SG43 API design meeting, June 29 2017

- Meeting agenda:
  - Propose that we start at lowest level, creating a layer to wrap around xml parser / HDF5 library / JSON / etc.
  - Establish some API design standards
    - Naming conventions, etc.
  - Initial ideas for full API
    - Start small by making just enough to open a GNDS file and extract cross section info

Low-level API: so higher levels don't care whether data is stored in xml,HDF5, etc.

- Interface could be similar to XML dom parsers.
- 'node' class for each layer of the hierarchy. Methods of 'node' class include:
  - tag() -> returns the name of this node (String)
  - find(String label) -> returns 1st child node with tag() == label.
    - If no child found, should it return an empty node instance?
  - get(String attributeName) -> returns value of requested attribute
    - If attribute not found, return empty string?
  - getchildren() -> returns iterator over child nodes
  - isEmpty() -> returns boolean
  - … eventually also need methods for setting attributes, adding or deleting child elements, etc.

More detail on low-level API

- Some file-specific details are taken care of here.
  - Example: JSON doesn't directly support attributes, so they must be implemented as child elements like

```
{"reactionSuite":
   "_attrs": {
         "projectile":"n",
         "target":"Fe56",… },
   "documentations": {…}
   "styles": {…}
   …
}
```

    Low-level API needs to recognize that these are actually attributes
  - Unlike XML or JSON, HDF5 doesn't preserve order of elements. If order matters, extra metadata is needed to tell element order. API also needs to take care of that

More detail on low-level API

- How should API handle actual data?
  - i.e. what should it return when pointing to the node

    <values>1.e-5 3.47 … 2e+7 1.78</values>?
  - Propose two methods
    - getData() returns a vector of doubles
      - However, <values> can store other types of numbers (float64, float32, integer32 etc.). Should this be split up into ‘getFloat64Data()’, ‘getInteger32Data()’ etc.?
      - Also, std::vector<double> or double[]? Leaning towards std::vector
    - getText() returns the String “1.e-5 3.47 … 2e+7 1.78”
      - Needed when serializing to an ascii format like XML

More detail on low-level API

- How does low-level API open a file if multiple types are supported?

  Option A: user tells what type the file is, i.e.
  - open(String filename, String datatype):
    - returns node instance (pointing to root node in the document)
    - datatype options: “XML”, “HDF5”, etc.

  Option B: try to autodetect file type?
  - Open(String filename):
    - Return type is the same

Moving past low-level API, propose some general guidelines for API design

- Some popular conventions among colleagues at LLNL:
  - Capitalize first letter of class names
  - lowercase first letter of method names
    - use camelCase rather than underscore_names
  - For method arguments use prefix “a_”
  - For class members use prefix “m_”
  - Other suggestions?

Example of using ‘a_’ and ‘m_’ prefixes to denote types of variable:

Simplified example from GIDI (Reaction constructor):
pass in a node and a PoPs particle database,
populate class members including label, ENDF_MT, cross section and outputChannel

Prefixes show which variables are class members vs. arguments that were passed in

```
Reaction::Reaction( node const &a_node, PoPs::database const &a_pops ) :
    m_label( a_node.attribute( "label" ).value( ) ),
    m_ENDF_MT( a_node.attribute( "ENDF_MT" ).as_int( ) ),
    m_crossSection( a_node.child("crossSection"), a_pops )
{
    m_outputChannel = new OutputChannel(
        a_node.child( "outputChannel" ), a_pops );
}
```

Take a brief dive into actual API design…

- Overall design philosophy for the API is to have a class corresponding to each level in the GNDS hierarchy. i.e.
  - ReactionSuite (or Protare?)
  - Styles
  - Resonances
  - Reactions
  - Reaction
  - CrossSection
  - Distribution
  - etc.

Suggested first stab at API: just enough to navigate files and extract cross sections

class ReactionSuite

ReactionSuite(node a_root, node a_pops) <- constructor, see example on slide 7

getStyles() <- returns Styles instance (then iterate over it to find all available styles)

getStyle(String a_label) <- returns Style instance (or pointer?) with desired label

getReaction(String label) <- returns Reaction instance by label

getReaction(int MT) <- users will demand this, so let's provide

Suggested first stab at API: just enough to navigate files and extract cross sections

class Style // note that several other classes inherit from this

Style(node a_root) <- constructor

getTemperature() <- returns String? PhysicalQuantity?

getDate() <- String? DateTime object?

Suggested first stab at API: just enough to navigate files and extract cross sections

class Reaction

Reaction(node a_root) <- constructor

getLabel() <- returns String

getENDF_MT() <- returns int

getCrossSection()

getOutputChannel()

Suggested first stab at API: just enough to navigate files and extract cross sections

class CrossSection <- inherit from base ‘component’ class?

CrossSection(node a_root) <- constructor

getForms() <- returns vector<String> with labels of available forms

getForm(String label) <- return desired form.

Trouble here: could be XYs1d, Regions1d, Reference or ResonancesWithBackground