Object-Oriented Development of SGML Applications

Bob DuCharme bobducharme@acm.org May 1993

SGML, or Standard Generalized Markup Language, is an ISO standard for defining structure for documents and then marking them up according to that structure. When an application can take a particular structure (defined in a Document Type Definition, or DTD) for granted in an input file, it can take advantage of that structure to implement features that use the document's data for much more than mere formatting and printing.

For example, the best-known example of a large body of documents that conform (to some degree) to a particular DTD is HTML files. Mosaic and similar applications use this structure to implement hypertext jumps, bitmap displays, and audio and video clips in addition to assigning fonts to particular text elements. (I say "to some degree" because the SGML community in general agrees that the vague collection of DTDs currently serving that function for HTML are incompatible and often badly designed. Also, applications like Mosaic and Netscape are far too forgiving of non-compliant documents and accept many appearance-related elements and attributes that violate SGML's intent of representing only structure. Because of this, many SGML proponents feel that HTML files do not represent good examples of SGML. Still, this paper uses several examples from elements defined for HTML because of the general familiarity with these elements compared with those of other DTDs.)

An SGML document has much in common with an object-oriented system. In addition to some key vocabulary, there are many parallels between the analysis and design processes used in developing such systems. These common points can be exploited to minimize the work necessary to turn an SGML system into a useful object-oriented application, which can greatly speed the work of the rapidly increasing number of SGML developers.

While it may be tempting to coin some horrid new acronym like "OOSGML," it's more important to remember that SGML systems and object-oriented systems are mutually exclusive. An intrinsic part of an object-oriented system is the specification of object behavior, while good SGML design specifies only object structure and intentionally excludes the behavior of elements (the "objects" that make up a document and the document itself) as much as possible to give maximum flexibility to developers using that data. Still, developers can gain many benefits by taking advantage of the concepts and approaches that SGML and object-oriented technology have in common.

To examine these benefits, this paper explores three areas:

- Key object-oriented concepts and terminology and how SGML concepts relate or don't relate to them.

- The most well-known Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) literature, and which of the major object-oriented development steps (as outlined by the literature's authors) can be skipped or automated when starting with an existing SGML system.

- STSGML, a toolkit that demonstrates these ideas by automating conversion from an SGML representation of element structure to representation as Smalltalk classes and objects. In addition to the core classes that automate this process, STSGML includes a sample application that demonstrates how little code is necessary to create SGML applications once the core classes have done their job. Unlike most existing applications of object-oriented technology to SGML systems, STSGML accepts data from arbitrary DTDs, demonstrating the potential flexibility of this adaption of SGML to object-oriented systems.

Previous Work

Queries on the Usenet newsgroups comp.lang.smalltalk and comp.text.sgml have revealed that there is definite interest in the use of Smalltalk for SGML development, but very little existing work. Most SGML work using an object-oriented approach has consisted of applications that are hard-wired to use only documents conforming to a particular DTD. [Zha95].

HyTime is an ISO standard (ISO/IEC 10744:1992) implemented as a set of extensions to SGML in order to represent the structure of hypermedia and multimedia documents. Many researchers in the field of multimedia have already decided that an object-oriented database manager makes more sense than relational technology to manage the relevant data types [BA94][New93], and enough have used HyTime to implement their systems that the majority of references to object-oriented technology in SGML literature seem to be in HyTime discussions.

SGML and Object-Oriented Technology

SGML and object-oriented technology have a great deal to offer each other. For object-oriented developers, SGML can offer the benefits of having much of the OOA and OOD already done for a given body of data. A developer using a pre-existing DTD will find that much, if not most, of the analysis and design work described in the classic object-oriented literature has already been performed.

Object-oriented technology can offer the SGML developer even more:

- Graphical user interface application development tools, which are sorely lacking at this point in SGML's history.

- Literature on OOA/OOD that can contribute much to the the scarce literature on DTD development. (See "OOA/OOD and DTD Development.")

- Automation of much of the process of developing finished applications. Once code is developed to implement certain primitives such as reading in a DTD, defining a class for each document element described there, and reading in an SGML document and instantiating its elements using the defined classes, then manipulating the document object and its component objects to perform useful tasks can be done with very little code. The STSGML demo application converts SGML-compliant HTML documents to TeX and RTF with barely a page of code in addition to the kernel code.

- The tools to manage bigger and bigger collections of data. As SGML applications use increasingly larger collections of data, developers are finding that relational database management tools are not a good fit. The variable size of text elements, the sharing of subelements, the frequent need to traverse hierarchical relationships, and the increasing use of SGML in multimedia applications make object-oriented database management systems (OODBMS) a much better fit to SGML than relational systems. [RH90] (This is described more under "Further Research.")

What They Have in Common

SGML and Object-Oriented Terminology

SGML and object-oriented technology have enough terminology in common to be helpful at certain times and confusing at others. To establish a vocabulary with which to discuss their relationship, this section reviews key object-oriented and SGML terms that overlap or are connected. The object-oriented definitions are from *Object Oriented Design with Applications* by Grady Booch and *Object-Oriented Modeling and Design* by James Rumbaugh et al.; SGML definitions are from ISO 8879, the SGML standard. Any italics in quotations are copied from the source.

Documents and Elements as Objects

The terms "class" and "instance" are used in SGML, but with more limited meanings than in their object-oriented usage. Other SGML terms make up for this limitation and can be mapped to an object-oriented context to show how SGML documents and their components can be treated as objects, or instantiations of predefined classes.

Section 4.96 of ISO 8879 defines a document as "A collection of information that is processed as a unit. A document is classified as being of a particular document type." Section 4.102 defines a document type as "a class of documents having similar characteristics; for example, journal, article, technical manual, or memo." Section 4.110 defines an element as "A component of the hierarchical structure defined by a document type definition; it is identified in a document instance by descriptive markup, usually a start-tag and end-tag," and section 4.114 defines an element type as "A class of elements having similar characteristics; for example, paragraph, chapter, abstract, footnote, or bibliography."

Neither the element type nor document type definitions mention how the DTD, when specifying the document’s components and their legal arrangements, defines the structural relationships among the elements of a document. [Gol90 pg19] Given this and SGML’s use of the term "attributes" to identify element characteristics, most of Rumbaugh’s definition of the word "class" applies to the SGML terms "document type" and "element type": "An *object class* describes a group of objects with similar properties (attributes), common relationships to other objects, and common semantics...The abbreviation *class* is often used instead of *object class*. Objects in a class have the same attributes and behavior patterns. Most objects derive their individuality from differences in their attribute values and relationships to other objects. However, objects with identical attribute values and relationships are possible." [RBPEL pg 22] Booch’s definition is more concise: "A class is a set of objects that share a common structure and a common behavior." [Boo91 pg 93]

The only glaring difference from the SGML definitions are the references to object behavior, which is mentioned nowhere in the discussion of SGML components. This issue is addressed below in “Behavior of SGML Objects.”

Instance

Discussing instances, Rumbaugh writes "Each class describes a possibly infinite set of objects. Each object is said to be an *instance* of its class. Each instance of the class has its own value for each attribute but shares the attribute names and operations with other objects of the class." [RBPEL91 page 2] SGML uses the term "instance," but only when referring to a document, not its component elements, as shown in this ISO definition in section 4.160: "instance (of a document type): The data and markup for a hierarchy of elements that conforms to a document type definition."

Dr. Charles Goldfarb, the inventor of the SGML language, writes that "One can therefore speak of documents and elements almost interchangeably: the document is simply the element that is at the top of the hierarchy for a given processing run. A technical report, for example, could be formatted both as a document in its own right and as an element of a journal."[Gol90 pg 11] In fact, the definition of the "document element" shows that documents are treated as a special case of elements, in ISO section 4.99: "The element that is the outermost element of an instance of a document type; that is, the element whose *generic identifier* is the *document type name*."

Keeping this in mind, if we view SGML elements as instantiations of element declarations and substitute these terms in Rumbaugh's description of instances, we get an accurate description of their relationship: "Each element declaration describes a possibly infinite set of elements. Each element is said to be an *instance* of its declaration. Each element has its own value for each attribute but shares the attribute names with other elements described by the same declaration."

Booch writes that "An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable." [Boo91 pg 77] So the parallels hold: SGML documents and elements, once their behavior is defined, can be treated as objects.

Attributes

ISO 8879 section 4.9 defines an attribute as "A characteristic quality, other than type or content." SGML recognizes two levels of attributes: primary and secondary[Gol90 pg 11]. Primary tags are the IDs unique to each declared element type that are sometimes called "generic identifiers" (often abbreviated as "GI") that are used as the "tags" that identify the start and, if necessary, the end of elements. Generic identifiers usually do this by including the element's name in angle brackets and by preceding the closing tag's element name with a slash. (For example, the <h2> and </h2> tags that enclose the second-level headers declared by many DTDs.)

Additional attributes such as the HTML anchor element's NAME attribute, which identifies the anchor's unique name within the document so that it can serve as a jump destination, and the HREF attribute, which identifies a jump destination if the anchor is clicked, are considered to be secondary attributes. In common SGML usage, "attributes" refers to these secondary attributes.

The STSGML system uses each element's GI to determine which object class it belongs to and then uses the secondary attributes as instance variables of that element type's "class." Both of these are in keeping with Rumbaugh's definition of an object attribute: "a data value held by the objects in a class...An attribute should be a pure data value, not an object. Unlike objects, pure data values do not have identity. For example, all occurrences of the integer 17 are indistinguishable, as are all occurrences of the string 'Canada.'"[RBPEL91 pp 23 - 24]

Goldfarb makes the following recommendation about naming SGML element attributes: "The GI is normally a noun; the attributes are nouns or adjectives that describe significant characteristics of the GI. (The use of verbs or formatting parameters, which are procedural rather than descriptive, is strongly discouraged because it defeats the purpose of generalized markup.)"[Gol90 pg 33]. While the second sentence demonstrates the SGML philosophy of avoiding "behavior" definition, the first holds with Rumbaugh's description of an attribute's purpose--that it describes a particular quality of an object instead of itself being an object.

State

Booch defines object state as encompassing "all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties." [Boo91 pg 78]

Determining possible states of objects and the role of state transitions plays a large role many OOA/OOD methodologies. The state transitions are what makes good analysis and design so important, because lack of control over these transitions, especially in a multi-user system, leads to the kind of concurrency problems that cause unpredictable behavior.

State transitions are not a big factor in an SGML system until you build an editor--especially a multi-user editor--on top of an object-oriented SGML system. Unlike, for example, a car undergoing assembly, there are few state transitions to plan around for an SGML object once its data and components are assigned at object creation time.

SGML is typically used as a source format that makes it easy to create multiple different output formats, so transformation of the data is done to copied data being output, while the source is left alone. When adding methods to an SGML system to allow editing of the data, especially multi-user editing, keeping track of various states would obviously play a more important role, but one whose implementation could hopefully be left to the OODBMS system that performs object locking and other database management functions that allow serialized multi-user access. Because such a large percentage of current work with SGML data is done to create different forms of output without affecting the original source, this paper only addresses state transition issues in the “Further Research” section.

Behavior

Booch defines behavior as "how an object acts and reacts, in terms of its state changes and message passing." [Boo91 pg 80] Good SGML DTD design deliberately avoids definition of element behavior to grant maximum flexibility to developers using that data. As the definition of the term "publishing" expands almost monthly, publishers must store their text, picture, and audio data in formats that may be used by media that have not yet been invented when the data structure is defined. Creators of the Netscape World Wide Web browser have taken much heat for defining new elements and attributes for HTML files used by their product that, in the name of giving developers more control, have actually limited the use of the data to computer screens. (For example, imagine developing an application to take these files and convert them to Braille. What do you with a Netscape elements like <blink>?)

The SGML developer who wants to take advantage of object-oriented tools must assign behavior to the SGML elements so that a document can easily be plugged into an object-oriented system. Because this can be done by an external application that, for the purposes of this paper, leaves the data and its structure alone, the integrity of the SGML data can be maintained.

So, we've identified the following correlations between SGML and object-oriented terms:

SGML term	**Object-Oriented Term**
attribute	attribute
document instance	instance, object
element	instance, object
element declaration	class
document class	class

OOA, OOD, and DTD Development

While traditional structured design advocates code re-use by encouraging the development of functions and procedures that can be called from many different contexts, a key advantage of the object-oriented approach is the emphasis on *design* re-use as well as code re-use. Most of the goals and even methods of DTD design have much in common with those of object-oriented analysis and design. For a developer using SGML data, the existence of a good DTD means that the majority of the analysis and design has already been done and stored in a clearly defined notation that can be easily parsed and used to declare classes in an object-oriented system. Analysis of typical SGML systems lets us define even more aspects that can be automated, reducing the developer's work even more.

While there is currently little literature on DTD development, there is a good deal on object-oriented analysis and design that can benefit the DTD developer. The basics of this literature discuss things that the DTD developer already knows; for example, Booch writes that "By studying the problem's requirements and/or by engaging in discussions with domain experts, the developer must learn the vocabulary of the problem domain. The tangible things in the problem domain, the roles they play, and the events that may occur form the candidate classes and objects of our design, at its highest level of abstraction." [Boo91 pg 191] This also applies to the developer of a DTD, who must become familiar with the different possible roles of different document elements in different output formats as well as the development and production process of publishing systems ranging from elementary school textbooks to nuclear reactor installation manuals. Ample OOA/OOD literature provides the DTD designer with many tools for following through on this process.

The DTD designer also faces problems similar to those of the object-oriented systems developer. For example, Rumbaugh writes that "In modeling an engineering problem, the object model should contain terms familiar to engineers; in modeling a business problem, terms from the business; in modeling a user interface, terms from the application domain." [RBPEL91 pg 17] The DTD designer and the object-oriented system designer must both learn the language of their clients and the principles underlying the language so that they can understand exactly what is needed. Like Rumbaugh's designer, the DTD designer must work to get beyond the terminology, which may be misleading. Publishing professionals are worse than computing professionals in applying terms to concepts long after the concept has evolved to have no relation to the term; perhaps the most well-known is the still-common use of the term "leading" (pronounced "ledding") to describe the amount of space between printed lines because this used to be accomplished by the insertion of small pieces of lead. A less picturesque example is the tendency by page layout people to think of the terms "emphasize" and "italicize" as synonymous, which causes problems when one considers how many output media allow emphasis but not italics. The SGML goal of separating structure from behavior (which, at the simplest level, means separating structure from appearance) is precisely what gives the object-oriented designer such a wide range of ways to use of the data.

SGML and Popular OOA/OOD Methodologies

Before enumerating typical behavior expected of SGML objects, examining the larger picture of the system analysis and design process makes it clearer exactly what work must be done to turn an SGML system into an object-oriented one. To do this, we attempt to identify the following within the most popular methodologies:

- The steps that correlate to DTD design and implementation--in other words, the work that has already been done for us that we want to re-use.

- The work that still needs to be done. In general, this means object behavior specification.

Of the many published approaches to OOA/OOD, Grady Booch’s is the most quoted, and by now considered the de facto most authoritative. His incorporation of principles from other methodologies, such as Wirf-Brock’s CRC (Class-Responsibility-Collaboration) and Jacobson’s Use-Cases, widens Booch’s perspective, and lends weight to his authority.

The Object Modeling Technique (OMT) was developed at General Electric by Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen. The recent acceptance by James Rumbaugh of a position at Grady Booch’s Rational Software corporation will consolidate the first and second most popular OOA/OOD design methods into what many foresee as a default industry standard.

An SGML document with behavior defined as a collection of methods for each element class is not a complete system. The requests for method execution have to come from somewhere, and in a modern system they probably won’t come from commands typed in at a prompt, but from other objects defined in software: dialog boxes, report generators, SGML entity managers. Although most available methodologies describe the development of an entire system, we’re more interested in creating a document object that can be easily plugged into new systems. This is more of an issue in some methodologies than others; by viewing the document as a server providing data by request to clients, we can refer to a client’s possible role without getting too specific about it in system design.

Booch

Booch’s recommendation that one separate logical and physical design of a system has particular relevance in SGML, in which good design means an awareness of the separation between logical structure and entity structure. See the section “SGML Entity Management” for more on this.

For object-oriented system development, Booch advocates a spiral approach, in which the four basic steps of analysis and design are repeated at finer levels of abstraction:

1. Domain analysis, or the identification of the classes and objects necessary for the system. For an SGML system (excluding the client using the documents), this work is already done for us in the DTD design and creation.

2. Identification of the class and object semantics. Because this part is deliberately left out of an SGML system, this seems to leave a lot of work for the developer. Through the analysis of currently typical SGML applications, however, we can identify enough semantics to automate the implementation of a useful core group.

3. Identification of the relationship between the various classes and objects. This too is already done for us in a DTD's explicit specification of the aggregation and ordering of each of a document's components.

4. Implementation of these classes and objects. This can be automated with a program that reads the DTD, creates classes for the declared document and element types, and implements the semantics identified for step two as the core group of methods for these classes.

So, for a simple enough application, a developer's task has been reduced from four steps to practically nothing in a simple enough application. In more complex ones, the developer augments the work of step two and, to implement these newly identified semantics, the appropriate work on step four.

Rumbaugh

The OMT approach concentrates on first building a model of the application domain and then working out implementation details during the design. There are four basic stages (some summaries omit the fourth):

1. *Analysis* is done through the development of three interdependent models of the problem domain: the object model describes the relevant objects; the dynamic model describes the behavior of each object by classifying its role in the various events that may happen within the system and the flow of data between objects; and the functional model shows the steps necessary to perform any transformations of object data values.

2. *System design* is the breakdown of a system into subsystems and the allocation of system resources to those systems.

3. *Object design* is the design of the objects that were identified and had their behavior classified in the first two steps.

4. *Implementation* is the actual coding of the working system. If the first three stages were done well, this is supposed to be a minor, mechanical part of development.

The system design stage correlates closely to the design of SGML entity structure. The section “SGML Entity Management” describes why this topic is not central to the concerns of this paper, but a promising area in which to extend this research. The fourth stage, implementation, corresponds to Booch's fourth stage and can be automated in a similar fashion.

This leaves us with the analysis and object design stages. Of the three models created as part of the analysis, Rumbaugh writes of the first that "the *object model* captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterize each class of objects. The object model is the most important of the three models."[RBPEL91 pg 21] It's the most important, and it's the one that's already done for us by the DTD, except for the behavior, or "operations" that characterize each class of objects.

"The dynamic model," he writes, "specifies allowable sequences of changes to objects from the object model."[RBPEL91 pg 110] Because we're working with static documents, we don't have to worry about changes once each object's values are assigned at creation time. Dynamic behavior is described with state diagrams, and he writes that we need "only construct state diagrams for object classes with meaningful dynamic behavior. Not all object classes require a state model."[RBPEL91 pg 112] So we can forget about it in the single-state model of an SGML document that we are working with.

The third model of the analysis stage is the functional model, in which we specify the algorithms necessary to transform data values. If we view an SGML document as a database, it reduces our work if we heed the following advice from Rumbaugh: "By contrast [to interactive programs], databases often have a trivial functional model, since their purpose is to store and organize data, not to transform it."[RBPEL91 pg 123] For a system in which editing is not an issue, transformation of data is done by a separate component, so the functional model plays a minimal role in analyzing an SGML system for conversion to an object-oriented system.

So of the three models constructed for the analysis phase, the object model, which identifies the objects and their attributes, relationships, and behavior, is the only one we need to worry about, and the DTD has done all but identify the object behavior. The DTD also takes care of much of the object design phase, in which we specify the structure of the individual objects identified in the analysis and the details of the behavior.

To summarize, what remains is the same thing that remains when we take the Booch approach: we look at the objects whose structure, relationships, and ordering are specified in the DTD, and assign some behavior to them, and we'll have a document object that can easily be plugged into a complete object-oriented system.

SGML Elements as Objects

A basic tenet of the object-oriented approach is that we create a model of object behavior before worrying about details of object structure, because the structure is supposed to support the implementation of the behavior. We can model the behavior of SGML document and element objects by examining their use in current typical publishing systems. Once the expected behavior is established, we can determine their most efficient structure.

Behavior

What is the behavior of SGML objects? We must be careful about modeling them on real-world concepts, because this can limit their usefulness--for example, modeling a paragraph of body text after its equivalent in a newspaper might ease its use in marketing literature but introduce difficulties when using it in on-line help. The behavior of SGML elements is the sum of the services they provide, so we have to examine the services that a developer needs. Doing so will show that, for a given DTD, an automated system can define a great deal of useful default behavior for the element classes defined in nearly any DTD.

These services fall into three categories:

- The most trivial methods would provide the ability to set and read each of an SGML element's attributes. For our purposes, methods that set attribute values will only be invoked at object creation, but they can eventually prove useful for other application features.

- Most SGML development consists of conversion of the data to incorporate some other form of markup--for example, TeX, RTF, a proprietary markup scheme used by an on-line document viewer, or even a set of tags defined by another DTD--so the ability to freely mix new text with the output is essential.

- Being a database format, SGML must allow applications to query various aspects of a document's state, as well as those of its elements, so that it can act on that information to create different publications. This would enable the creation of language reference manuals, users manuals, tutorials, and quick reference manuals for the same software product on different platforms all by extracting different subsets of information from the same SGML database.

Implementing the first category of methods would be trivial. For each attribute defined for a given object, the system must also define a method to set that attribute's values and one that returns that attribute's value when queried. To return the text contents of an element, the latter method must be recursive, because many of the nodes in a document tree are interior nodes.

For mixing new text with object content when it is output, a convenient approach is one taken by the languages perl (a popular one among SGML developers), awk, C, and C++: format strings. If specified symbols represent the contents of each of an element's attributes, they can be combined into a string that has other characters to be output at those positions relative to the attribute values. For example, if "<>" represents an element's text and "<NAME>" represents the value of its attribute NAME, then the string "{\b <NAME>}: {\i <>}" would format the element's output with the RTF codes necessary to print the NAME value, bolded, followed by a colon, a space, and the the element's text in italics.

This string could be passed as a parameter to the object when requesting its contents, but the aggregate nature of the documents means that asking for a formatted version of a given element is also asking for all of its subelements, so it makes more sense to store a format string as a class variable of each element type. This way, we can specify all of the format strings before asking for the formatted aggregate version.

The third category of behavioral methods is the broadest, and the one that developers moving beyond the automated system will spend the most effort developing. Still, a core group of features can be observed in current typical systems and added as part of the automated system to provide much value to developers who do not take the trouble to extend this category.

Useful methods to return information on an SGML element object would include the following:

- A boolean method that indicates whether the string passed as an argument is part of the object's content. A typical use would be an application that pulls out sections of a reference work with a given string in the section's title in order to create a more specialized document.

- A method returning the class of the root element in a document tree--in other words, the class of document that the element is a part of. An application processing a chapter object could then treat it differently depending on whether it is a part of a software users manual or a cookbook.

- A method returning the class of an element's parent. A list item whose parent is another list item would be part of an embedded list, and require formatting different from a list item whose parent is regular body text.

- A boolean method that indicates whether a document element is the first of its kind sequentially--in other words, whether the sibling preceding it is an object of the same class. This is useful for formatting programs that want to, for example, put more space before the first item of a list than before the other items.

- A method returning the DTD being used, or at least some information about a document's structure so that someone querying that document would be aware of the choices available.

Some of these methods would be relevant to all SGML element objects, and some would not. For example, the last three: the root of a document tree has no parent or predecessors, and storing the information about a document tree's structure with every node of that tree would be very inefficient if tree traversal methods are available to find the single node that stored this information. The document object, the root node of the tree, is the logical place to store this information.

These kinds of similarities and differences among element objects give us clues as to the classes we will need and, as Booch calls it, the "class structure," or hierarchical relationship created by the inheritance patterns of the different classes.

Structure

Treating element types as classes means that for each element type (including the document element type) defined in a DTD, the object-oriented system using this data will declare a new class with that element's attribute types as instance variables. The process of reading a document instance then becomes the process of instantiating objects to these declared classes.

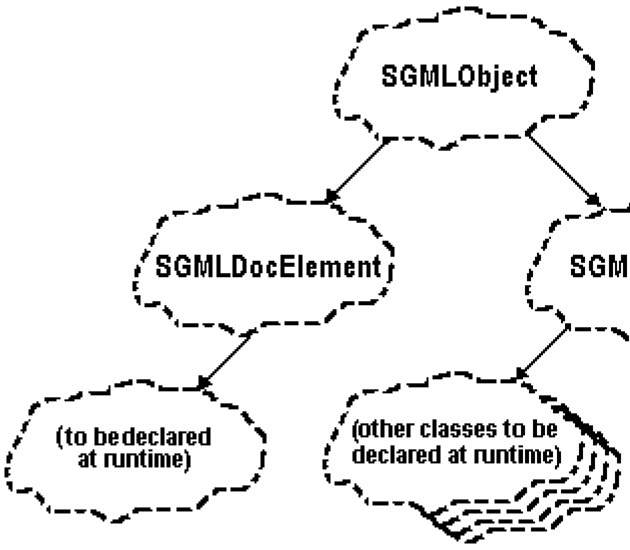
The developer can add methods to these classes after their definition, but as we saw in the previous section, many will be common to all of them, so we will define an abstract class (a class designed purely to provide common behavior to descendant classes without itself ever having instantiations) SGMLObject from which all of these can be derived. This also makes it easier to add new methods that can be used by all the element classes, because these methods only have to be added to the SGMLObject class.

We have seen that it is possible for the root node of a document tree to have attributes (in our example, the DTD) that others do not, and for component elements of the document to have an attribute that the root element does not, so we derive two more abstract classes from SGMLObject called SGMLDocElement and SGMLElement.

Why are these abstract classes? Why not instantiate document elements to these two classes as they are read in instead of defining a new subclass of one of them for each element type? The answer underscores a principle advantage of using an SGML DTD for creating an object-oriented system: a DTD may have separate attributes defined for each element type, and we can use these definitions to create attributes, or instance variables, for the SGML element objects. This creates specialized classes for each document element, which is part of the point of inheritance--by allowing us to define an ancestor and the attributes that set a class apart from its ancestor, we can easily re-use the code originally defined for the ancestor.

We also define one more descendant of this hierarchy: the concrete class PCDATA, descended from the SGMLElement class. Instances of PCDATA, or "parsed character data," make up the majority of the leaf nodes in a document tree; internal nodes of the tree contain other nodes, but the leaves generally contain the character data that comprises a document. (I say "generally" because it doesn't have to; they may contain references to other entities as well, such as picture, sound or video files.)

The following shows a Booch-style diagram of these classes:


SGMLObject
SGMLDocElement
SGMLElement
(to be declared at runtime)
(other classes to be declared at runtime)
PCDATA

Aggregation and SGML Element Objects

The principle relationship between SGML element objects is that of aggregation. At the logical level, nothing defined in a DTD exists apart from the document element, and nearly all the elements exist as part of larger and larger elements until you get to the document itself. (At the physical level, the location of the entities is irrelevant; worrying about it is the job of the entity manager.) The Bible is made up of testaments, which are divided into books, which are divided into chapters, which are divided into verses. A cookbook might be divided into chapters that are divided into recipes, which can themselves be divided into components that follow a specified structure: title, optional illustration, optional attribution, introduction, ingredients, instructions, and suggestions for accompanying dishes.

Rumbaugh cautions against overuse of aggregation: "when in doubt, use ordinary association" [RBPEL91 pg 37]. However, he includes several tests[RBPEL91 pg 58] to consider and the relationship of an SGML document or element to its subelements is clearly one of aggregation: "Would you use the phrase *part of*?" Yes. "Are some operations on the whole automatically applied to its parts?" Yes; when printing a book with no qualifiers, the intent to print its component chapters with the book clear. "Are some attribute values propagated from the whole to all or some parts?" Yes; if a legal code section has a status attribute with a value of "proposed" or "revoked," this would apply to its subsections. "Is there an intrinsic asymmetry to the association, where one object class is subordinate to the other?" Yes; subordinate relationships are clear in hierarchies such as the Bible and cookbook hierarchies described above.

STSGML

The STSGML (for "Smalltalk SGML") application is an example of how an SGML document instance, set up to be treated like other Smalltalk objects, can be plugged in and used. STSGML has two key object classes that demonstrate the use of an SGML document and its components: an object of the STSGML class is the engine that reads a DTD, declares the element classes and their methods, and reads in parsed SGML data, instantiating its elements to the appropriate declared classes. An instance of the STSGMLWindow class provides a menu-driven interface to the use of the STSGML engine, allowing the user to save a document object in a plain text file or a formatted file, with sample formatting scripts provided to turn data conforming to two widely-used DTDs to RTF and TeX files. Appendix A of this paper includes the text of the STSGML on-line help, which explains how to use it.

Why Smalltalk?

There are two reasons:

- Being an interpreted language, it allows the creation and instantiation of new classes defined with names and attributes determined at runtime.

- To examine the potentially intrinsic object-oriented properties of SGML, using Smalltalk forced me to think in object-oriented terms as much as possible. I have heard of C++ programmers who learned Smalltalk and returned to C++ as better C++ programmers; Smalltalk’s treatment of even classes and metaclasses as objects forces you to model absolutely everything in terms of objects communicating via messages.

The application uses Digitalk’s Smalltalk/V Win32 2.0, because it was the most affordable Smalltalk implementation in a windowing environment at the time. While Smalltalk/V allows the distribution of runtime executables, this turned out to be impossible for STSGML, because the subset of Smalltalk/V classes included with the runtime module do not allow the declaration of new classes at execution time.

Structure and Behavior of SGML Objects in STSGML

STSGML reads a document DTD to gather information about an SGML document’s structure, but instead of reading the document instance directly, it passes the document data to SGMLS, a publicly available validating SGML parser, and reads the SGMLS output. This frees STSGML to concentrate on manipulating a document’s structure instead of worrying about whether the document conforms to the structure defined for it.

Each new class that is declared as a new element definition is read in from the DTD and given a name consisting of the document element, in all upper case letters, concatenated with the element name in all lower case. For example, the h2 element in the HTML dtd would be defined as the HTMLh2 class. This makes it possible to define different classes for elements of different DTDs with the same name; names like h1, h2, h3, etc. to denote heading levels are common in many DTDs, not to mention more obvious element names like "chapter" or "section."

The class derived from the SGMLDocElement is simply the name of the document element in all upper case.

In addition to the instance variables and methods described below, newly declared element classes derived from SGMLObject’s SGMLDocElement and SGMLElement descendant classes will have attributes and methods declared by the STSGML engine object for each of that element’s attributes defined in the DTD.

Two methods will be defined for each new attribute. The first, which sets the attribute’s value, will have the same name as the attribute, plus a colon, indicating that an argument follows (for example, "h2:"). The other, which will return the attribute’s current value, will have the same name as the attribute. These method names follow Smalltalk convention for methods that set and return attribute values.

Smalltalk methods generally have the syntax

```
receiver methodSelector
```

to send a message to the receiver object telling it to execute its *methodSelector* method. To assign the variable x the value returned by sending this message to the receiver object, the syntax would be the following:

```
x := (receiver methodSelector)
```

(As with other languages, the parentheses may not be necessary, but ensure the intended precedence of operations.) Methods that expect arguments to be passed have colons in their name; the following tells the receiver object to execute its *methodSelector:* method with the string "blue" as a parameter:

```
receiver methodSelector: ’blue’
```

Methods that pass multiple arguments are written as multiple words with colons preceding each argument. For example, the *methodSelector:startingAt:* method might be invoked as follows:

```
receiver methodSelector: ’blue’ startingAt: 3
```

The remainder of this section describes the class and instance variables and methods defined for the abstract classes SGMLObject, SGMLDocElement, and SGMLElement. (The concrete class PCDATA, derived from SGMLElement, has no special methods or variables as part of its declaration.)

SGMLObject

The only class variable for the SGMLObject class is the format string, which stores data to be interpolated with the contents and attribute values of the element when a client object requests a formatted string.

The only instance variable in an SGMLObject is a list of its components called "elements" which is implemented as a Smalltalk OrderedCollection. A document representing the Bible would have two objects in its elements list; the second of these, representing the New Testament, might have fourteen objects in its element list, representing the fourteen books of the New Testament, and so on.

The leaf node elements in a document tree, which would be instances of the class PCDATA, will have a string as the first and only object in their element lists.

Class methods:

formatString: assigns a formatting string for the class.

formatString returns the currently assigned format string for the class.

new initializes a new instance of the class and calls the *init* instance method.

Instance methods:

addElement adds an element to the list of the receiver's subelements.

asFormattedString returns the object in string format with any formatting text interpolated.

asString returns the object as a string.

at: returns the object's component at the location specified by the number passed as a parameter.

*before:ifNone:*returns the object that precedes the object in the receiver collection, and executes the code passed as a parameter to ifNone: if there is no such object. SGMLObject merely returns the result of performing the standard OrderedCollection *before:ifNone:* method on the receiver's element list.

hasSubstring: returns a boolean True if the receiver contains the substring passed as a parameter, either as its own content (that is, as a string in its own elements list) or as the content of one of its component elements.

init initializes the list of the object's elements as an instance of the OrderedCollection class.

asFormattedStringWithReplacements: just like asFormattedString, but with replacements specified in the dictionary passed as an argument. Each dictionary entry is keyed on the target string and has a replacement string as a value.

SGMLDocElement

An instance variable stored with an SGMLDocElement object in addition to the inherited elements variable is called "dtd." It doesn't store all the information stored in an actual SGML DTD, but just the subelements and attributes that make up each element defined in a DTD. This is implemented as a Smalltalk Dictionary object keyed on element name, with the entry for each element name being an instance of the class ElementDeclaration. An ElementDeclaration object stores the attribute names and subelement names in two Smalltalk lists known as "attributes" and "contentModel" respectively. Because STSGML relies on the SGMLS program to do the validation, other DTD information is not necessary to convert a document into a useful object.

The SGMLDocElement class has no class variables.

Class methods:

elementName returns the name of the element that the class represents. In the case of the SGMLDocElement class, this just returns the class name as a string.

Instance methods:

attributeNames returns a list of any attributes defined as instance variables of the document object.

dtd returns the object's DTD information.

dtd: sets the document object's dtd instance variable to the dtd Dictionary object passed as a parameter. The readDTD method of the STSGML engine creates a dtd object when it reads in the DTD file corresponding to a given document instance.

SGMLElement

SGMLElement's only instance variable in addition to the inherited elements variable is "parent," which points at a given element object's parent. This is necessary for traversal of the document tree.

This is not a class variable because the same element can have different parents depending on the context--for example, an illustration element might appear within many different other elements in the same publication.

The SGMLElement class has no class variables.

Class methods:

elementName returns the name of the element that the class represents. For example, an HTMLh2 class represents an h2 element.

Instance methods:

attributeNames returns a list of any attributes defined as instance variables of the document object. This is defined separately for SGMLElement and SGMLDocElement instead of being defined in SGMLObject and inherited by the two subclasses because they use different methods to look up their corresponding DTD information--SGMLDocElement just looks in its own dtd instance variable, while SGMLElement must traverse a document tree using the parent attributes to find the DTD where it is defined.

docElementDTD returns the DTD where this element is defined by traversing the document tree to the root and returning the DTD of the SGMLDocObject at the root.

isFirst returns True if the element object is the first of its kind in a list.

parent returns the element's parent object.

parent: sets the pointer to the receiver element's parent object. Used when the document is read in.

parentClass returns the class of the element's parent (as opposed to returning the parent itself, as the *parent:* method does.) This is useful because an element's context is often important when determining its behavior at runtime--often more important than its content.

Included Sample Data

STSGML includes several sample SGML files with which it has been tested. The World Wide Web home pages of the National Center for Supercomputing Applications (http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html) and the American Memory project (http://lcweb2.loc.gov/amhome.html), a federal archival project making text, sound, pictures and video available to the public, are included as samples of HTML files. The first book of John Milton's "Paradise Lost" and the first two chapters of Mark Twain's "Tom Sawyer" are also included, using data from the Text Encoding Initiative, a project spread across several universities in the United States and Britain to make classics of literature publicly available in SGML format. The DTD and document object are labeled "OTA" because this text was stored in the Oxford Text Archives at Oxford University.

Expanding STSGML

Obviously, a more sophisticated interface would allow an end-user greater access to more combinations of features and greater flexibility in how they could manipulate a document object.

Another improvement that would make it possible to easily use the system as a "server" for almost any other Windows application would be to implement a Dynamic Data Exchange (DDE) capability so that requests could be granted to running programs without user intervention. Smalltalk/V provides the classes DynamicDataExchange, DDEClient, and DDEServer to make this possible. Once this was done, it would be simple to write a macro in a word processor like Word or WordPerfect that asked for some SGML data to be formatted a certain way and then imported that into a word processing document.

Further Research

Once the most basic features are added to an SGML system to make it object-oriented, other properties of an object-oriented system can be added as needed. Candidates for these additional attributes are the kind of thing that separates one object-oriented methodology from another; for example, Booch describes a possible concurrency attribute [Boo91 pg 165] for classes that offers a choice of sequential, blocking or active process behavior. Since many large SGML systems involve simultaneous authoring and editing by multiple users, features to implement serialization of transactions would clearly be useful.

SGML and Object-Oriented Databases

Persistence [Boo91 pg 165] is another potential class property that would play an obvious role in an object-oriented SGML system, because documents and their elements obviously last beyond the execution of programs that create them. The better fit of OODBMS systems to SGML than relational systems, along with the lack of existing implementations, indicates an expansive area of potential work to be done. An obvious extension of the STSGML system would be the use of a tool like Gemstone or Versant to implement persistence with STSGML’s Smalltalk objects.

SGML Entity Management

Section 4.123 of ISO 8879 defines an entity manager as "A program (or portion of a program or a combination of programs), such as a file system or symbol table, that can maintain and provide access to multiple entities." Essentially, its job is to map logical references to entity references, so that an application dealing with a document in terms of its logical structure can still manipulate actual document instances. Charles Goldfarb takes great pains to distinguish between the abstract, or logical structure of a document, and its entity structure, or the organization of its resource storage[Gol90 93]. He did this to maintain the system independence of logical document structure specification. (For example, a single document can be made of multiple files and multiple documents can can be stored in a single file, but this should be irrelevant to the DTD’s specification of the document’s logical structure--especially when you consider operating systems that don’t store information in "files," like MVS or OS/400.) The issue of entity management is therefore outside of a discussion of the use of a document’s logical structure to automate the creation of an object-oriented system, yet still hovering close by.

A good place to start research on an object-oriented approach towards the organization of system resources would be the the field of operating systems, where the application of object-oriented technology has already played a role in the development of commercially available operating systems (NextStep, OS/400).

Entity managers are currently not that common, because so many documents have an entity structure simple enough to require minimal management. As SGML is used increasingly for applications such as multimedia that require more complicated entity structures, there is a growing need for powerful entity managers, and it promises to be a growing area of SGML software.

Appendix A: STSGML On-line Help

The following shows the on-line help included with STSGML, implemented with Smalltalk/V’s hooks to the Microsoft Windows winhelp help engine. The "Help" choices on the "STSGML" and "Formatting Text" menus each bring up the help panel explaining the choices on those menus.

STSGML

STSGML (for "Smalltalk SGML") demonstrates how an SGML document instance can be treated as an object (in the object-oriented sense of the term) and perform simple but useful work on it. It offers a menu-driven way to read in a document, assign format strings to document elements, and to then output formatted or unformatted versions to text files. Sample data files are included to convert World Wide Web pages (home pages of the National Center for Supercomputing Applications and the American Memory project) and classic literature (The first book of Milton’s "Paradise List" and the first two chapters of Mark Twain's "Tom Sawyer") marked up with the Text Encoding Initiative DTDs to RTF and TeX format.

Under the hood, STSGML declares a new object class for each element within a document and two methods for each document attribute: one to set the attribute value and one to return the attribute value.

All newly declared classes are derived from the classes SGMLObject, SGMLDocElement, and SGMLElement, which themselves have various methods declared for the new classes to inherit. These methods allow element manipulation typical of SGML applications, and are used by the various routines called by the STSGML menu choices.

Bob DuCharme May, 1995 bobducharme@acm.org

Read DTD and Document Instance

STSGML will prompt you for the name of a DTD file and a document instance file to read in as the current document object. If the whole thing is stored in one file (in other words, if the document instance file begins with its own DTD) select that filename at both prompts.

Plain Text Version to File

Selecting this prompts you for a filename and stores the current document object as a simple text file with no SGML markup in a file with that name. If there is no current document object, a dialog box reminds you to select "Read DTD and Document Instance" first.

Formatted Version to File

Selecting this prompts you for a filename and stores the current document object as a text file with the format string of each element class applied to it. To assign format strings to element objects, see the Formatting Text menu.

If there is no current document object, a dialog box reminds you to select "Read DTD and Document Instance" first.

Help

Displays this help information.

Formatting Text

Choices on this menu enable you to assign format strings to each element class. In a format string, <> (a less-than symbol followed immediately by a greater-than symbol) represents the content of the element, <attname1> represents the value for the ATTNAME1 attribute of that element, and anything else in the contents will be output literally. (Remember to put the attribute name in all upper-case letters.)

For example, if a document element class Novel had an element known as ChapTitle with the string "Setting Sail" as its content and the number 2 as the value for its chapnum attribute, then a context string of "Chapter <chapnum>: {\b <>}" for the ChapTitle element class means that this instance would be output as "Chapter 2: {\b Setting Sail}" in formatted output.

For information on reading in document instances and outputting them to files, see the STSGML help screen.

Add Format String

Selecting this prompts you for a document element class, an element name, and a format string to assign to it. To assign the format string described above, you would enter "Novel" at the first prompt, "ChapTitle" at the second, and "Chapter <chapnum>: {\b <>}" at the third.

Read Format String File

Select this to read in and execute a file of format string assignments. STSGML will prompt you for the name of the file. Each line must begin with one of the following:

; A comment. Anything on a line that has this as its first character will be ignored.

format A format string specification. This must be followed by the document element name, the element whose format string is being assigned, and the format string itself. See example below.

gsub A global substitution. This must be followed by a space, a slash (/), the target string, another slash, the replacement string, and a closing slash. For a literal slash, precede it with a backslash (\). See example below.

Examples:

```
; This file will do some substitutions to turn a TEI file into a TeX
file.
; The first format statement should go into the input file as one long
line.
```

```
format ota ota "%% loading
fonts\font\sc=cmcsc10\font\ssf=cmss10\font\ssi=cmssi10\font\hdone=cmssdc
10 scaled\magstep5\font\hdtwo=cmssdc10
scaled\magstep4\font\hdthree=cmssdc10
scaled\magstep3\font\hdfour=cmssdc10
scaled\magstep2\font\hdfive=cmssdc10
scaled\magstep1\font\hdsix=cmssdc10<>\bye "
format ota head "\par {\bf <>}\par \par "
format ota dTitle "\par {\hdfour <>}\par "


format ota byLine "\par {\it <>}\par \par "
format ota l "<>\par "
format ota edStmt, "\par <> \par "
format ota title  "\par <> \par "
format ota role  "\par <> \par "
format ota name  "\par <> \par "
format ota p   "\par <> \par "
gsub /&/\&/
```

Help

Displays this help panel.

Footnotes

[BAH93] K. Boehm, K. Aberer, and Christoph Hueser, "Extending the Scope of Document Handling: The Design of an OODBMS Application Framework for SGML Document Storage," GMD Technical Report No. 811, Darmstadt, Germany, 1993.

[BA94] K. Boehm, and K. Aberer, "An Object-Oriented Database Application for HyTime Document Storage," accepted for publication in *Proceedings of Conference on Information and Knowledge Management 1994 (CIKM94),* Gaithersburg, MD, December 1994.

[Boo91] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Redwood City, California: Benjamin/Cummings Publishing Co., 1991.

[Gol90] Charles Goldfarb. *The SGML Handbook.* Oxford: Oxford University Press, 1990.

[New93] Steven Newcomb, *The MarkMinder and HyMinder Engines* (marketing literature). Tallahassee, Florida: TechoTeacher, Inc., 1993.

[RBPEL91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Englewood Cliffs, N.J.: Prentice Hall 1991.

[RH90] B.N. Rossiter and M.A. Heather, "Strengths and Weaknesses of Database Models or Textual documents" page 137 EP90: Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography." R Furuta, editor, University Press, Cambridge, 1990, 125-138.

[Zha95] J. Zhang, "Application of OODB and SGML Techniques in Text Database: An Electronic Dictionary System," *SIGMOD Record*, (24:1), March 1995, 3-8.

Bibliography

Booch, Grady. *Object-Oriented Analysis and Design with Applications.* Redwood City, California: Benjamin/Cummings Publishing Co., 1991.

Bryan, Martin. *SGML: An Author's Guide*. Wokingham, England: Addison-Wesley Publishing Co., 1992.

Christophides, V., S. Abiteboul, S. Cluet, and M. Schol. "From Structured Documents to Novel Query Facilities" in SIGMOD 94.

Goldberg, Adele, and David Robson. *Smalltalk-80: The Language.* Reading, Massachusetts: Addison-Wesley Publishing Co., 1989.

Kim, Won, and Frederick H. Lochovsky (ed.) *Object-Oriented Concepts, Databases, and Applications.* New York: ACM Press, 1989.

Goldfarb, Charles. *The SGML Handbook.* Oxford: Oxford University Press, 1993.

ibid. *Entity Management in SGML* unpublished, 1993.

Rumbaugh, James, and Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Englewood Cliffs, N.J.: Prentice Hall 1991.

Softquad, Inc. *The SGML Primer*. Toronto, Canada: Softquad, 1990.

Text Encoding Initiative. *A Gentle Introduction to SGML.* http://etext.virginia.edu/bin/tei-tocs?div=DIV1&id=SG University of Virginia.

Wilkie, George. *Object-Oriented Software Engineering: The Professional Developer's Guide*. Wokingham, England: Addison-Wesley, 1993.