Smalltalk XML Pull Parser

Much of the last 18 years (of 30) or so of my work has involved processing SGML and XML. By far my preferred technique for dealing with XML has been pull parsing – except when I can use XSLT or XQuery. I've got a very mature library to handle this in Java, and moving to Smalltalk it's one of the first things I missed, so I wrote a least-possible-effort XML pull parser. This work is published in the Cincom Public Store as the XMLPullParser package.

What is XML pull parsing, when and why you would use it?

An XML pull parser converts XML text into a (conceptual or actual) stream of tokens. In the most common case, those tokens are very high level, and in fact there are only 5 events: StartDocument, EndDocument, StartTag, EndTag and Text. Attributes are reported within the StartTag event as a list. Thus, the following XML:

```
<document>
  <tagA a1='att1' a2='att2'>blah</tagA>
  <tagB>aaa<tagC>bbb</tagC>ccc</tagB>
</document>
```

would be converted to the follow steam of tokens:

```
StartDocument
  StartTag{document,[]}
    StartTag{tagA,[Attribute{a1,'att1'},Attribute{a2,'att2'}]}
      Text{'blah'}
    EndTag{tagA}
    StartTag{tagB, []}
      Text{'aaa'}
      StartTag{tagC, []}
        Text{'bbb'}
      EndTag{tagC}
      Text{'ccc'}
    EndTag{tagB}
  EndTag{document}
EndDocument
```

The pull parser is like a stream in that it has a 'current' token, a 'next' method, and an 'eof' state. Note that in this example the parser isn't reporting ignorable whitespace. Usually the parser has settings so that you can control whether it returns namespace declaration attributes, handles xml:space, and other features. Sometimes the pull parser can be configured to return PIs, comments, and ignorable whitespace, and other low level events which make it suitable as the basis for driving SAX and DOM building, even to the point of allowing roundtripping.

The parser doesn't have to use objects to represent the tokens – they may simply be represented as parser state, so the token type would be represented by a constant and the token values accessed via methods on the parser. In that case you can only access the current token.

This is very similar to SAX, but with the control inverted, and as mentioned above, if the pull parser reports the appropriate level of events then it is trivial to drive SAX handlers from a pull parser.

The two attractive features of the pull parser approach are:

- Unlike a DOM, but like SAX, it doesn't store the document.
- Unlike SAX you don't have to build an FSM, either explicitly or implicitly, to process the document. It's trivial to build content processors using recursive descent patterns, especially if the parser returns real token objects that aren't recycled, and using a pull parser results in code that is easy to write and understand. A simple extension allows for LL(*) parsing.

For random access to XML, especially XML that isn't being used to represent structured data, a DOM is probably the best bet. I've often found however that using a pull parser to read the XML and create data structures better suited to my domain than a DOM is well worth the effort.

The architecture of the XMLPullParser package

There are three main parts to this package:

- The parser, consisting of two classes: `XML.PullParser` and `XML.XPPStructureException`.`PullParser` is the primary class, and you can effectively use this package knowing only this class. `XPPStructureException` is raised in response to certain methods you can call on `PullParser`.

- The events. There are five kinds of events/tokens that the parser produces: {Start/End}Document, {Start/End}Tag, and Text. These are reified as the following hierarchy:

  ```
  XML.XPPEvent
     XML.XPPStartDocument
     XML.XPPEndDocument
     XML.TagEvent
        XML.StartTag
        XML.EndTag
     XML.XPPText
  ```

  `XML.XPPEvent` and `XML.TagEvent` are abstract and never instantiated.

  The names of tags are `XML.NodeTag`, which are a triple of namespace, type (local name) and qualification, and attributes are `XML.Attribute`, which also use `NodeTag` for the name. `NodeTag` is part of VW.

- The implementation.`XML.XPPSAXDriver` is a class that you should never use directly. It currently hides an ugly secret that I'll discuss later.

- Miscellaneous.The single extension method to `XML.SAXWriter` is useful for serializing XML – you can call methods on a `SAXWriter` to produce XML and then use

  ```
  saxWriter output contents
  ```

  to access the serialized XML. `PullParser` has utility methods that allow you to send tree fragments to a `SAXWriter`.

`PullParser` API and examples

At it's core, the API is extremely simple – one class method and two instance methods on the parser, plus accessors and test methods on the event objects. There are a lot more methods than that, but they're all convenience methods layered on top of the others.

`PullParser` presents a stream abstraction – although not the full VW stream – with two methods: `current`, which gives you the current event, and `next`, which advances to the next event and returns it. You know when the end of the stream is reached because the event is a `XPPEndDocument`. There is currently no atEnd method.

The events all contain all of the obvious test methods: `isStartDocument`, `isEndDocument`, `isStartTag`, `isEndTag` and `isText`, three additional test methods `isStartTag:`, `isEndTag:` and `is:` which incorporate an event type test with a tag test, as well as the type specific accessors that you would expect. Attributes are retrieved from `XPPStartTag` events using `at:`, `at:ifNone:` and `at:ifFound:ifNone` methods, text is retreived with `text`, and the tag (name) is retrieved with `tag`. All methods that take attribute names or tag names expect either a string or `NodeTag` object, with a string being equivalent to a tag with an empty namespace (as defined in the `NodeTag` implementation).

`PullParser` replicates each of the above event methods, including the type specific accessors, and delegates them to the current event. Thus you can often process XML without any reference to event objects, simply using the parser object.

You create a `PullParser`, which is bound to the input source i.e. it's not reusable, using a class method, `parse:`, which takes any object that you could pass to `XML.XMLParser>>parse:`. Here is an example that collects all of the text in an XML document:

```
| stream |
stream := 'SomeFile.xml' asFilename readStream.
[ | xpp output |
        output := String new writeStream.
        xpp := XML.PullParser parse: stream.
        [ xpp next isEndDocument ] whileFalse: [
                xpp isText ifTrue: [ output nextPutAll: xpp text ]
        ].
        ^output contents.
] ensure: [ stream close ].
```

For each `PullParser>>is*` method there is a `PullParser>>mustBe*` method, that throws an `XPPStructureException` if the test fails.

For each `PullParser>>mustBe*` method, except `mustBeEndDocument`, there is a `PullParser>>consume*` method, that throws an `XPPStructureException` if the test fails, but also calls `PullParser>>next` on success. You can't advance past the end of the document, which is why there is no `consumeEndDocument` method.

The combination of the delegating accessors and the `is*`, `mustBe*` and `consume*` methods, form the second layer of the API – convenient but not canonical, and all implemented cleanly in terms of the core API. This is the kind of API you would expect in a recursive descent parser library.

The third and final layer of the API is built upon the second, and follows a somewhat different pattern, more along the lines if the standard `ifFalse:ifTrue` and `while:` control operators. There are six (purely conceptual) operators: `if`, `match`, `skip`, `while`, `collect` and `text` which are combined as orthogonally as possible with a number of other concepts. I'll go through each of the operators and corresponding `PullParser` methods

Operator: `match`

Method `match:peek:` throws an exception if the current event isn't a start tag matching the given name. If it succeeds, then the second (block) argument is evaluated without consuming the start tag event. The block can take zero or one argument. If it takes one argument then the current event is passed to the block.

Method `match:take:` throws an exception if the current event isn't a start tag matching the given name. If it succeeds, then the second (block) argument is evaluated after consuming the start tag event. The block can take zero or one argument. If it takes one argument then the start tag event that was consumed is passed to the block, which can be useful for processing the attributes. After the block executes the parser must be at an end tag matching the start tag. This end tag is then consumed.

Method `matchAndPeek:` throws an exception if the current event isn't a start tag. If it succeeds, then then the second (block) argument is evaluated without consuming the start tag event. The block can take zero or one argument. If it takes one argument then the current event is passed to the block.

Method `matchAnyTake:` throws an exception if the current event isn't a start tag. If it succeeds, then the second (block) argument is evaluated after consuming the start tag event. The block can take zero or one argument. If it takes one argument then the start tag event that was consumed is passed to the block, which can be useful for processing the attributes. After the block executes the parser must be at an end tag matching the start tag. This end tag is then consumed.

The orthogonal concepts in these four methods are: a) is the event is consumed before the block is evaluated (consumption meaning that the matching end tag will be also be consumed after the block is evaluated)?; and b) must the start tag match a specific name, or will any start tag do? Furthermore, any block supplied in this API can take zero or one argument, with the same meaning as above, wherever the concept is applicable. These concepts are universally applied throughout this API, and the rest of the description will assume you understand this regularity.

Operator: `if`

Methods `if:peek:`, `if:take:`, `ifAnyPeek:` and `ifAnyTake:` are conditional version of the `match*` methods. If the current event isn't a start tag, or doesn't match a supplied tag name, then the block doesn't get evaluated.

Methods `if:peek:else:`, `if:take:else:`, `ifAnyPeek:else:` and `ifAnyTake:else` add a niladic else block to the previous forms. The else block is evaluated if the first block isn't. The else block must take no arguments.

Operator: `while`

Methods `while:peek:`, `while:take:`, `whileAnyPeek:` and `whileAnyTake:` are repeating versions of the `if*` methods. They may execute zero or more times.

Methods `while:peek:separatedBy:`, `while:take:separatedBy:`, `whileAnyPeek:separatedBy:` and `whileAnyTake:separatedBy:` add another (niladic) block to the previous while methods, that is evaluated between every pair of elements. Thus this second block is evaluated one less time than the first block.

Operator: `collect`

Methods `collect:peek:`, `collect:take:`, `collectAnyPeek:` and `collectAnyTake:` are collecting versions of the `while*` methods. They return an `OrderedCollection` containing the result of the blocks evaluated. If

there are no evaluations then the collection will be empty.

Operator: `skip`

Methods `skip:` and `skipAny` throw an exception if current event isn't a start tag, or doesn't match a supplied tag name. If it succeeds then it consumes the start tag, and all events up to and including the matching end tag, taking correct account of tag nesting. Thus it doesn't stop at the first matching end tag, but rather it keeps a count of the nesting depth.

Methods `skipIf:` and `skipIfAny` are conditional versions of the previous two methods. They don't throw an exception if the current event isn't a start tag or doesn't match a supplied tag name.

Methods `skipWhile:` and `skipWhileAny` are repeating versions of the previous two methods. They may execute zero or more times.

Operator text:

Methods `textOf:` and `textOfAny` throw an exception if current event isn't a start tag, or doesn't match a supplied tag name. If it succeeds then it consumes the start tag, the (optional) text event within it and the following matching end tag. It returns the text content of the text event that was consumed, or `''` if there is no text event within the tag.

Methods `textIf:` and `textIfAny` are conditional versions of the previous two methods. They don't throw an exception if the current event isn't a start tag or doesn't match a supplied tag name, rather they return `''`

Methods `textIf:else:` and `textIfAnyElse:` operate similarly to the previous two methods but they allow you to supply a block (or any #value responder) to provide a value if the tag doesn't match. Note that a matching tag/name that is empty will still provide `''` as you would expect from XML

Implementation details

This package was written using the minimum possible effort. I looked at the existing XML parser in VW, and I can see that it could be adapted to work as a pull parser. I didn't do that however. Instead I wrote `XPPSAXDriver`, which collects all of the events produced by the SAX parser in an array, and then pretends to be a stream of events. This has two consequences:

- The entire document is parsed before any pull parser api calls are made, so parse errors occur before you would expect.
- The entire document is stored in memory, as `XPPEvent` objects, which nullifies a primary benefit of pull parsing i.e. low memory usage independent of document size.

It's worked for me so far, thus I've never fixed it. And in any case, to do so would have been premature optimization given I wrote this for use in a shipping product with known behaviour. It really should be fixed now that I'm releasing this as a general component.

Future evolution of the XMLPullParser package

Really pulling, without the SAX hack

Obviously the implementation needs to *actually* pull, probably by adapting the existing VW parser. In fact the existing parser could be refactored to be a pull parser, with all of the low level pull events, with a SAX driver built using XPP and a DOM builder built using the SAX driver. However, what I'll probably do is

extend the existing parser just enough to make it work.

I could, alternatively, use libxml2, which would give also access to RelaxNG validation, which really is useful. But that would be a lot of work, and I think more work would need to be done on an efficient DLLCC mechanism for handling String data before it would be worthwhile.

Stream composition

The parser should be refactored to be a pure stream so that existing stream composition techniques and APIs can be applied. This would also make it more obvious how to layer additional functionality via stream composition, such as backtracking via temporary token storage (required for LL(*) structure processing), and maintaining a token stack to make contextual processing simpler.