An Unofficial Introduction To Groves

Eugene Eric Kim <eekim@eekim.com>

Version 0.1
April 5, 2001

Introduction 1 (01)

This paper is my attempt to clarify what groves and property sets are. There are a number of excellent papers on groves, several of which I mention in the References section below. However, these collection of papers lack simple, concrete examples, comparisons with competing data modeling languages, and the rationale behind many aspects of the groves design. The primary purpose of this paper is not to rehash what has already been said by others (although there is a good bit of that here), but to address that which has not already been said. In doing so, I hope I can shed some light on groves. 1A (02)

This paper is a work in progress, chockful of personal commentary. There are very likely inaccuracies as a result of my limited understanding of certain concepts. I would greatly appreciate feedback, clarifications, and corrections. 1B (03)

What Are Data Models? 2 (04)

Groves are usually described as a data modeling language, and so a good way to start explaining groves is to describe what a data model is. Data models are representations of data. They provide a way to think about or view data. 2A (05)

Consider, for example, an employee in a company. What are the different ways we can represent an employee? We can start by thinking about the characteristics an employee has, things like a name, a salary, a starting date, a boss, and so on. We can also think about employees as people, separating the people-specific characteristics -- name, age, gender, etc. -- from the employee-specific ones -- salary, starting data, etc. 2B (06)

All of the different ways of thinking about an employee constitute data models, and there are a number of different languages that allow us to express these abstractions. Suppose we wanted to think about an employee as a person with a salary, and a person as an entity with a name and an age. Example 1 shows one way of expressing this data model. 2C (07)

```
class Person {
  string name;
  int age;
}

class Employee : Person {
  float salary;
}
```

Example 1. A data model for an Employee. An Employee is a Person with a Salary. 2D (08)

Figure 1 expresses this same data model graphically. 2E (09)

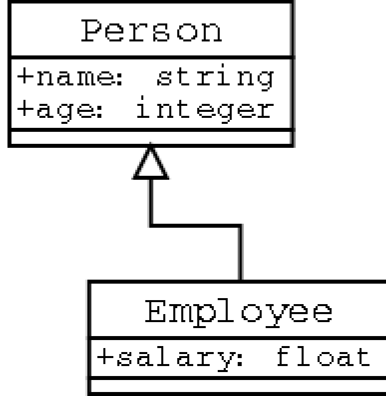

Figure 1. UML diagram for the Employee data model in Example 1. 2F (010)

Both Example 1 and Figure 1 should look familiar to computer programmers, and for good reason. One of the programmer's primary jobs is data modeling -- figuring out a way to represent data so that the software can manipulate it appropriately, and then expressing that data abstraction in a programming language. All programming languages, in fact, are data modeling languages, since they provide a way to express data models. 2G (011)

What generally differentiates one data modeling language from another is intent and the level of abstraction.[1] The intent of a programming language is for the expression of a data model (i.e. source code) to be translated into a machine-readable representation of the data. As a result, the data modeling part of a programming language tends to be close to the machine, describing low-level structures in a way that sometimes obscures the higher-level meaning of a data model. 2H (012)

Towards the opposite end of the extreme, you have data modeling languages such as E/R and UML diagrams, which are more human-centered and present a higher-level abstraction of the data. The cost of these abstractions is the lack of information that may be valuable in implementing these data models. For example, E/R diagrams allow you to specify the cardinality of a relationship (i.e. one-to-n, n-to-one, n-to-n), but they do not allow you to express other kinds of constraints, and they have no notion of primitive data types. Consequently, they are not adequate to generate equivalent source code automatically and completely, although they could be used to generate stub code. 2I (013)

Groves And Addressability 3 (014)

As a data modeling language, groves fall between the two extremes, leaning a bit towards the lower-level languages. The intent of groves is to enable addressability of data -- in other words, provide the capability to describe unequivocally the location of a chunk of data. 3A (015)

What exactly does this mean? At the lowest level, all data represented by a data model is addressable at any granularity, since software can address and retrieve any chunk of data stored in an internal data structure. The value of groves is that it provides an explicit way of specifying the smallest units of data that can be addressed by an application and a navigational scheme for describing the location of this data. 3B (016)

Literally, a grove is a Graph Representation Of property ValuEs. Groves provide constructs for describing directed graphs consisting of nodes, which have properties, and arcs, which connect the nodes. The basic idea is that any data, especially structured documents, can be modeled using directed graphs. A tree structure is a type of directed graph, and anyone familiar with parsing structured documents, be they XML documents or source code, will understand the utility of tree structures for representing those documents. 3C (017)

In order to describe the structure of a grove, you need a data modeling language. Property Sets are that language. They consist of an SGML DTD used to define the schema for a grove. I won't describe the property set syntax in detail here, as the ISO specification does a good job of documenting it. 3D (018)

Property sets let you define nodes, which essentially are a set of properties. These properties have names and a data type. Data types can either be primitive, familiar types such as strings and characters and integers, or they can be nodal -- another node, a list of nodes, or a node map (known

in property set parlance as a named node list). Represented graphically, a node and its nodal properties are connected to each other by an arc, with the direction of the arc pointing towards the property. 3E (019)

All groves have one, and only one, root node, which is the starting point of the data model. All other nodes in a grove are properties of some other node. We call the parent node the origin, and every node in a grove except the root node has an origin. 3F (020)

Consider the following example. Suppose you want an abstract data structure that models the outline document in Example 2. 3G (021)

```
Paper on Groves

+ Introduction
+ What are groves?
    + Graphs
    + Property Sets
+ Conclusion
```

Example 2. A sample document containing an outline. 3H (022)

You could represent this document as a tree of nodes, with each node representing an individual point, and each point containing a "text" property that contains the text of the point, as shown in Figure 2. 3I (023)

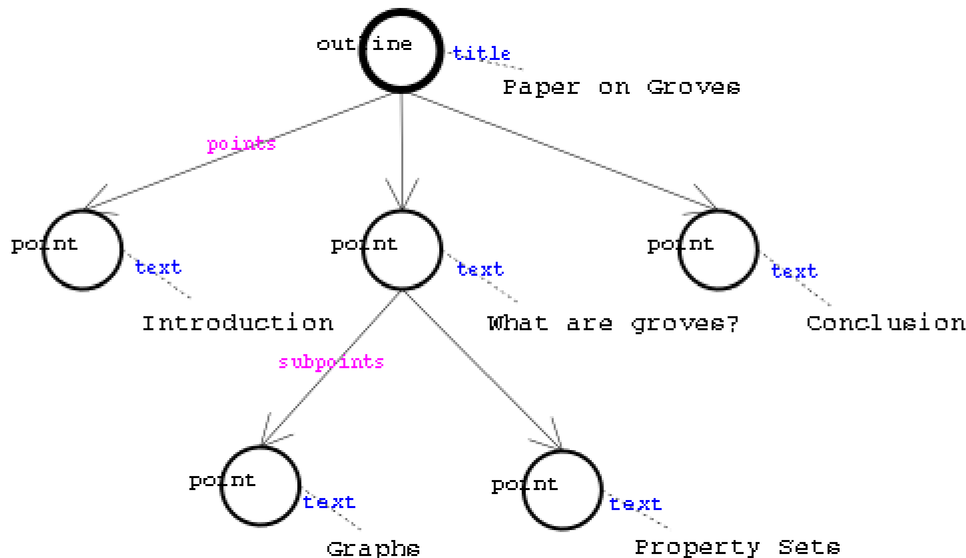

Figure 2. A grove for the outline document in Example 2. 3J (024)

In order to describe the schema for this grove, we need to define two types of nodes. The first is the root node, which we can call the "outline" node. This node has two properties: a title property, which is a string type, and a list of points. We will represent points as a second type of node, also with two properties: the text contained by the point, which is also of type string, and a list of subpoints, which also consists of point nodes. The corresponding property set might look like Example 3. 3K (025)

```
<propset>
  <desc>Defines the classes and properties to be used for
    constructing outline groves.</desc>
  <classdef rcsnm="outline">
    <propdef rcsnm="title" datatype="string"/>
    <propdef rcsnm="points" ac="point" datatype="nodelist"
      noderel="subnode"/>
  </classdef>
  <classdef rcsnm="point">
    <propdef rcsnm="text" datatype="string"/>
    <propdef rcsnm="subpoints" ac="point" datatype="nodelist"
      noderel="subnode"/>
  </classdef>
</propset>
```

Example 3. A property set describing a grove schema for an outline document, such as the one shown in Example 2. Applying this schema to Example 2 gives the grove shown in Figure 2. *3L* (026)

This grove provides a way to address chunks of the data based on the document's structure. If you wanted to refer to the point entitled "Introduction", you could say, "Go to the first subpoint of the outline." Or, if you wanted to refer to the point entitled "Property Sets", you could say, "Go to the second subpoint of the outline's second subpoint." *3M* (027)

Careful observers will note that the property set in Example 3 looks suspiciously like a class schema used in object-oriented application development. The element for defining nodes is even called "classdef", rather than "nodedef". I'm sure that this was a deliberate decision, although I think that it was an unfortunate one, as I'll explain in a second. However, note for now that a property set could easily be used to generate a set of class stubs in any object-oriented programming language. *3N* (028)

With groves, you cannot address more granularly than a node. This means that in the case of this particular outline grove, you can only address an entire point, not a single word or character in the text of a point. You might view this constraint as somewhat disingenuous, as once the grove is represented in an internal data structure, the software will be able to address a single character in the text of a point, not just the text as a whole. However, the point of this constraint is not to represent accurately what is and is not possible in software, but to restrict how data is addressed. *3O* (029)

Would you ever want to limit the addressable granularity of a text document to a paragraph, for example, rather than a sentence or a word or even a single character? Probably not. However, you may want to have multiple addressing schemes for a text file, one of which addresses each paragraph while another addresses individual characters. In order to support the former addressing scheme, you need two different data models: one that specifies paragraphs as the most granular chunk of data in a document, and another that specifies characters. *3P* (030)

Plain Text Groves *4* (031)

Examining two different data models for a raw ASCII text file will reinforce this point. Consider the text file shown in Example 4: *4A* (032)

```
First line.
Second line.
Third line.
```

Example 4. A sample text document. *4B* (033)

One way to model this document is to think about each line as a string and the document as a list of lines. Modeling it as such would limit the addressable granularity of the document to a single line. This restriction is fine for a number of applications, such as the line-oriented editor ed. Figure 3 shows what the corresponding grove might look like. *4C* (034)

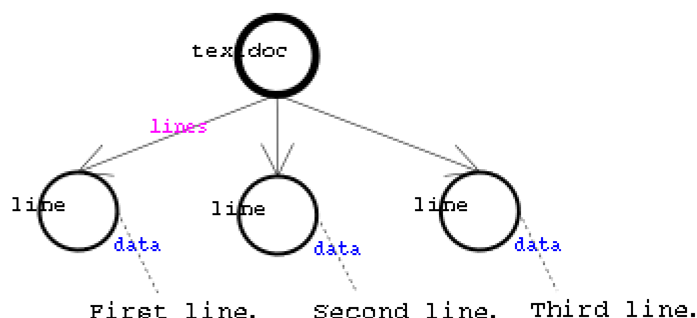

Figure 3. A line-oriented grove for the document in Example 4. *4D* (035)

Example 5 shows the property set for this grove: *4E* (036)

```
<propset>
```

```
  <desc>A list of lists of strings, representing a text
    document.</desc>
  <classdef rcsnm="textdoc">
    <propdef rcsnm="lines" ac="line" datatype="nodelist"
      noderel="subnode"/>
  </classdef>
  <classdef rcsnm="line">
    <propdef rcsnm="data" datatype="string"/>
  </classdef>
</propset>
```

Example 5. A property set corresponding to the grove depicted in Figure 3. *4F* (037)

Note that I did not use the primitive datatype strlist for the lines property. If I had, you would not be able to address individual lines. Instead, I defined a new node, called "line", which contains the value of the line in a property called "data". *4G* (038)

Suppose that you wanted to develop a more sophisticated editor, such as vi, that required character-level addressability. You can modify the property set above by defining a new node, representing an individual character, and changing the "data" property of a line from a string to a nodelist of character nodes. Figure 4 shows the corresponding grove: *4H* (039)

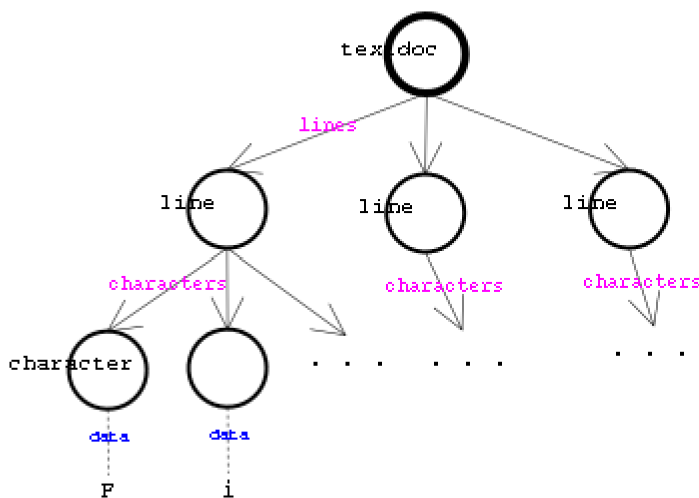

Figure 4. A line-oriented grove with addressable characters. *4I* (040)

Example 6 shows the new property set for this grove: *4J* (041)

```
<propset>
  <desc>A list of lists of characters, representing a text
    document.</desc>
  <classdef rcsnm="textdoc">
    <propdef rcsnm="lines" ac="line" datatype="nodelist"
      noderel="subnode"/>
  </classdef>
  <classdef rcsnm="line">
    <propdef rcsnm="characters" ac="character" datatype="nodelist"
      noderel="subnode"/>
  </classdef>
  <classdef rcsnm="character">
    <propdef rcsnm="data" datatype="char"/>
  </classdef>
</propset>
```

Example 6. The property set corresponding to the grove in Figure 4. *4K* (042)

Given this data model, you can now address individual characters in the text document. For instance, if you wanted to address the "c" in the word "Second", you could say, "Go to the third character in the second line of the document." *4L* (043)

However, this is not the only possible scheme for representing a text document as individual characters. Instead of viewing the document as a list of lists of characters, you could view it as one

giant list of characters, as shown in Figure 5. 4M (044)

An alternative data model for the text document in Example 4 is to view the document as a single list of characters, as shown in Figure 4. 4N (045)

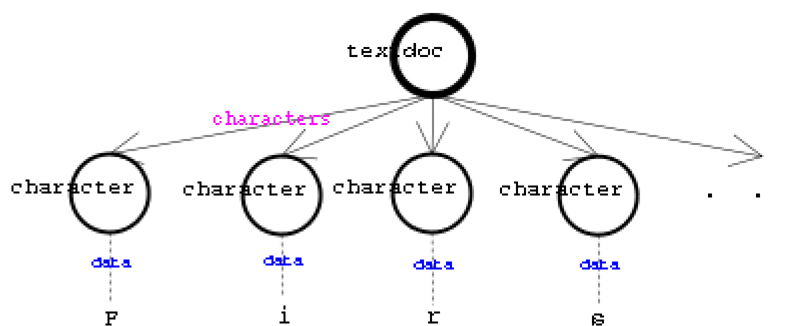

Figure 5. A "single list of characters" grove for the document in Example 4. 4O (046)

Example 7 shows the corresponding property set: 4P (047)

```
<propset>
  <desc>A list of characters, representing a text document.</desc>
  <classdef rcsnm="textdoc">
    <propdef rcsnm="characters" ac="character" datatype="nodelist"
      noderel="subnode"/>
  </classdef>
  <classdef rcsnm="character">
    <propdef rcsnm="data" datatype="char"/>
  </classdef>
</propset>
```

Example 7. Property set corresponding to the grove in Figure 5. 4Q (048)

You can still address single characters, but the addressing scheme has changed. In this example, you can no longer specify the third character of the second line, because there is no concept of lines in the data model. Instead, if you wanted to address the "c" in "Second", you would specify the 15th character in the document (assuming newlines were included as characters). 4R (049)

Groves As Views 5 (050)

An important observation about data models is that they provide ways to view the data. In this sense, a data model does not necessarily need to reflect how the system actually stores the data, although this certainly represents one way to view the data. 5A (051)

Earlier, I noted that programming languages model data at a very low level, primarily for machine representation. I also observed that property sets could easily be mapped to classes in several different programming languages. While this is true, and may in some cases even be desirable, this is not the primary purpose of groves. Groves are not a data specification language, but a language for denoting addressability of data. 5B (052)

This is why I expressed discomfort over the use of the name "classdef" to define nodes. Not only is it an internally inconsistent naming scheme (i.e. if they're called nodes, then it seems to make more sense to name the definition elements "nodedef"), it implies that property sets are a data specification language. I think it can be conceptually useful to think about them this way, but I also think it can be dangerously misleading. 5C (053)

Consider Examples 6 and 7, the two character-level data models for text documents. Both of these property sets could be mapped directly into an application's data structures. However, is this desirable? Would you want to create a "character" class, whose only field is a variable of type "char"? Perhaps in some cases, but if you're concerned about overhead, probably not.[2] It would make more sense to have your line class (in the case of Example 6) or document class (in the case of Example 7) contain a variable consisting of a sequence of character primitives (or possibly a string, if the programming

language supports that primitive), rather than a sequence of character objects corresponding to some character class. 5D (054)

Additionally, the addressing scheme implied by the grove in Example 7 does not require the data structure also implied by the grove. You could just as easily support such an addressing scheme with the data structure implied by Example 6. In order for multiple applications to behave exactly the same way when it comes to addressing, they must all agree on a common underlying data model. This does not mean that all applications must use the same internal data structures or the same data storage scheme. Data models are views that are largely independent of how the data is stored, and you can support more than one data model in a single application. 5E (055)

The primary purpose of the node abstraction is not to specify what the classes in your software should be, but to designate an addressable chunk of data in your document. 5F (056)

Groves have several mechanisms for supporting multiple views within a single application. Grove plans allow you to specify which nodes in a property set are important for a particular application, and which ones you can safely ignore. For example, an XML processing application might want to be able to address processing instruction nodes, whereas XLink may care less about processing instructions. A grove plan allows you to say, for this application, don't bother populating these nodes. 5G (057)

Another view that groves support are content tree views. Frankly, I'm not sure what the purpose of content trees is, but I'll explain what I do know. Property sets allow you to specify a single property within a node as the content property. This property must be either a string, a char, or a subnode, and nodes are not required to specify a content property.[3] Assuming that some properties are designated as content properties, another way you can view a grove is through a content tree view, which consists of multiple, possibly disjoint trees of content properties throughout the grove. 5H (058)

More Grove Examples 6 (059)

While groves were created to model documents, they are general enough to model most, if not all, types of data. For instance, it is quite easy to model relational databases using groves. 6A (060)

Consider an EMPLOYEE table in a relational database, with NAME as the primary key. (That's right; no duplicate names in this company!) 6B (061)

```
NAME          AGE    SALARY
Joe Schmoe    25     50,000
Jane Doe      30     75,000
```

Example 8. The EMPLOYEE table. 6C (062)

You can represent each row as a node, assigning a property to each column field. To represent the table, you can use a named node list to store all of the row nodes, with the NAME property acting as the key to the named node list. The property set would look like Example 9. 6D (063)

```
<propset>
  <desc>A grove for the EMPLOYEE relation.</desc>
  <classdef rcsnm="EMPLOYEE">
    <propdef rcsnm="rows" ac="row" acnmprop="NAME"
      datatype="nmndlist" noderel="subnode"/>
  </classdef>
  <classdef rcsnm="row">
    <propdef rcsnm="NAME" datatype="string"/>
    <propdef rcsnm="AGE" datatype="integer"/>
    <propdef rcsnm="SALARY" datatype="integer"/>
  </classdef>
</propset>
```

Example 9. Property set corresponding to the EMPLOYEE table. 6E (064)

Note that this property set could easily be used as a schema for creating database tables. A property set is simply a schema language for describing data models, whereas a database schema describes a

relational data model. In other words, the latter is a subset of the former. 6F (065)

If you want to be really cute, you can probably model a grove with a grove. A node's properties would consist of a named node list containing property nodes, which in turn would contain name and datatype properties, among others. I'll leave the actual construction of such a property set as an exercise for the reader. 6G (066)

Irefs, Urefs, Links, and Transclusions 7 (067)

On more than one occasion, HyTime guru W. Eliot Kimber has described groves as "roughly trees," which I think is a wonderful characterization. Technically, groves are directed graphs, but in reality, they are trees with some extensions that allow you to construct more complex structures. These constructs include internal references (irefs) and unrestricted references (urefs). 7A (068)

Every nodal property type (node, nodelist, nmndlist) has an attribute called noderel, which specifies the relationship between the nodes. The most common noderel is subnode, which specifies a parent/child relationship between the nodes -- in other words, a tree. 7B (069)

To specify a more general, nontree relationship between the nodes, you use the noderel attribute "iref". Using irefs, you can connect a node to an already existing node in the grove, as shown in Figure 6. 7C (070)

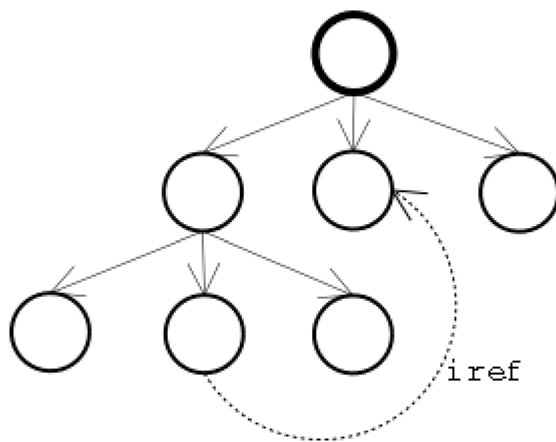

Figure 6. Internal references in a grove. 7D (071)

Remember that all nodes contain an origin, an intrinsic property that stores a node's parent node. Irefs can only be made to nodes that already exist in the grove, and they do not change the origin property of a node.[4] As a result, all groves can be expressed as trees, with irefs interconnecting nodes throughout the tree. 7E (072)

Urefs, or unrestricted references, connect nodes between different groves. One or more groves connected via urefs are called hypergroves. 7F (073)

Both irefs and urefs can be considered links, although the semantics of these linking mechanisms are not necessarily the same as those we are familiar with on the World Wide Web today. Urefs, in particular, seem analogous to transclusions, Ted Nelson's term for links that include content from other documents. 7G (074)

One instance where you might use a uref is in creating a data model for e-mail. E-mail can act as an envelope for other document types -- in other words, MIME attachments. You want to be able to address the standard content in an e-mail -- headers and body text -- as well as the content contained in the MIME attachments. One possible solution would be to represent each MIME attachment with their own grove, depending on their MIME type, and creating a "MIMEattachment" property in the e-mail grove, which would be a uref to the root nodes of these other groves. 7H (075)

Conclusion 8 (076)

Groves are meant to describe data models, specifically to enable addressability. They are primarily for human consumption, although they could easily be used to generate machine-readable data structures. The only constraint a property set places on a software's internal data structure is that the data structure must be able to support the grove's addressing scheme. Beyond that, the data structure does not have to look anything like the structure implied by a document's property set. 8A (077)

The rule of thumb when constructing a property set for a document is, every addressable unit in a document should be contained in a node. A document can have several different property sets to support multiple addressing schemes. 8B (078)

Finally, groves are not the end-all-and-be-all of data modeling languages. There will certainly be situations when other modeling languages will be better suited for a task than groves. 8C (079)

Notes 9 (080)

[1] Douglas Schenck and Peter Wilson distinguish data modeling from information modeling in their book, *Information Modeling the EXPRESS Way*: 9A (081)

"Information modeling is an outgrowth of data modeling and the question of classifying a model as being an information or a data model can be somewhat fuzzy. Data modeling is concerned with specifying the appearance and structure within a computer system of the data which represents particular types of information. Information modeling, as we implied earlier, has a goal of describing information so that the representative data *could* be computer processed. Note that this does not require that it be processed or even that it should be so processed. Thus, one distinction between data and information modeling is that one is explicitly targeted for computer processing of data while the other has the potential (which may, of course, be realized) for such processing. 9B (082)

The other major distinction is in the treatment of the interpretation rules. In an information model these must be made explicit and formally documented. In a data model, the rules are typically implicit; even if they are made explicit, they are informally documented." (Schenck and Wilson 10-11) 9C (083)

By Schenck and Wilson's estimation, languages like UML would be an information modeling language, while groves would be a data modeling language. While you could, for example, represent the notion of an "is a" relationship in a grove, that representation would be implicit. 9D (084)

For the purposes of this paper, I classify modeling languages on both ends of this spectrum as data modeling languages. 9E (085)

[2] There are good reasons for wrapping a language's primitive types in an object. Java, for example, has object wrappers for all of its primitive types so that its generic container classes, such as the vector class, can store any type of data. (Horstmann and Cornell 175) 9F (086)

[3] The various quirks regarding primitive types for groves seem further proof that groves were not intended to be a data specification language. First, the fact that only string, char, and subnode properties can be designated "content properties" seems to indicate that the addressable content of documents should be represented as strings, whether or not that is what they are. This makes some sense. If you are going to apply a style to a node's property for display, you'll probably want to treat that property as a string, even if it's an integer or some other non-character datatype. 9G (087)

Second, floating points are a notably missing datatype. You would think that floating points would be useful for modeling certain types of data. Again, if all you're concerned about is addressing data, not how that data is represented internally, maybe this is not a problem. However, if this were the case, you probably wouldn't need an integer primitive either. Yet groves have an integer primitive. 9H (088)

As a side note, the following grove primitives are a mystery to me: intlist, strlist, enum, compname, and cnmlist. When would you ever need an intlist, strlist, or a cnmlist, as opposed to a nodelist consisting of nodes with these respective primitives? Why do you need an enum, when nodes seem to accomplish the same thing? And are component names really worthy of their own primitives? 9I (089)

[4] You'll notice that I have several gripes about grove naming conventions. "Origin" seems an appropriate name if you're describing a directed graph, because nodes in a directed graph do not necessarily have a parent/child relationship. However, the origin intrinsic property of a node grove actually refers to the parent node. Calling it an "origin" would make more sense if its value actually changed in an iref relationship, but because it doesn't, "parent" seems to be a better name. *9J* (090)

References *10* (091)

In my opinion, the three best references on groves are Paul Prescod's paper, "Addressing the Enterprise: Why the Web Needs Groves", the "Property Sets and Groves" excerpt from W. Eliot Kimber's presentation, "Practical Hypermedia: An Introduction to HyTime" at the HyTime '96 conference in Seattle, and the Property Set Definition Requirements section of the HyTime standard (ISO/IEC JTC1/SC18/WG8). *10A* (092)

The references cited in this paper are: *10B* (093)

Horstmann, Cay S. and Gary Cornell. *Core Java 1.1: Volume 1 -- Fundamentals*. Mountain View, CA: Sun Microsystems Press, 1998. *10C* (094)

Schenck, Douglas and Peter Wilson. *Information Modeling the EXPRESS Way*. New York, NY: Oxford University Press, 1994. *10D* (095)

Document developed using Purple