**Document Creation Process**

The major task in creating a document is accomplished in the GeneratePackage method.

**GeneratePackage**

1. Create new DocPackage. Properties are initialized to new items with no specific content.
   1. DocControl – contains the following (much of this is read in from the Control Panel tool window)
      1. DebugControl – controls verbosity of diagnostic information
      2. PrintControl – various image generation-related factors and controls
      3. DocInfo – file paths and document name
   2. DocPartsIn – Dictionary<string, DocPart>
   3. DocPartsOut – Dictinary<string, DocPart>
   4. ContentItemSet – SortedList<string, ContentItem>
      1. ContentItem contains:
         1. DeName (document element name)
         2. ContentQuery (typically a string with embedded, to-be-resolved, variables
         3. ContentValue (the ContentQuery with all variables resolved)
2. DebugControl
   1. CreateMap
   2. CreateXmlMap
   3. IncludeProperties
   4. InDiagnosticsMode
   5. ShowScale
   6. DiagnosticsLevel
3. PrintControl
   1. CreateDocument
   2. Scale
   3. LineFactor
   4. WidthFactor
   5. SpaceWidthFactor
   6. TextContrast
4. LoadPackage

The “package” is a set of files that are used in the creation of a Word document. They generally adhere to the pattern of the actual files included in a “.docx” file when exploded out. The files are used to “load” a basic pattern for the document to be created (like a template of sorts). Within a folder named for the document name (in this case “Resume1” there will be various files and directories.

The files in the document directory (Resume1 in this case) include the following:

Resume1.res – A “resource” file containing custom (document specific) styles and a “property cache” which streamlines the usage of a predefined set of properties (typically such things as table formatting, etc.) to apply within the document.

Resume.xml – A “high-level template” file that defines the general structure of the document (including such things as the document size) is stored. This file has references to some of the pre-defined properties from the Resume1.res file.

\_rels Folder – This folder contains the “.rels” file which is an xml file that contains the file relationships, including attributes that specify the type of file, its location and an ID.

docProps Folder – This folder contains the “app.xml” file, which appears to be a file of document properties (count of pages, words, characters, etc.). It may be simpy an output file, updated when a document is created. Not sure how it got where it is. It doesn’t seem to be integral to the processing, possibly just a typical output.

map Folder – This folder contains the “map.xml” file, which contains the document structure including textual values (with embedded variables) which form “queries” for the ultimate values, styles and pre-loaded properties (property cache). Need to run the program to understand how this is created and/or used.

word Folder – This folder contains four files as follows:

* docResource.xml – “a” styles and property cache – need to figure out how this is used and/or created.
* document.xml – seems to contain some “a” document structure; has identifying “tags”
* styles.xml – contains “w” styles; apparently a full “default” set, plus named custom styles of “a” origin
* w-document.xml – appears to be the final “output” product of the Word document creation process.

NOTE: Of all the files mentioned in this section, only the w-document.xml file is updated when a document is created. The rest appear to be “inputs”. This needs to be validated.

All of these files are loaded to DocPackage.DocPartsIn (even though w-document.xml appears to be strictly an “output file”.

The Following Process is Used:

* 1. Loads the “.rels” file which specifies the relationships among the various parts of a Word document.
     1. The .rels file contains xml namespaces. The last “slash(/)-delineated” node of the “Type” attribute is the “type of file” that is pointed to in the relationship. For example, these types include the following:
        1. extended-properties
        2. officeDocument
        3. map
  2. Loops through the XElements representing a line of xml in the file. For each “relationship file” indicated by the XElement, the “Target” attribute indicates the relative path within the exploded Word file structure that the specific relationship file exists.
  3. These files are loaded are loaded to an object called a “DocPart” which contains the “Type”, “Target” and “Id” of the file, plus the entire xml of the file placed into an XElement named DocPartXml.
  4. Not sure yet whether I am working with a typical / standard set of relationship files or one that I “honed down” to become a relevant base to work with. Need to figure this out.
  5. These are probably components I have selected in order to be able to generate a Word document – like a minimal set.

1. Delete the “Resume1.docx” file which is the “Word” output of this process.
2. Create a new DocEngine.
3. Run DocEngine.GenerateDocument(), passing in:
   1. The DocPackage (with inputs loaded)
   2. The “width factor”
   3. The “space width factor”
   4. The “line factor”
   5. The “scale”
   6. Seems that all of b-e above could be integral to the package? Or maybe at least aggregated into some type of document generation control object. Need to see where they are used and for what.
4. In the DocEngine.GenerateDocument method.
   1. The “tagId” (DocHelper) is reset to 1000 (the beginning number for “tags”)
   2. The Doc property of the DocumentEngine (the Doc generated by the engine) is created new, passing in the “Package”.
5. In the constructor of Doc (with the package passed in as a parameter)
   1. Doc is a derivative of DocumentElement. So even our document, at the highest level is a DocumentElement.
   2. In Doc.Initialize() multiple properties are initialized. Important properties include:
      1. NameManager
      2. SectionSet (contains all the actual contents of the document)
      3. FontSet (serves as a cache for fonts)
      4. PenSet (serves as a cache for Pens)
      5. PageSet (holds document pages based on dynamic pagination)
      6. DocDefaults
      7. Tags collection (string “tag name”, object)
      8. A number of paths and various xml documents
6. An important concept is whether the package being processed is “Adsdi” or “Word”. These are two “modes” of operation (at a high level) within the generation of a document. First pass through “IsAdsdi” is true. The “Adsdi mode” is used in the original creation of the document from templates, resources and content. This generates a Wordprocessing document. The Wordprocessing document “outputs” its xml with the “w” in the namespace. In the ImageEngine, this xml is then loaded to a new Doc object in the “Word mode”, which uses the composed document to drive the image creation.
7. When IsAdsdi = true, the “docXml” is loaded from package.DocPartsIn[“officeDocument”]. When IsAdsdi = false, the “docXml” is loaded from package.DocPartsOut[“document”].
8. “docXml” contains the “a:” namespace xml structure of a document. This structure includes the wrapping document element plus the body, section properties (which include page margins and page size and orientation), the “shell” table (we want to be able to have flexible document structures and not be dependent on always having a “shell”, and within the “shell”, the main cell (mnCell). Currently within the mnCell, there is a mnTbl, which appears at present to be fully commented out (in the xml).
9. The Doc.DocResource object is loaded from package.DocPartsIn[“docResourse”].DocPartXml. This object contains styles definitions and pre-defined (named) property sets (i.e. for table borders, etc.)
10. Then (in both “IsAdsdi=true and false) the this.LoadFromXml(docXml) is called, loading the document elements from xml recursively.
11. LoadFromXml(docXml) creates a hierarchical collection of DocumentElements that represent everything from the document, to the body, to all its component parts (sections, paragraphs, runs, properties, etc.). These hierarchical elements are later processed to create the Word document objects in a similar hierarchy. They are “definitions” of what needs to be created. They possess many of the same properties as the Word objects (maybe could eventually be extentions or containers, if that was more useful than limiting).
12. The first thing done is that the document (Doc) is initialized by calling the Initialize() method on the base (DocumentElement) class. Even the document itself is a DocumentElement.
13. Significant processing is accomplished in the Initialize method of the base “DocumentElement” class. Some of the processing is generic and some is base on the particular DeType (document element type). There are lots of DeType values including all types of structural elements and things like justification, italics, bold, various types of properties, etc. The idea is that a DocumentElement (base class for lots of types) can contain properties and collections of other DocumentElements in a very generic fashion.
14. All DocumentElements (regardless of the specific type) contain a reference to the top level “document” via the this.Doc property.
15. There are some distinctions of processing between when “this pass” is over the “IsAdsdi” docXml or over the “Word” docXml.
16. All DocumentElements have a “parent” property. This is part of the generic aspect. For the “document” (top) level DocumentElement, the parent property is null.
17. There is also a “depth” property that indicates where the DocumentElement sits in the object hierarchy. The “document” DocumentElement is at depth = 0.
18. Each DocumentElement has a collection of children (also DocumentElements).
19. Each DocumentElement has a “ContentQuery” string which contains a mix of text and variables (to be resolved) that eventually becomes the content in the document.
20. There are “tags”, “classes” and “names” that can come from both the “a” side and the “w” side.
21. Each element has an absolute path property which is a representation of exactly where in the hierarchical document the particular element exists. For example the absolute path to the document is “null/document0”, but this is eventually replaced in the type-specific processing in the “Doc” class with “res1” with other lower level paths being defined as “res1/body0/sect0”, etc.
22. Each DocumentElement is given a generated name based on its type and a number. The “document” DocumentElement is typically “document0”. I believe that names can be explicitly assigned. Tags are also automatically generated and are kept in a name-based collection with a reference to the DocumentElement object for very quick tag based referencing. Tags are more central to document manipulation since we may only have a small number of tags that point to elements that are manipulated whereas every element is named.
23. The RelPath property is the DocumentElement’s path relative to the “tgt” or target of the placement of the content. This implies that the content will not be placed at a higher level. It also implies a possibly consistent high-level structure. We may want to use a simpler document without any higher level structure – placing paragraphs or other elements directly into the sections (i.e without any “table structure”).
24. Each DocumentElement also contains a “Properties” collection. Each “property” is also a DocumentElement (I think).
25. Each DocumentElement (as mentioned above) has significant default processing occur in the base (DocumentElement) class. But each DocumentElement is also a derived class object (“document”, “body”, “paragraph”, etc., with type-specific processing as well. If the “xml” for the element is null, no lower level processing occurs. But when the xml is not null, then the this.LoadChildren(xml) method is called.
26. LoadChildren is a method of the base class (DocumentElement). The “children” processed are the members of the set created by getting the XElements from xml.Elements(). This the “top” level, parent object is not processed but all subordinate elements are.
27. The LoadChildren method in the base class has significant processing that is type-specific because each child element is used to drive the new object of the specific type derived from DocumentElement. This is primarly true in the way of document structure. The various properties (of sections, paragraphs, runs, etc.) are processed within the specific types and not in base.LoadChildren.
28. After the document is loaded, the “PrintControl.CreateDocument” flag determines whether a “WordprocessingDocument” object gets created.
29. A “using” statement controls the life cycle of the WordprocessingDocument. Upon its creation, the following processing occurs:’
30. There are a number of high-level parts of the Word document that get created. Many are simply placed in the code and still need to be looked at to determine whether they are necessary or desired. Such high-level parts include “ExtendedFilePropertiesPart”, “StylesWithEffectsPart”, “EndnotesPart”, “NumberingDefinitionsPart”, “FootNotesPart”, “WebSettingsPart”, “ThemePart”, “DocumentSettingsPart”, “FontTablePart”.
31. Each of the parts above are created new as objects added to the document then a reference (to the newly created object) is passed to DocHelper for population.
32. The second “part” added is the “MainDocumentPart”, where the bulk of the processing relevant to the resume (or other document) occurs.
33. In the “AddMainDocumentPart” method.
34. A “StylesDefinitionPart” is added.
35. The “body” is created and added by executing: “document.MainDocumentPart.Document = new Document(body)”; thus by creating a body and then creating the new Document by passing in the body object, the “document.MainDocument.Document comes into being.
36. Multiple namespace declarations are added to the document.
37. The body is created using the “BuildBody” method of the DocEngine (where we are primarly processing). The BuildBody method constructs the entire document by performing recursive hierarchical processing.
38. Each DocumentElement in the hierarchy drives the creation of a corresponding Word object in the method named “BuildElement”.
39. The strategy is to build a Word document as a structure and then add the content (in the AddContent method). The content is based on “Resume” type objects (could be other types) that embody the data for the resume. It is this data that would be retrievd from a database based on user entry of a new resume.
40. The content is mapped into “tags” which are associated with Word document objects which can then be directly updated with content (slick).
41. PICK UP THIS DOCUMENTATION PROCESS IN THE IMAGEENGINE PROCESSING OF THE “W” xml.