COSC 3311 Software Design
Report 2: XML Translation
On the Design of the System

Gunnar Gotshalks

1 Introduction

This document describes the design work and testing done in completing the tasks as specified in the specification "Report 2: XML Translation", which can be found at the following URL by following the link for the report 2 specification in the September 16 bulletin.

http://www.cs.yorku.ca/course_archive/2004-05/F/3311/

The initial program text that was given to students in the following file in the course directory.

prism:/cs/dept/course/2004-05/F/3311/report2.tar

2 XML-file representation classes

Report 1 specification had a concluding section that described that the classes TERM and VARIABLE were determined from the grammar rules describing a polynomial. In the specification for Report 2 we are given two grammars. One grammar describes the structure of the input file that represents XML files for the purposes of the Report. Another grammar describes the tagged structure of an XML file. At first glance it appears that the second grammar most closely represents the inherent structure of an XML file but on closer examination objects in the first grammar appear in the second grammar. It was decided to use both grammars in determining the classes to be developed

The following are the candidate classes for representing an XML structure.

File, Section, SectionHeader, TaggedComponent, Paragraph, BulletedList, ListItem, Text, Word, Space, GraphicCharacter, Tag, TagName, Line

2.1 The classes to keep

The following list shows the sub-list of candidate classes that it was decided to keep.

1. File – renamed as XML_file – represents the sequence of sections in an XML file; it is the root of the internal representation.
2. Section – represents the list of components in a section, including the section header.
3. TaggedComponent – represents the parent type for all tagged components; i.e. components that begin with a tag such as a bulleted list.
4. Paragraph – represents the list of components in a paragraph
5. BulletedList – represents the list of components in bulleted list
6. ListItem – represents the list of components in a list item within a bulleted list.
7. Text – Represents the parent type of user data in an XML file such as Space and Word
8. Word – Represents a word in the user data.
9. Space – Represents spacing in the user data.

2.2 The classes to reject

The following shows the sub-list of candidate classes that it was decided to reject.

1. SectionHeader – is replaced by Text, no other attributes are needed at this time.
2. Tag – Each tag with a name not beginning with "/" corresponds to a class derived from a grammar rule that has a start tag as their first component. The objects of interest are not the tags themselves but the objects that are denoted by the presence of a tag. The end tags, such as, </SECTION> and

</UL>, are represented by the termination of the list of the components that appear, in the grammar rules, between the start and end tags. Tag may become important when tags have their own structure but at this point tags do not have sufficient structure to warrant being represented by a class. Tags are represented by attributes within the classes representing tagged components.

3. GraphicCharacter – is replaced by the Character class in Eiffel with constraints on the values to not be whitespace or control characters. An alternative would be to create a class that inherits from Character and adds additional features to distinguish additional subsets of characters that the class CHARACTER does not distinguish but that is too general for the purposes of this report. This could be a future enhancement.
4. Line – Is an artifact of the input ASCII file structure, primarily a restriction on what human readable files should look like. It is not an inherent property of the meaning of an XML structure.

2.3 The inheritance structure

The selected classes have the inheritance structure shown in Figure 1. The selected classes TEXT and TAGGED_COMPONENT are deferred, as they are not leaf level components that contain user data. The classes TEXT_OR_TAGGED and TXT_TAGGED_LI arise from the grammar rules for Section, BulletedList and ListItem, which have sequences of items from the choice of the sub-classes suggested by their names. The class WITH_CHILDREN distinguishes between XML-file components that have sub-components (children) and those that do not. The class XML_COMPONENT is the parent of all the XML-file components.

The class NEWLINE is introduced because of the constraint to create an internal representation that can be used to output an exact duplicate of the original input file. An instance of NEWLINE is not a tag, it is a character, and as a consequence it is a subclass of TEXT. It turns out that the class NEWLINE is also required for the Postscript translator as NEWLINE is frequently translated to be a space character needed to separate two words in the translated output.

The class XML_FILE does have children, the list of its sections, but it is not of type WITH_CHILDREN because it does not have tags. If for example, the structure was closer to that of an HTML file, then the entire file would be enclosed in the pair of tags <HTML> and </HTML>, in which case XML_FILE would inherit from WITH_CHILDREN.

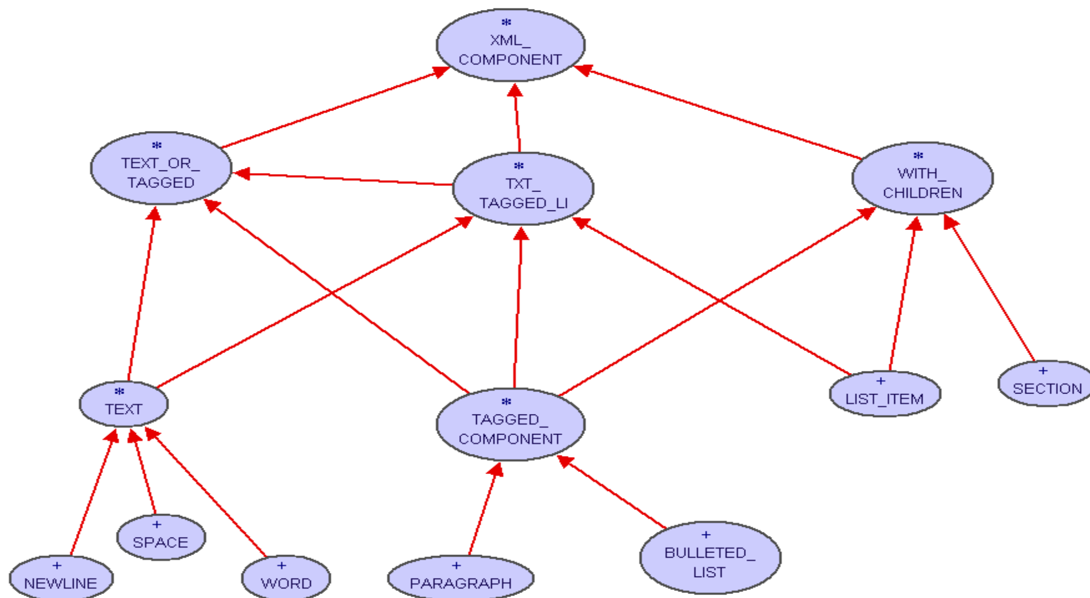

Figure 1: Inheritance structure of the XML-file components

2.3 Representation features

The class XML_COMPONENT is too general to contain common features. The class exists to be able to reference any class from an appropriate list of sub-types.

The class WITH_CHILDREN contains all the representational features required for all XML-file components that have sub-components. The features are the following.

> `the_start_tag : STRING` — An exact copy of the start tag in the input file. While the tags are listed in upper case in the grammar rules, experience shows that tags are usually accepted in any case including mixed due to the realization that for some files tags are typed by people and they do no like having meaningless restrictions such as upper case only. No useful grammar would distinguish between SECTION, section and Section, it would be better to have different characters in the names to distinguish different tags, than relying on case sensitivity.
>
> `the_end_tag : STRING` — An exact copy of the end tag in the input file.
>
> `components : LINKED_LIST[XML_COMPONENT]` — The list of all the sub-components.

The classes PARAGRAPH, BULLETED_LIST and LIST_ITEM do not need more features to represent their components but they do redefine the lists to have, respectively, the component types TEXT, TXT_TAGGED_LI, and TEXT_OR_TAGGED. The class SECTION needs two lists. The main list is the list of components consisting of the words and spaces in the body of the section; this list is inherited from WITH_CHILDREN, with the components redefined to be of type TEXT_OR_TAGGED. The secondary list is the list of words and spaces in the section header, so SECTION has the following additional representational feature.

```
section_header : LINKED_LIST[TEXT]
```

The class TEXT introduces the following feature, as the only representational feature needed for the classes NEWLINE, SPACE and TEXT

```
component : STRING
```

The class XML_FILE has the following representational feature.

```
components : LINKED_LIST[SECTION]
```

Figure 2 shows the client-suppler (uses) structure for the XML-file components. The main point of interest is that components with children represent the children as a linked list of components of a deferred type corresponding to the choices in the grammar rule defining the component. Neither the user data – NEWLINE, WORD and SPACE – nor the tagged components are used directly, because they appear as references in the appropriate list of components of a parent class type.

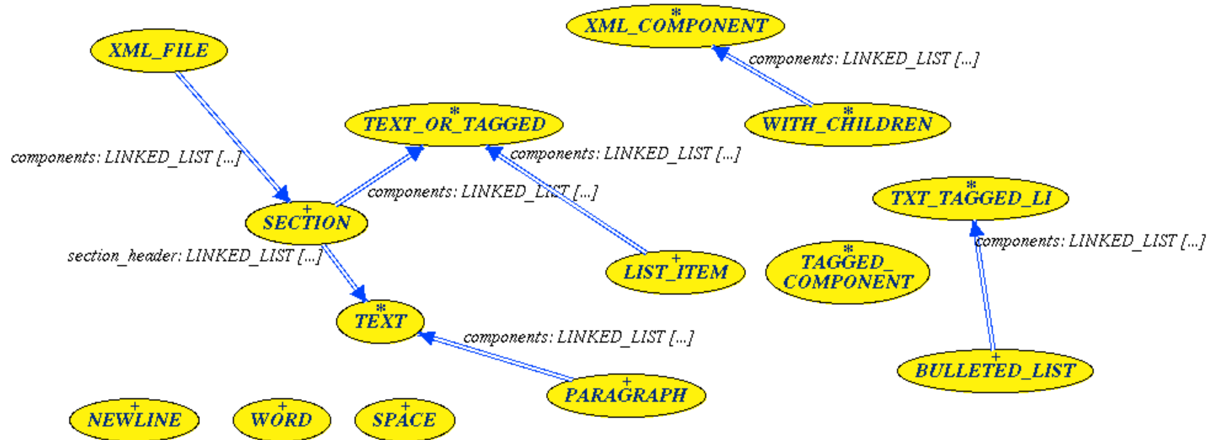

Figure 2: The Uses structure of XML-file components

2.4 Discussion

Just adding new tags is insufficient to define an extension to the XML structure because a tag tells us nothing about the structure for the corresponding tagged component and its relationship with other components. With the constraint to create an internal representation, adding more tags is inherently a difficult task because introducing more tags implies the following changes are required.

- Creating at least one grammar rule for each tag to define its structure
- Changing the scanner to recognize the new grammar rules
- Adding new classes for the new tag structures
- Changing the inheritance structure to include the new classes

There is no other way to solve the problem.

3 Transforming ASCII input into internal representation

The design of the sub-system responsible for creating the internal representation from an external file is based on structured design. In this case the transformation of the external representation to the internal representation is done using two communicating processes as shown in Figure 3, where the following describes the objects.

In – the input channel consisting of physical lines of an ASCII XML file.
Out – the output channel consisting of the objects in the internal representation.
M – a communication channel along which XML components are transmitted.
Scanner – a process to transform the ASCII XML file to a sequence of XML components.
CreateRep – a process to transform the sequence of XML components into the collection of internal objects.

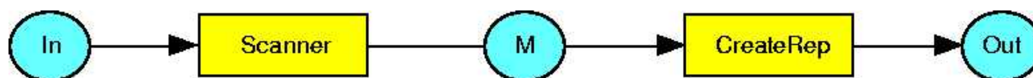

Figure 3: Processes to transform the input representation to the internal representation

3.1 Scanner process

The Scanner, XML_SCANNER, is a separate ADT (a separate process) that is an expert on the input structure of XML files. The scanner transforms the input sequence of characters corresponding to the ASCII view of and XML file to an intermediate representation consisting of a sequence of tokens in the following feature where a token consists of either a word, space, newline, or tag.

```
input_components : LINKED_LIST[XML_COMPONENT]
```

Tags are represented by instances of SECTION and TAGGED_COMPONENT corresponding to the tag name, including end tags. Start and end tags are distinguished by whether `the_start_tag` or `the_end_tag` attribute is not void. No other representational features are used for the intermediate representation.

The structure of the transformation process corresponds closely to the structure of the XML file as defined by the ASCII-view grammar as follows.

`scan_input` corresponds to the grammar rule File to process a sequence of lines.

`process-a_line` corresponds to the grammar rule Line to process a sequence of components in a line.

`extract_component` corresponds to the choice in the Line rule. It uses the first character of a token to distinguish between text, tag and space.

`extract_word` corresponds to the grammar rule Word to process a sequence of graphic characters.

`extract_space` corresponds to the grammar rule Space to process a sequence space characters.

`extract_tag_or_word` corresponds to the grammar rules for Tag and Word to process a tag but since an invalid tag is considered to be a word the choice remains until a tag is confirmed or denied by the input structure.

`create_tag_from` corresponds to the grammar rule TagName to verify that a tag structure exists and to create it. If the tag structure does not exist then no component is returned.

The correspondence of the process with the grammar is not exact to make the process shorter and easier to write. An exact correspondence would make the process more complex than necessary considering the assumption that there are no errors in the input. Upon further consideration it can be seen that whether or not tags and text are on separate lines is immaterial to the internal representation, since the internal representation does not depend upon that input structure. Finally, it is unlikely that the restriction that tags are on separate lines would exist in real files. That can be seen in HTML files as suggested in the specifications and that the in the original input example the section header had the section start tag and the header text all on one line and the list items have the tag and text on one line, indicating that it is probably not a good design decision to enforce that requirement.

The process as written is more flexible than the specified grammar and will accept a wider range of input that that specified by the grammar. The disadvantage is that should error checking for the specified grammar be required, the process would have to be changed to conform to the grammar but that is highly unlikely to be a future modification for the reasons stated in the preceding paragraph.

3.2 CreateRep process

The second process transforming the input file to the internal representation is not a separate ADT. The scanner has abstracted the input file to a simplified sequence of XML components that can be trivially parsed by processing each component in sequence. As a consequence, the process can be spread across the internal representation classes, as each class is an expert on its internal representation.

The process for transforming the scanner output into the internal representation corresponds to the tagged-view grammar. Each of the leaf classes has a creation feature that makes that component from a list of XML components. The first component in the list is always the first component in the structure being built. Each creation feature extracts, from the input sequence, the components it needs to build itself and passes on a shorter list of components to its sub-components and to its successor components. Calling the `make_from` feature for the sub-component creates it; see Figure 4.

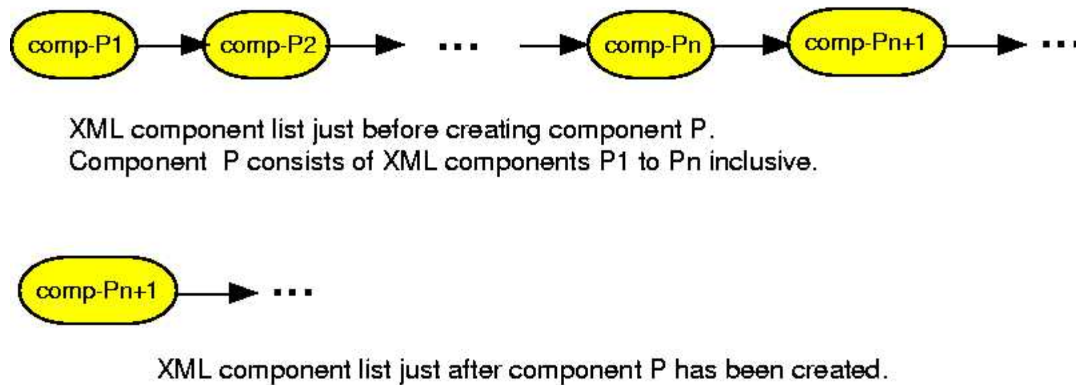

Figure 4: Schematic illustrating how a component with children processes the list of XML components.

4 The emitters

The process that outputs (emits) the translation of the XML file can be designed in two ways. Split the process across the representational classes, or have a separate process that communicates with the internal representation.

4.1 Splitting the output process

This design is based on the fact that each XML component is an expert on its structure and hence can be an expert in producing a translation that corresponds to it. This leads to a conceptually simple design by having each leaf class implement a part of the output process. In this report we have two translations so each appropriate class has two output features as follows. The class XML_COMPONENT has deferred features to ensure that every effective XML component has implemented the feature.

```
put_ascii_copy(output : PLAIN_TEXT_FILE)
put_postscript(output : PS_FILE)
```

The advantage is that each part of the output process is simple to write because the internal representation is immediately accessible.

There are two disadvantages. The first is the process is spread across many files, which makes it more difficult to see how the different parts work together than it would be in a single file. The second is that every time a new translation is required another feature has to be added to the appropriate classes. As the number of components increases the number of changed files increases.

4.2 Having separate processes

This design is based on the structured design method, whereby an input sequence of processes converts the external input representation into the internal representation, and a different output sequence of processes converts the internal representation to the external output representation.

The output process is implemented in a set of classes different from the internal representation. The output process iterates over the internal representation and as it receives XML components the output process produces the translation. To make this work the internal representation, in place of output features, exports its representation features to the output processes.

The advantages are two fold. The first is adding a new output process requires no changes to the internal representation. The second is a complete output process is implemented in one class making it easier to see how the different parts work together.

The disadvantage is that each output process implements a traversal algorithm for the internal representation but this can be overcome using the iterator pattern.

4.3 The better design

On balance having separate processes for both input and output is the better design. With such a design, the input process is an expert on the input, the output process is an expert on the output, and the internal representation is an expert on itself and does not require any knowledge of the input and output representations. The coupling between the different representations is minimized.

4.4 The PS_FILE class

The class PS_FILE was created to extend the PLAIN_TEXT_CLASS to include counters for the display line length. The following attributes are in the file.

- physical_line_length : INTEGER – The number of characters in the current output line. Used to make the physical output lines short enough to display in about 70-80 character width windows.
- physical_line_length_limit : INTEGER is 70 – The limit at which to consider creating a new physical line.
- max_char_in_line : INTEGER is 100 – Absolute maximum number of display characters on a line.
- current_max : INTEGER – Maximum characters to put on the current line – changes with indentation
- char_line_count : INTEGER -- Number of display characters in the line.
- last_was_word : BOOLEAN -- true if last TEXT item was a word. Set and reset by SPACE, NEWLINE and WORD, to avoid having multiple space output commands in a row.

The class has features to increase and decrease the display line length, set and reset the last_was_word flag, and to set the count of display characters on a line to zero.

The class provides the feature put_count_string(string, count) that outputs the string to the file and increases the display count by the parameter count. This avoids the need for the output routine to count the actual number of displayed characters in the string – a problem with escape characters in word. The caller knows the display length, so they can easily give it an argument.

5 On the Postscript output

The translation from the internal representation to Postscript was done as stated in Section 2.2.1 in the specifications with one exception. It was discovered that when an input has the sequence of tokens “Word EOL Word”, then a space needs to be output to prevent displaying the words as one word. As a consequence a space is output for newlines. This can lead to having multiple spaces in sequence. To prevent this PS_FILE as a flag `last_was_word` that is set true when a word is output, and set to false when a space, corresponding to a Space or Newline component, is output. Further, a space is only output if the last output was a word. This avoids output of multiple spaces in sequence.

The output does not look good with nested bulleted lists. The problem is that with the output of space with a newline and not permitting repeated spaces then if the first line of a bulleted item does not begin with a list item tag, then the first line is not indented. Similarly, Text immediately after an end bulleted list tag continues on the same line as the last list item. If one outputs a newline for <UL> and </UL> tags so text that follows is indented correctly, then with nested bulleted lists, then if no text appears the newlines create extra space that should not be there.

This is a common problem in programming – different conditions require different semantics. Typical practice is to use flags, as done in this program. This is a poor technique because it does not scale up – the complexity and likelihood of error goes up exponentially. The proper technique is to write a grammar to provide the proper context for all the cases, and embed into the grammar the appropriate semantics for each case. A grammar makes it easier to understand the context is at each location in the grammar, than a collection of flags.