymorphic IUE_values, which will be described in section 6. The data in these run-time-typed objects is accessed in a type-safe manner. E.g., if line1 had a type line_t then it would be accessed as either of:

```
myline = DAtype<line_t>::get(myval);
myline = DAtype<line_t>::get(ob, "line1")
```

4.3 Forms of Attributes

The previous section gave the flavor of the consistent attribute interface. The user interface does not determine the internal representation of the object—dynamic attributes support various "forms" of attributes which provide great flexibility in terms of time/space tradeoffs. We currently support seven basic forms: **hard, firm, soft, static, read-only, hook** and **generic**. We use the term *covering* to refer to the process of overshadowing a superclass' definition of an attribute with another. Covering can be used to change the form of an attribute or to subtype non-hard attributes. We briefly describe both the forms of attribute and the kinds of covering.

Hard attributes are equivalent to C++ data members, in fact, they are internally implemented as such. A hard attribute always takes the full attribute space in each instance (maximum space), but provides the fastest access. Creation/deletion of object instances are affected slightly. Access to a hard attribute is, when possible, expanded inline, and otherwise done through a virtual dispatch. Hard slots cannot be covered—once hard they stay hard and their type cannot change. This form is intended for attributes which demand high-speed access and for classes where most instances will change the attribute from its default.

Soft attributes do not take up space in the object. Each attribute takes on a run-time default (discussed below) until such time as the user changes it. If the attribute is never changed from the default, it takes no space. If the value is changed it is stored in the class' object-level context (a multi-level extensible hash-table). As can been seen in the performance charts (table 1), there is a time time/space tradeoff. The access to this attribute will be "virtual" and supports full covering with new forms. This form is intended for mutable attributes for which, at this inheritance level, access is not time-critical.

Firm attributes are a recent addition to our design to provide even more time/space tradeoff flexibility; firm attributes occupy a time/space point between hard and soft attributes. If they are assuming the default value firm attributes take up only a pointer's worth of space in each instance. When changed from the default they take up one pointer plus the real size of the attribute. Again, there is a space/time tradeoff (table 1). Access through a pointer requires a virtual dispatch, and direct accesses require a single pointer dereference. Unlike hard slots, which cannot be covered at all, firm attributes can be covered by subclasses. They are intended for speed critical attributes which will undergo subtyping constraints in derived classes or for attributes which will be frequently accessed but changed from the default in at only a moderate number of instances.

Static attributes these are attributes for which there is a single instance shared amount the entire class. Like hard and firm attributes they are very efficient. Like firm attributes they can be covered with subtypes. They can also be covered with any other forms (hard, soft...) which will then allow them to vary per instance.

Read-Only (immutable) attributes these are attributes which, from the associated class' point of view, are read-only. If derived classes permit modifications, the attribute can change value while the user is not looking, so it is not really immutable. The attribute will usually be be implemented as a hook function with no associated storage, or as a hook that uses soft storage which can be changed from the default only at creation time.

Hook attributes are the term used to refer to attributes that, for the current class, are computed rather than stored. They are important for maintaining constraints in constraint-based derivation. Read-only attributes are implemented using hooks with only a get-hook supplied. Put hooks can also be supplied, but their semantics must be clearly documented because a put-hook can have side effects that change the value of other attributes. We do not, currently, support general ref hooks, though they may be part of a future implementation.[2] Hook attributes can be covered by either soft or hard attributes. A hook without reference methods cannot be used to cover a soft or hard attribute. A special type of hook, the caching hook, is now being added. This hook function supports get, put, ref and reset, with the semantics that the first time it is accessed, its value is computed and stored. Subsequent puts and refs access the stored version, and reset forces the next get to recompute the value. (We note that the actual implementation is a combination of a hook attribute with a firm attribute). We are providing the cached-hook because the compute-if-used idiom is anticipated to be frequently used within the IUE.

Hardened attribute is the term we used to refer to an attribute that is a soft or hook attribute at some level of the hierarchy, but which is hard in a subclass. The access to this covered attribute can not always be direct; only functions which declare objects, references or pointers of the hardening class, or derivations thereof, will have direct access. When access is through an ancestorial polymorphic pointer or reference (as it is in member functions of classes "above" the hardening), the attribute will be accessed through a virtual dispatch yielding performance close to that of the hard attributes. This provides an important flexibility. For example, the system designers can develop code using "soft" attributes, say to render the scene using an infrared sensor. Each spatial object can be given a soft attribute for thermal emissivity and other related attributes. The "IR-rendering" will operate properly (using defaults or values stored in the soft attribute). If, however, users need more efficient rendering of a few types of spatial objects, they can subclass, covering the soft attributes with hard attributes and *vóila* the existing code runs 10-100 times faster! In such cases the user need not develop "code" at all—they simply document the subclasses of interest, use the code-generator and recompile. If they really need to eke our the absolute highest performance, they can specialize the rendering method for their objects and regain regular hard attribute (data member) performance.

Firmed attribute is the term we used to refer to an attribute that was a soft or hook attribute and is converted to firm in a deeper class. It is between hardened and firm in access performance.

Run-time attribute is the term used to refer to an attribute that is defined at run-time. The access is considerably slower but provides important flexibility. If the name of the attribute requested coincides with a preexisting attribute, the access functions of that preexisting attribute are used. This mechanism is intended to support rapid prototyping and flexible interface functions. It is useful for rapid prototyping because it allows temporary "extensions" to an object without modifying its definition. E.g. programmers developing temporal reasoning systems could use this to add their own "time-stamp" field to any IUE object. For important subclasses the time-stamp could be added as a predefined attribute, and the existing code would gain in performance. See table 1 for some performance examples. For interface development it allows run-time textual user input to be used to access any attribute of any object in the system for either display or (if the attribute supports it) for modification. Thus the system-supplied interface will directly be able to support user-defined classes in a consistent manner.

4.4 Default values

The IUE requires default values for all attributes. In the IUE-specification each attribute is allowed to define a default value. There are also default defaults based on the attribute types, e.g. 0 is currently the default value for integer attributes which do not supply a default in the specification. For more complex type, i.e. class instances, the "nil" object of that type is the default default. The code generation facility insures a valid "nil" for each type specified.

[2] It is not always meaningful to provide a ref-hook function because there may be no storage associated with the attribute.

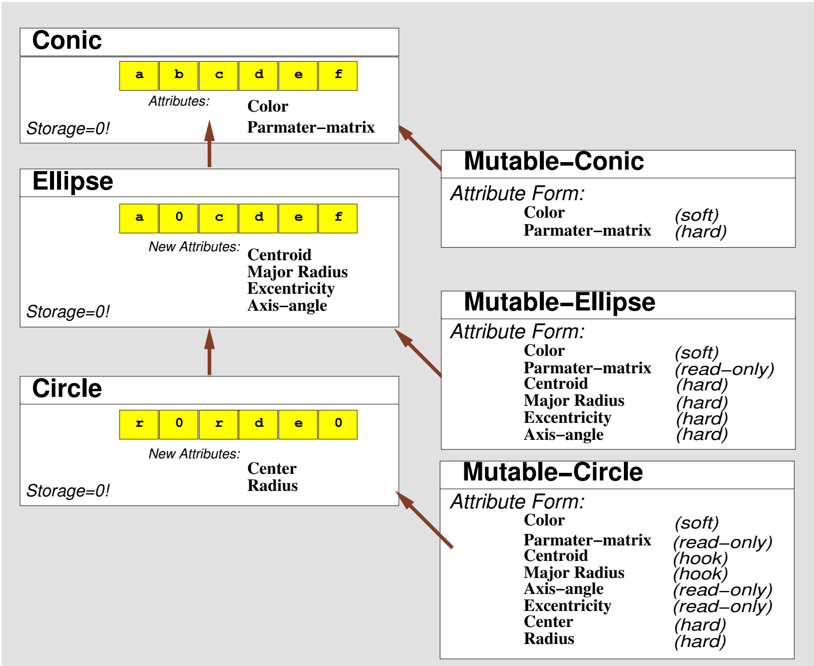


Figure 3: *The dynamic attributes solution for the classic circle and ellipse problem. See section4.5 for explanation.*

The use of defaults insures that returned values, even on error, are syntactically and functionally valid. The defaults for each attribute are stored in a *context object* associated with each class. As discussed in section 6.1, a run-time search algorithm allows defaults for an attribute to be stored in other locations as well.

4.5 New Twist on the Classic Example

Now that we have introduced the dynamic attributes mechanisms, we can revisit our old friend, the circle and ellipse problem. The new solution is consistent with the mathematical, or constraint-based, view. To show a little more generality we included the abstract class conic as a superclass of ellipse, see figure 3.

The main trunk of the hierarchy shows the new attributes added at each level. The attributes defined are available to each derived class. The color attribute (and any other attributes unrelated to the constraints used in the local hierarchy) are soft attributes. The others would be hook attributes, or, more likely, read-only attributes. For each of the main classes we derive a mutable form. While only one is shown, multiple mutable forms might exists corresponding to different internal representations. The form of each attribute is shown. The hook functions for centroid and major-radius in the mutable circle would simply access the obvious circle attributes and could be inlined, providing data member access speeds. The parameter matrix would not be stored, but computed when requested. If used frequently it would be better to make it a cached hook. Note that the storage for either an (immutable) ellipse, or (immutable) circle, if they take on the defaults for all attributes, is zero!

5 Performance Analysis

This section discusses the performance of the existing dynamic-attributes prototype.

The timings were obtained on a SUN SPARC630 running 4.1.3 using Lucid C++ compiler lcc version 3.1b7 with an optimization level of -05. Each timing uses 100 instances of a class accessed via an array of pointers. Each loop adds the recovered/stored value to a variable of the appropriate type to insure that the optimization does not remove the access from the loop. Hard, Firm and Raw C++ times were obtained using loops over 100000 accesses on each of the 100 instances. Soft and generic access used 1000 interactions on the 100 instances. Times were obtained by sampling the system clock before and after a loop of accesses, and hence “system time” is included.

The main performance figures are presented in table 1. As can be seen, hard attributes have achieved our performance goals, and the performance of hardened attributes shows the performance vs. flexibility tradeoff is clearly a good choice. The firm attribute access was also quite good, nearly equaling hard attribute access on reads. Because of the additional checks needed (to see if it is the default), firm puts and refs are about 50-70% slower than hard puts and refs.

The access times for soft attributes are 2-3 orders of magnitude slower, but are comparable to non-optimized access times of "slots" in CLOS. A number of performance enhancements have already been added since the first dynamic attribute proto-type (where get times were on the order of 200-300 μsec.) It is expected that another 20-30% performace gain is obtainable on gets and puts/refs to existing storage. The very large times for puts/refs to soft attributes with defaults is because of storage allocation and the need for the extensible hash tables to grow (which copies over elements). The hashing aspects of the system have not been optimized.

Note the timings for the generic access mechanism and for soft attributes are dependent on length of the attribute name because of the hashing. The names used for the timing were short (2-3 characters).

The above timing results are preliminary; a few unexplained behaviors are being investigated. One such anomaly is that soft access to doubles was (consistently) faster than soft access to integers. This is likely related to register allocation and may not hold up in complex computations. We have yet to determine why the hardened attribute access was faster than the virtual C++ access, though it may have to do with visibility issues during optimization.[3]

6 Some implementation details

Underlying the implementation of dynamic attributes are five main elements: the IUE meta-class objects, IUE_contexts, IUE_values, the DAtype mechanism and code generation. We very briefly touch on these elements.

Early in the IUE design it was decided to have a meta-class object. The dynamic attribute mechanism uses this meta-class object to maintain tables about the attributes, their form and their defaults. We note that these tables were desired for other reasons as well and thus this did not significantly impact the system building requirements.

Within the meta-class object are two objects of type IUE_context. One is the object-level-context, which is an extensible 2 level hash table (key: object ID then attribute name) used to store the soft attribute values. There is also a default-level-context which is a single-level extensible hash table (key: attribute name) which stores the default values for all attributes defined in that class. Unsuccessful seeks in the object-level-context then search the default-level-context. The default-level-contexts are linked together to mirror the class hierarchy. Unsuccessful seeks in a default-level-context proceed up the links until either a value is found or the search is exhausted. Contexts also play a major role in the implementation of IUE data-exchange.

Each context needs to be able to store any type of value. Thus we developed a polymorphic type, IUE_value, which can store any base type or any IUE type (or any external struct properly declared to the system).[4]

The polymorphic IUE_value also provides an important programming simplification because they can be used as a lexically scoped polymorphic variable that "cleans up after itself". During the initial DA design we felt this was a very important feature, though when the compilers support the ANSI defined covariant return it will be less critical. The will still be useful because covariant return uses pointers or references and hence places memory management burdens on the user while IUE_values automatically take care of this. They are also useful for generic containers. Finally, they play a vital role in implementation of the the IUE data exchange where the parser needs to read and create any IUE-type, even those declared after the IUE is distributed, and put them into a collection to be used for creation of other objects it reads from the file.

While run-time type information is coming to C++, its still not here (and was not in the proposed C++ specification when the IUE design began). Thus we developed a run-type system for dynamic attributes the

[3]We have found that lcc does not always inline a method when we would expect it should. This can interfere with the propagation of optimizations.

[4]In the initial design (and the current specification) and prototype IUE_values were more limited. Because of current issues of C++ "temporary" destruction, the current implementation is a bit more complex than we would like (LCC and G++ have not implemented the latest ANSI decisions on temporaries) and hence we have not upgraded the specification. When they the compilers better handle temporaries the design will again be sleek, but will still support all C++ types and all IUE objects.

Attribute Form	**Get**(int)	**Get**(double)	**Put**(int)	**Put** (double)	**Ref**(int)
Hard	.21	.26	.26	.47	.21
Hardened	.28	–	.28	–	.25
Firm	.26	–	.40	–	.37
Hardened Generic Access	19.20	–	28.58	–	NA
Soft(default)	65.00	46.60	1666.00	–	1666.00
Soft(existing)	59.90	–	73.30	–	65.00
Soft Generic access (existing)	60.00	–	75.00	–	NA
Raw C++ data member access	.21	.25	.25	.45	NA
Virtual Function C++ access	.52	.83	.57	1.43	NA

Table 1: *Table showing performance of various forms of dynamic attributes. See text for discussion. Entries with "–" were not computed; NA is* not applicable. *Writes to object with defaults require storage allocation (and hence larger times).*

IUE, which we call the DAtype systems. This templated set of classes/functions allows the type system to automatically support user extensions to the IUE. The type system supplies accessors to get, in a type safe manner, values into and out of IUE_values. It also provides a flexible mechanism for initialization of global type related data. In particular, it allows the IUE-specification to impose a order in which static data members will be initialized before main. Now if only the C++ committee has addressed that...

The final element underlying dynamic attributes is code generation which will be discussed in more detail in the following section.[5] We briefly describe, mostly by example, the nature of the code that is generated. Let us presume a class A with a soft attribute named p1 of type point. Assume a class B derives from A and hardens that attribute while subtyping it to be of type 2d-point.

Let us start with the functions generated for class B. The code generator provides the basic access functions:[6]

```
inline 2d-point const&
    B::p1( return hard_slot_called_p1;)
virtual inline 2d-point const&
    B::virtual_p1( return p1();)
```

In addition it generates a function B::table_get_p1 which is stored in the meta-class and provides type-safe access to the data member when accessed via the generic get mechanism. It also generates the appropriate declarations in the associated header files.

[5]While the Dynamic attributes implementation currently depends heavily on the code generation facility, an earlier version was heavily based on CPP macros. A macro or preprocessor based version will be again provided in the future.

[6]The example as presented presumes the new C++ covariant returns, otherwise we would have to declare point B::virtual_p1().

For the soft attribute of in class A the code generator provides:

```
virtual point const&
    A::virtual_p1(DAtype<point>::get(*this,"p1"));
inline point const&
    A::p1( return virtual_p1();)
```

where the function DAtype<point>::get is a call to the generic mechanism to retrieve the value from the object-level-context of the associated meta-class object.

Consider the following definition and function calls:

```
B b;
A a,*c=&b;   //declare an object a and pointer c.
a.p1();      //call p1 for a real A object
b.p1();      //call p1 for a real B object
c.p1();      //call p1 for a polymorph
```

The call a.p1() expands inline to a call to a.virtual_p1() which virtually dispatches to go look things up in the object-level-context. The call b.p1() expands inline to directly provide access to the appropriate data member. The call c.p1() expands inline to A::virtual_p1(c) which virtually dispatches to call B::p1(c) which returns the data member.

Note that the function B::p1() is *not* virtual and thus hides the function A::p1(). This hiding provides the inline access to the data member *if and only if* it is known that the data member must actually exist. Also note how it provides for covering with a subtype.

The same idiom, using a non-virtual access function which hides inherited versions, and redefining the virtual accessors provides the support for hook attributes and firm attributes; for both of forms special table functions are generated for generic attribute access. For read-only attributes we do not generated code for ref and put accessors and insure that the access via the generic mechanism does not modify the attribute.

The initial implementation, which was tested with C-front, lcc and g++2.58, did not support templated objects or non-iue structs but showed proof of concept and satisfactory initial performance. The new version, with full support for templated and non-iue structs, is approximately 17K lines of code (not including the code generator) and currently runs under lcc. It is being re-ported to G++, but differences in the way the compilers handle templates has made this non-trivial. While we are now supporting templates we have not had the opportunity to reimplement the full system using templates; only the type-definition, initialization, and access mechanisms are templated. We expect a fully templated version next year which should allow for increased performance on soft accesses.

6.1 Contexts and multi-key access

Another way that dynamic attributes differ from regular state access is that an the value of an attribute may be associated with a combination of objects. To put it another way, one may specify an object's state in the context of one or more other objects. Thus, an attribute may have different values in combination with different other objects. The "dictionary" in which the attributes of a combination of zero or more objects is looked up is an IUE_context object. This section briefly discusses this feature.

While C++ permits polymorphic operations by virtual dispatch on a single argument, CLOS has a more general multi-way dispatching techniques. The flexibility this offers is generally useful. While it would be nice to have full multi-way polymorphism, adding it to C++ is beyond our scope. The multi-key technique to be described provides a partial step; accessing attributes with multiple keys can be viewed as a weak form of dispatching to a behavior with an polymorphic argument list. Given that IUE_values can store strongly typed function pointers, one could use the multi-key technique to implement a multi-way dispatch.

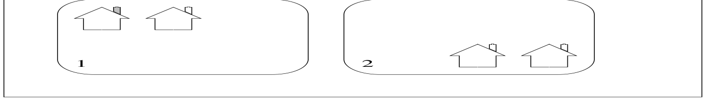
Dynamic attributes are stored in an IUE_context object, which is an indexed table. The *key* is a fixed-length sequence of IUE_object identifiers. Each key accesses a property list (a table of attribute-value pairs, indexed by attribute name). During execution, IUE_context can be connected into a hierarchy for value inheritance. This is accomplished by giving each IUE_context an ordered list of *parents*, and, for each parent, a specification of what subset of the key is used to access it. This search list is specified (and can be modified) during execution, though startup code can exist to link IUE_contexts into an initial hierarchy.

Given a key (representing a combination of zero or more IUE_objects) and an attribute, an IUE_context looks to see if it has a value stored. If it does, it returns it. If not, it sequentially queries each parent IUE_context until a value is returned. These queries can recurse, searching the data inheritance hierarchy up as many levels as desired. Cutoff conditions, based on depth, key value and parent identity, may be imposed.

Storing a value for an attribute always makes a local entry in the IUE_context, not in a parent. By default, inherited values do not change when a parent's value is changed. (A copy-on-write policy makes it appear that each IUE_context has a unique copy of each attribute, even if it was actually obtained by inheritance.) There is, however, a method for forcibly updating an inherited attribute in the descendants of an IUE_context. (Simply omit the copy-on-write. Then descendants will see the updated value.)

While an IUE_context may seem like a form of indexed database, it has a number of important differences. First, a key is always a sequence of object IDs, not values of arbitrary type. Secondly, the inheritance mechanism, with copy on write, allows for significantly greater flexibility and decreased storage requirements. For instance, there is an IUE_context provided for each subclass of IUE_object. Its key is a single IUE_object ID, and it is used by default to store attributes for each object of that class. Its sole parent is an IUE_context whose key is null. The parent thus provides intelligent defaults for attributes of that class. This is all done without taking space for each attribute in every object. We note that the object-level-context and default-level-context mentioned earlier are simply specialized IUE_context objects with keys of size 1 and 0 respectively.

An example is shown in figure 4. We want to know what color to draw the chimney of a particular house in image 1. We query an IUE_context (called pbi-context) which stores attributes indexed by [part, building, image]. Our particular key is the ordered triple of object IDs [Chimney, Our House, Image 1]. We may use the *get* method of pbi-context, passing it




```
color = pbi_context.get (part.ID() + building.ID() + image
color = part.get ("color", pbi_context, building.ID() + in
```

Figure 4: In an IUE_context, attributes can be associated with a combination of objects. If no value is stored for this combination, the parents of the IUE_context are queried, using a subset of the combination. See text for a discussion.

the key and the attribute name "color." We may also use the *get* method of the first IUE_object in the key, Chimney. We pass "color" and the key [Our House, Image 1], to which Chimney's ID will automatically be prepended.

If pbi-context has a color stored for this triple, it returns it. If not, it checks its parents. The first parent is an IUE_context of [part, building] pairs. Here we check if there is a color for Chimney on Our House (regardless of what image it is in) using the key [Chimney, Our House]. If no color is found, two parents are checked. One uses the key [Chimney], the other [Our House]. These would specify an inherent color for Chimney (in any image, as part of any building) or for any part of Our House (in any image). If no color is stored in any of the above IUE_context, one more is checked — the other parent of the original IUE_context, pbi-context. It is given the key [Image 1]. It may have a color which is to be used by default for anything drawn in Image 1. If it does not, the *get* fails.

Parent links may have a limit on how many levels up to search. If this limit is 1, no recursion is done; only the parents explicitly specified in the list are searched. If it is 0, there is no limit. Also, a parent link may specify particular IUE_context above which the search should not go. A parent link may also specify that if the key matches certain patterns, the link should or should not be used.

To date this multi-key aspect of dynamic attributes has not been highly used in the design (it was a late addition), but since it uses the same mechanism it costs little to gain this flexibility. Get/Ref/Put Accessors have optional forms that take contexts and keys to use in searching for a value.

7 Code Generation

This section discuss various aspects of the IUE's code generation facility including why it is in TEX, what it does, and its usefulness to date.

Code generation is a goal of many CASE software projects and a number of commercial generators exist, e.g. the full CASE systems Information Engineering Facility from Texas Instruments and Information Engineering Workbench from Knowledge-

Ware both support generation of Fortran, C and COBOL code from a "specification" (The specification, however, is far from being a document.) Today, most generators are based on "formal specification" languages or use graphical tools to produce graphical representations which are then interactively augmented to build the specification, see [Tan-1994]. Specialized tools for generation of graphical user interface skeletons, e.g. TAE and InterfaceBuilder, are also relatively common.

The idea of software development as a process of documentation, of which the actual code is just a component, is however making a come back, see [Welsh-1994]. Work by some of early masters of the 70's, including Dijkstra, Hoare, Wirth [Dijkstra-1976, Wirth-1971] took this view, and it was epitomized in Knuth's WEB system and work on "literate programming" [Knuth-1992]. While accepted as sound software engineering practice[Parnas and Clemens-1986], development of CASE tools to support for this approach have been lacking. Such tools are beginning to appear, see [Welsh-1994, Welsh and Han-1993], but to our knowledge none support code generation. The goal of a software project is not software it is a documented system including specification, code and rationale.

The use of TEX for the code generator is a combination of serendipity and choice, and naturally follows from the software-as-documentation viewpoint.

When the IUE specification project began, it was determined to use a commonly available system for typesetting the document. For a few reasons, including is cost (free), its portability and committee member familiarity, the decision was to use LATEX for the developing the overview specification. As the project progressed it was natural to continue with using LATEX supplementing it with a small set of TEX macros to provide more structured output of class definitions. Initially these macros simply formated the information, changing fonts, setting indentations, etc. Working with these macros, we recognized they provided sufficient information for the development of a complex indexing mechanism to help the specification team find related components and methods. In our own section (sensors and related things), the number of objects and methods was becoming so large that hand verification of types/methods was impeding our work. We realized that the macros for typesetting and indexing contained sufficient information for us to actually insure that types/methods used in one part of the document were defined in others. As the project progressed and we needed to do some proto-typing of parts of the hierarchy it was again clear that all the information we needed was in the specification. So a TEX-based code-generator was born.

Using the specification for generation of the code had 3 other important advantages.

1. It would allow direct usage of the existing specification. This would save on coding effort. In addition this would insure we used the variable and class names agreed upon by the committee. (While the "names" issue might seem trivial, determining the proper terminology was a important aspect of the committee's effort and at times lead to long discussions.)
2. We considered doing a one time conversion of the specification into a existing CASE tool suitable for code generation. However, TEX was a common language among the committee and other CASE tools are not. Thus such a conversion might negatively impact the committees interaction and oversight of changes during development. We felt that at this stage it was important to keep insure the "CASE-tools" were reasonably familiar to the domain experts even if it meant they were a little less palatable to code developers. Furthermore, a one-shot conversion would be either require a parser or converter nearly as complex as the TEXbased code-generator or it would be very labor intensive. We note that after TEX-based code-generation traditional CASE tools can be, and are, used to maintain the system.
3. Generating the code for new classes from the document would a promote a coding practice wherein any change to the class interface and/or representation was *first documented then implemented*. This would insure documentation and code development stay in sync, and would also promote more consideration of changes to object interfaces.
4. While systems exist for maintaining code and document consistency, they operate by extracting documentation from comments in the code. It was hoped that by developing the documentation first we would be more likely to obtain readable/understandable documentation and better maintain the development history of the software.

In addition, the initial IUE goals were for a dual-language system using both C++ and CLOS. The code-generator originally supported both.[7]

The TEX-based code generator provides macros to document IUE classes. This includes information on related classes including superclasses, subclasses, components (classes used as attributes), associated classes (classes used as arguments to methods). It includes a description of the class and how it is data exchanged. This information is used to generate program comments in the code files as well as determine the appropriate include statements. Special macros exist for includes which cannot be determined from the related class information. This information is used to generate the basic C++ class structure including inheritance. It is also used to generate basic information for the meta-class.

The class definition includes information on the various attributes, including their type, their default values and their documentation. Full visibility (public/private/protected) for each function or accessor is provided. All data is private and the generator mangle the name to make it less likely programmers will want to violate our encapsulation of the state. There are separate macros for the various forms of attributes (hard, soft, etc.).

The code generator produces all the access functions for attributes, as well as the code needed for proper initialization of associated items in the meta-class. In particular we insure that any type has its initialization code called before that a value of that type is created. This allows class constructors to presume existence of the meta-class object and its associated data. For efficiency, much of this initialization occurs before "main".[8]

Default values can also specify a method call to compute the default value. This computed default is computed only when the meta-class object is initialized. If a per instance computed "default" is desired, a cached-hook should be used. For hook attributes default methods stubs are generated which operate as soft slots (i.e. one can get from and put to them) until the user updates the stub.

The class macros also documents the functions (methods) of the class including their arguments (type, name, documentation), return type, documentation of functionality, restrictions, exceptions, implementor and implementation status. It generates the appropriate definitions in the header file and also generated "stub" methods which can optionally print out their arguments. The methods include sufficient information to permit semi-automatic "merging" of newly generated code with existing code, and such a tool is under development. The code generator automatically generates constructors (copy and argument-less) and supports specification of additional constructors.

The generator supports the full range of C++ types, including base C-types, defined structs, pointers, references and modification with const. For functions it supports friend declarations, inline, static and definition of pure virtual functions.[9]

The document allows "priorities" to be assigned to classes and methods and to generate code or printable documentation for only a range of priorities.

The first few versions of the code generator did not support templates because the original IUE specification had chosen not to use templates. Code generation was applied to the "cleaned up" version of the initial specification, in April (94) and of the 542 specified classes, 338 were code-generated to produce "working" skeletons.[10] Implementations were developed for approximately 28 classes. The generated code comprised approximately 91K lines of code and the implementation 6K lines. For the implemented classes the generated code accounted for approximately 65% of the necessary code.

Since April a number of changes to the IUE specification have been agreed upon including the use of templates in various aspects of the system. In addition more classes were specified. The current status is, approximately, 575 specified classes of which 256 have been code-generated and 104 have implementations. The generated code is approximately 109K lines and implemented code 42K lines.

[7] While the CLOS side is now out of date, we hope (resources permitting) to bring it back online late next year. As in C++ the CLOS generated code will generate stub-classes and methods. If important to the IUE we will implement the dynamic-attributes technique in CLOS using the meta-object protocol.

[8] The generate code automatically handles classes with self-references (i.e. a pointer to the same type). For longer cyclic references, *e.g.* x has-a $*y$ where y has-a z and z has-a x, the DAtype initialization mechanism is sufficiently powerful to force the required ordering, but requires minor editing of the initialization routines.

[9] The latter is more complex since it requires that no generated code (including defaults) invokes `new` on the abstract classes.

[10] Most of the non-generated classes were either incorrectly specified or required unimplemented template-like constructs such as sets or lists.

After the decision to use templates, we began extensions the TEX-based code generator and DAtype mechanisms to handle templated object definitions. The first release of these extensions were recently delivered to AAI.

8 Conclusions and Future work

This paper discussed two important software tools being used to support the IUE: dynamic attributes and a TEX-based code generator.

Dynamic attributes efficiently encapsulate state representation and allow significant flexibility in object-oriented designs. They allow the IUE to efficiently use a mathematically-oriented hierarchy with the derivation-by-constraint paradigm. While they help keep object size small—trading time for space—they allow specializations to add space for an attribute and then have existing code and libraries operate at nearly the performance achievable had the system designers made the same space-time choices.

In the near future we will be also be working on providing various performance enhancements, development of a TEX-free version and on fully templated version, including templated IUE_values. We hope to develop the templated IUE_values in to a full "envelope" package, building on the ideas of [Coplien-1992]. This will provide "self-extracting" strongly typed polymorphic variables for each IUE type.

The TEX-based code-generator supports the dynamic attribute mechanism and also produces all the code for meta-class instances, class definitions. It generates properly defined stubs for all methods. By working directly from the specification we maintain documentation/code consistency. The generator has produced over 100K lines of working IUE code. In the future we will be working on various enhancements to the TEX-based generator including more extensive document-based type-checking, better generation for abstract classes, full support for mixin classes and full generation of CLOS classes and stubs.

Acknowledgements

Thanks to the IUE committee members, the work has been fun and very enlightening. Thanks also to R. Lerner and others at Amerinex Artificial Intelligence for continual feedback and patients with the erratic pace of development. Special thanks to Cindy Loiselle of AAI who cleaned up the TEX macros for documentation preparation and helped document and clean the code generation macros. Finally we acknowledge Yong Su Kim, a Columbia student (BA CS '93) who help develop the initial type-checking and indexing set of macros which started us down this particular TEXpath.

References

[Booch, 1991] G. Booch. *Object Oriented Design with applications*. Bengamin/Cummings, Redwood City CA, 1991.

[Coplien, 1992] J.O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

[Dijkstra, 1976] E.W. Dijkstra. *A discipline of programming*. Prentice-Hall, Engelwook Cliffs, NJ, 1976.

[Ingalls, 1978] D. Ingalls. The smalltalk-76 programming system design and implementation. In *Proc. of the Fifth ACM Symposium on Principals of Programming Languages*, pages 9–18, 1978.

[Knuth, 1992] D.E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford, CA, 1992.

[Parnas and Clemens, 1986] D. Parnas and P. Clemens. A rational design process: how and why to fake it. *IEEE Transactions on Software Engineering*, 12:251–257, 1986.

[Pokorny, 1994] Cornel Pokorny. *Computer Graphics: an object-oriented approach to the art and science*. Franklin, Beedle & Associated, OR, 1994.

[Tan, 1994] Y.M. Tan. *Formal Specification Techniques for Promoting Software Modularity, Enhancing Documentation, and Testing Specifications*. PhD thesis, MIT, EECS, Jun. 1994. C.S. Tech. Report # 619.

[Welsh and Han, 1993] J. Welsh and J. Han. Software documents: Concepts and tools! Technical report, Software Verification Research Centre, Univ. of Queensland, Queensland 4072, Austrialia, November 1993. #93-23.

[Welsh, 1994] J. Welsh. Software is hsitory! In *A Classical Mind- Essays in Honour of C.A.R. Hoare*. Prentice-Hall, NJ, 1994.

[Wirth, 1971] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.