ISO/IEC JTC1/SC18/WG8 N1733
(supercedes N1710)

Class and Object Vocabulary

In the paradigm of object-oriented programming (OOP), data structures called objects interchange messages. Objects have fields that are either variables containing values or methods that define how the object reacts to messages of a particular form. Objects belong to classes that specify the fields defined for objects in the class. This metaphor has proven to be a valuable tool for understanding both processes and data. In particular, it offers a framework for describing the information represented by an SGML document.

This paper therefore presents a vocabulary inspired by the terminology and philosophy of OOP to describe a set of objects in terms of classes and properties of those classes. While similar to the terminology that inspired it, this vocabulary differs for the following reasons:

1. OOP is concerned with defining processes; an SGML document is a data structure that does not define any processing. Therefore, the framework defined in this paper consists solely of objects and classes without messages or methods.

2. OOP vocabulary is not standardized; a term with one meaning in one OOP language may be used slightly differently in another. The vocabulary described here must be precise so it can be used to define SGML concepts clearly.

3. OOP uses an intensional model: an object is an instance of a class, or one class is derived from another by intent, because it was constructed that way. An intensional model is appropriate for an efficient programming language because a programming language operates on well-defined constructs. The vocabulary presented in this paper, in contrast, is an extensional one: an object is an instance of a class, or one class is derived from another by coincidence, because that is the way the universe happens to exist. An extensional model is appropriate for identifying useful classes and objects and the relationships among them. It can be redundant. For example, an SGML parser or an intensional model would probably not define separate objects corresponding to an element declaration and for an element type. In an extensional model, however, there is no overhead associated with talking about a class of element declarations, an instance of that class, a class of element types, an instance of that class and the obvious relationships between instances of these two classes. While an extensional model is useful for exploring ideas, the concepts it provides are likely to be recast in intensional terms before they are incorporated in actual software.

An accompanying paper presents a model of the information represented by an SGML document using the definitions given below. The model is intended for use in understanding the concepts of SGML. As suggested in the above discussion of intensional and extensional modeling, it is not intended to serve directly as the basis for implementing an SGML parser. Furthermore, these terms will not necessarily be used in a possible future revision of IS 8879. They are presented here simply for use in intermediate work in formulating a precise description of the information represented by an SGML document.

Many of the definitions below are presented twice: first in intuitive terms and second in a more rigorous, mathematical form.

1. The Universe

Because this vocabulary takes a set-theoretic view, it deals with a universe that is a set. To avoid conflict with the SGML use of the word “element”, members of sets are called ***things*** instead of elements. This vocabulary is intended for use in defining models. A particular model (such as a model for SGML) will define the universe it uses. It may be the case that every thing in the universe is an object in the sense defined below, but this vocabulary does not require models to define their universes in such fashion.

Every model is based on primitives. In a practical sense, the primitives are simply well understood concepts that do not need further explanation. In a mathematical sense, they are undefined atoms. This vocabulary considers property names and possible property values to be primitives and makes to attempt to define them. However, any model developed with this vocabulary can define property names and values as necessary.

2. Objects

This section defines objects, one of the basic units discussed throughout this paper.

The matching of a ***property***[1] ***name*** with a ***property value*** is called a ***property assignment***, more formally:

Definition: A ***property assignment*** is an ordered pair. The first coordinate is the ***property name***; the second coordinate is the ***property value***.

Property assignments are used to define objects:

Definition: An ***object*** is a finite set of property assignments, all of which have different property names.

Note that everything that can be known about an object is determined by its fixed set of property assignments.

The verb ***exhibit*** is used to indicate the property value matched with a particular property name by an object. Formally:

Definition: If an object contains the property assignment (n, v), the object is said to ***exhibit*** the property value v for the property name n.

3. Classes

It is useful to discuss groups of objects that have property assignments with the same property names, especially when there are restrictions on the exhibited property values. This section defines terms for doing so.

A ***rule*** describes the permitted property values that can be exhibited for a property name by a group of objects.

Definition: A ***rule*** is a one-place predicate that partitions the universe into elements that satisfy it and those that do not.

The notion of rules, allows the formulation of ***property definition***:

Definition: A ***property definition*** is an ordered pair $(\boldsymbol{n}, \boldsymbol{r})$ where $\boldsymbol{n}$ is a property name and $\boldsymbol{r}$ is a rule.

A ***class*** is simply a finite set of property definitions. Formally:

Definition: A ***class*** is a finite set of property definitions, all of which have different property names.

Definition: If a class contains the property definition (n, r), the class is said to ***provide*** the property name n.

Classes and objects are related as follows:

Definition: An object is said to be an ***instance*** of a class if

1. The object exhibits a property value for every property name provided by the class.
2. Every property value the object exhibits for a property name provided by the class satisfies the rule associated with the property name in a property definition in the class.

Note that an object that is an instance of a class can exhibit property values for property names that are not provided

1. The word "attribute" is reserved for SGML attributes. "Characteristic" was used in an earlier draft of this paper which used "property" in another sense. Since that use of "property" is no longer needed, "property" is used here because it is easier to type and pronounce than "characteristic".

by the class.

Example: An SGML element type is a class that provides a property called ***generic identifier***. Each element of that type is an object that is an instance of that class, and all such elements exhibit the same value for the generic identifier property.

4. Derivation

Objects can be instances of multiple classes. In fact, all instances of a class *C* may also be instances of a class *D*. This may happen because *C* provides all the property names *D* provides, with rules that are at least as restrictive as those *D* defines for the same property names. In this case, *D* is said to be a ***base class*** of *C* and *C* is said to be derived from *D*. Formally:

Definition: A rule *r* is ***at least as restrictive*** as a rule *s*, if every value that satisfies *r* also satisfies *s*.

Definition: A class *C* is ***derived*** from a class *D* if:

1. *C* provides every property *D* provides (and possibly others);
2. *C*'s rules for every property *D* provides are at least as restrictive as *D*'s rules for the same properties (and possibly more so);

Definition: A class *D* is a ***base class*** of a class *C* if *C* is derived from *D*.

The following statements can easily be proven for derivations:

1. If a class *C* is derived from a class *D* then all instances of *C* are also instances of *D*. However, there may be instances of *D* that are not instances of *C*.
2. If a class *C* is derived from a class *D* and there are instances of *D* that are not instances of *C*, then *D* is not derived from *C*.
3. If a class *C* is derived from a class *D*, and *D* is derived from a class *E*, then *C* is also derived from *E*.
4. A class may be derived from more than one other class.

Example: A class-object model for an SGML element structure might define a class of elements with property names that indicate its generic identifier, its attribute values, and its content. The model might also provide a class of elements of a particular type. The latter class is derived from the former.

It is sometimes useful to discuss property names ***inherited*** by a derived class:

Definition: If a class *C* is derived from a class *D* then *C* is said to ***inherit*** from *D* the property names that *D* provides.

5. Meets

The ***meet*** of two classes is a third class such that all objects that are instances of both original classes are instances of the meet and all instances of the meet are also instances of both original classes. The meet provides all the property names provided by the original classes with a combination of both their rules. Formally:

Definition: A class *C* is the ***meet*** of two other classes *D* and *E* if:

1. *C* provides a property name if and only if either *D* or *E* provides the property name.
2. For every property name provided by both *D* and *E*, *C*'s rule is the conjunction of *D*'s and *E*'s rules; that is, *C*'s rule is that *D*'s rule and *E*'s rule must both be satisfied.
3. For every property name provided by *D* or *E* but not both, *C*'s rule is the same as the rule in the original class.

Note that while the meet of two classes is a modified union of the property definitions in the original classes; the set of objects that are instances of the meet is the intersection of the sets of instances of the original classes.

Example: Both HyTime architectural forms and SGML element types are classes. The meet of an architectural form and an element type is a class of elements satisfying the requirements of both.

6. Class Generators

The next topic is easily introduced by analogy. The reader may be familiar with the concept of a metalanguage, which is a language used to define other languages. A language is a set of rules for defining sentences and a metalanguage is a language whose sentences are definitions of other languages. The portion of SGML that defines document type definitions is a metalanguage; a particular document type definition defines the language whose sentences are document instances of that type. The same relationships can be expressed in terms of classes and objects.

In particular, some classes are used to define other classes. Such a class is called a ***class generator*** and its instances are called *types*. One approach to defining class generators would be simply to say that a class generator is a class all of whose instances are also classes. This approach would restrict instances of class generators to be simply classes with no other property assignments. In other words, it would require that all the property definitions provided by the class generator had rules requiring the associated property values to be rules. Since it may be useful to allow generated classes to have other property assignments, instead a class generator is defined to be a class that provides a property definition with a rule that requires the exhibited property value to be a class. So that there will be no ambiguity in identifying the class defined by an instance of a class generator, class generators are not allowed to have multiple property definitions that define classes.

Definition: A class is a ***class generator*** if among its property definitions there is exactly one with a rule that can be satisfied only by a class.

Definition: An object is a *type* if it is an instance of a class generator.

Definition: A type is said to ***provide*** a property name if the class it exhibits provides the property name.

Example: ISO 8879 implicitly defines a class generator whose instances are element types and another whose instances are attribute definitions. A DTD defines instances of these classes which exhibit the classes of element types and attribute definitions whose instances appear in a document instance. Some of these relationships are diagrammed in the following table, where some rows correspond to provided property names, and others correspond to exhibited property names:

Some SGML Property Names

Class	Provides/ Exhibits	Generic Identifier	Attribute Definition List	Content Model or Declared Content	Attribute Specification List	Content
Element Type Generator	Provides	X	X	X		
Element Type	Exhibits	X	X	X		
	Provides	X			X	X
Element	Exhibits	X			X	X

Note that an element type both provides and exhibits generic identifiers. However, the element type generator class's property definition has a very different rule for generic identifier than does a particular element type.