First Article Inspection Requirement Report Generation from QIF Using C++, CodeSynthesis, and Mozilla Xerces

John Michaloski, john.michaloski@nist.gov
Monday, May 30, 2016

Abstract

This document details an automated approach to generating AS9102 reports from inspection material represented in Quality Information Framework (QIF) Extensible Markup Language (XML). AS9102 is a Society of Manufacturing Engineers (SME) Aerospace Quality Standard, also known as First Article Inspection Requirement (FAIR) report. The described automated approach should save companies significant amounts of time and resources in filling out FAIR reports. Much detail is given in the document describing the C++ software development used to parse, navigate, and extract relevant FAIR information from the QIF XML. In particular, the software applications and libraries including XML Schema Definition (XSD) from CodeSynthesis and Xerces from Mozilla Developer Network are given extensive and detailed explanation in the application of these tools in the FAIR generation. The accompanying QIF C++ code is available on the NIST GitHub repository at https://github.com/usnistgov/QIF.

Notation

ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
DME	Dimensional Measuring Equipment
DMSC	Dimensional Metrology Standards Consortium
DOM	Document Object Model
FAIR	First Article Inspection Requirement
GD&T	Geometric dimensioning and tolerancing
HTML	HyperText Markup Language
IMTS	International Manufacturing Technology Show
NIST	National Institute of Standards and Technology
PDF	Portable Document Format
PMI	Product and Manufacturing Information
QIF	Quality Information Framework
SME	Society of Manufacturing Engineers
STD	Standard Library
STL	Standard Template Library
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

W3C	World Wide Web Consortium
XML	Extensible Markup Language
XPATH	XML Path Language
XSD	XML Schema

Background

Quality of a product may be defined as ``its ability to fulfill the customer's needs and expectations''. (United Nations Industrial Development Organization, 2006) For the manufacturer, Product and Manufacturing Information (PMI) conveys information (including quality) such as geometric dimensioning and tolerancing (GD&T), 3D annotation (text), surface finish, and material specifications. Quality is defined in terms of performance requirements, which vary from product to product. For discrete parts, the primary performance requirements, commonly referred to as characteristics, are dimension (e.g., length, diameter, thickness, or area), geometric tolerances (e.g., flatness, cylindricity, etc.), and appearance (e.g., surface finish, color, or texture). To ensure overall quality, delivered parts must meet the required quality characteristics. Thus, part quality is measured by its conformance to the performance requirements. Of interest in our work is the use of quality standards, and especially feedback from quality standards, to improve part design and fabrication, and as a consequence the quality of products.

Quality measurement performs the part inspection, in which the actual measurement points are saved, with measured points corresponding to desired nominal points. Overall, quality results represent parts as collections of production knowledge, which includes design, manufacturing, and inspection data. For example, a hole can be expressed with geometric design data for the hole location, diameter, and depth. Maybe a drill will machine the hole. A hole can also be associated with GD&T data to ascribe the tolerance of the hole location, diameter, and depth as well as relationships to other features. If the tolerance is very tight, maybe a reaming after a drilling is desirable. Saving the quality information in a standard Extensible Markup Language (XML) format, is the role of the Quality Information Framework (QIF).

First Article Inspection Report (FAIR) is a formal method of providing a measurement report for a given manufacturing process. (SME International, 2014) The method consists of measuring the properties and geometry of an initial sample of items against given specifications, for example a drawing. Items to be checked in a FAIR are wide and varied and may include distances between edges, positions of holes, diameters and shapes of holes, weight, density, stiffness, color, reflectance, or surface finish. Despite the name, the inspected article may not necessarily be the 'first' produced. First article inspection is typically called for in a contract between the producer and buyer of some manufactured article, to ensure that the production process reliably produces what is intended.

"*Shortly after the start of widespread AS 9102 adoption, suppliers offered anecdotal evidence of the volume of first articles and the level of human resources devoted to their completion. It was not unusual for them to cite numbers in excess of 40 hours to complete an AS 9102 compliant first article inspection. Moreover, the volume of first article inspections at some suppliers was reported to be 100 or more* *annually. This combination of effort and volume represented a substantial drain on their limited technical resources.*" (Quality Magazine, 2007)

This document describes an automated approach to generating FAIR reports. In theory, the described approach will save companies substantial amounts of time and resources in filling out these necessary, but time-consuming, inspection reports. The approach is based on using measurement results as described in XML according to the Quality Information Framework (QIF) standard. Given inspection part feature/characteristic definitions and corresponding measurement results defined in the QIF standard XML format, it is straightforward to automate the FAIR report generation. Underlying C++ code that parses the QIF XML and translates the QIF XML into the FAIR reports will be described. This QIF C++ code is available on the usnistgov github web site found at https://github.com/usnistgov/QIF.

QIF is an American National Standards Institute (ANSI) standard sponsored by the Dimensional Metrology Standards Consortium (DMSC) that defines an integrated set of XML information models to enable the effective exchange of metrology data throughout the entire manufacturing quality measurement process – from product design to inspection planning to execution to analysis and reporting. An instance of QIF is a QIF document, which contains quality information in standard XML format – including first article inspection data. This document describes how the QIF can be transformed from a XML document into FAIR reports Part 1 and 3 (SME International, 2014).

FAIR reports handle both assemblies (including numerous parts) and individual part inspections. For this document, only single parts are discussed. Further, it will be assumed that a complete quality and measurement results section is included in the QIF document that satisfies the FAIR requirements.

Measurement Results

This section will review an implementation for generation of First Article Inspection Requirement reports from the QIF XML. The sample QIF XML file used in this document is "QIF_Results_Sample.xml", which can be found in the QIF distribution under the "sampleInstanceFiles" directory. (Dimensional Metrology Standards Consortium , 2015)

AS9102a SME Aerospace Quality Standard

AS9102 is a Society of Manufacturing Engineers (SME) Aerospace Quality Standard, known as, First Article Inspection Requirement (FAIR). QIF contains measurement results that can be used to generate an AS9102 (SME International, 2014). Only AS9102 Forms 1 and 3 (SME International, 2014) are considered relevant to QIF. Form 2 discusses material and manufacturing processes and is outside the scope of QIF. Information for filling out a FAIR may be extracted from a QIF 2.0 XML QIFDocument instance file. The instance file must conform to the QIFDocument.xsd information model.

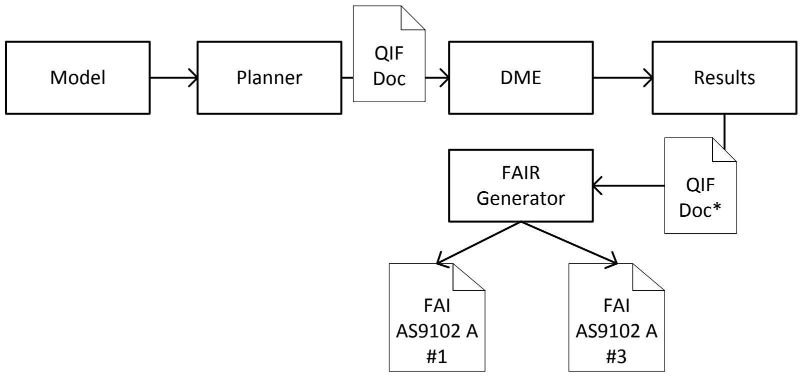


Figure 1 Possible FAIR workflow using a QIFDocument instance

A possible FAIR workflow is shown in Figure 1. In the Model phase, a part is designed with its Product and Manufacturing Information (PMI) included, specifically geometric dimensioning and tolerancing (GD&T) information. Within the PMI, characteristics (e.g., cylindricity tolerance) are associated to one or more part features (i.e., holes). Once the feature and characteristic are developed, the Planner assigns nominal measurement points for the inspection device to measure based on some quality tolerance rules and produces a QIF document (QIF Doc in the diagram). Dimensional Measurement Equipment (DME) performs the physical inspection and produces QIF measurement results to be incorporated into the QIF document. Of note, the same QIF document can be used for the before/after results measurements (as denoted by an asterisk). However, after the DME inspection the actual measurements (Results) are included in the QIF document.

Report Generation

The FAIR report generation depends on the parsing of the QIF XML using CodeSynthesis. The CodeSynthesis "XSD" software tool (CodeSynthesis, 2015) generates C++ code for XML Parsing and Tree Mapping. The primary purpose of this generated C++ code is to serialize and deserialize QIF XML. For our purposes, serialization is a process by which a program's C++ internal representation is transformed into an XML serial data format. Likewise, deserialization (i.e., XML parsing) is used to convert the XML into a program's C++ internal representation.

CodeSynthesis uses the Xerces C++ XML Schema (XSD) and XML tools (Mozilla, 2015) to generate code that will parse the QIF XML, as well as to verify compliance to the QIF XSD specification. CodeSynthesis is an open-source, freely distributable, code generation licensing, cross-platform World Wide Web Consortium (W3C) XML Schema to C++ data binding compiler. When provided with the QIF XML Schema (XSD), the CodeSynthesis XSD tool generates C++ classes that represent the given QIF inspection vocabulary and provide XML functionality such as parsing and serialization code. Once parsed, you can access the XML data using C++ types and functions that semantically correspond to the inspection domain rather than dealing with generic XML mechanisms. Typically, but not always, the C++ representation matched to the application domain (in this case inspection) provides an easier programming method than dealing directly with the XML.

The following code snippets assume that the Code Synthesis XSD tool has already generated the corresponding C++ classes to the QIF XSD. These coding examples were done under the Windows Operating System using Microsoft Visual C++ 2010, but have been validated with Ubuntu/Linux OS and the GNU C++ compiler.

First, the QIF XML document is parsed into Xerces Document Object Model (DOM) node representation (World Wide Web Consortium, 2004) and CodeSynthesis C++ tree representation using the QIFDocument constructor generated by CodeSynthesis. When generating C++ code, the CodeSynthesis XSD tool must have the flag for Xerces DOM nodes enabled. Then, a filename pointing to a file containing the QIF XML is passed, and if successful is parsed into a navigable DOM tree. Error handling is done with a try/catch exception handler, but is not detailed here.

```
CFairReport fair;

// Parse QIF XML
std::string filename=::ExeDirectory() + "\\QIF_Results_Sample.xml";
std::auto_ptr<QIFDocumentType> qif (
		QIFDocument (filename,
		xml_schema::flags::dont_initialize|xml_schema::flags::dont_validate|xml_schema::flags::keep_dom)
);

DOMElement* e = static_cast<DOMElement*> ((*qif)._node ());

//
// Generate FAIR Form 1
std::string report1 = fair.GenerateFAIRReport1(e);
SaveReport(::ExeDirectory()+ "FairReport1.html", report1);

//
// Generate FAIR Form 3
std::string report3 = fair.GenerateFAIRReport3(e);
SaveReport(::ExeDirectory()+ "FairReport3.html", report3);
```

Assuming the QIFDocument was correctly parsed into a DOM tree, the root node of the tree is saved as a Xerces DOM Element e. There is no verification or validation of the parsed QIF to insure that the FAIR report items are available, only blanks will be produced in the forms if the information is missing. Once parsed FAIR reports 1 and 3 can be generated given the QIF root element. The method to save the report string is a utility function SaveReport, which saves the string as a file when given a file name. The utility function ExeDirectory() returns the executable path location, whose folder is used to store and retrieve the files.

FAIR Report 1 Generation

FAIR generation will be discussed for the AS9102a version even though there is an updated and newer AS9102b version. Differences between the versions are minor. FAIR report 1 can be solely represented using XPATH for navigation of the QIF XML tree to retrieve values. It is beyond the scope of this document to explain the workings of XPATH, and the reader is referred to (World Wide Web Consortium, 1999) for further explanation and examples. Table 1 gives the FAIR Form 1 fields, their column number on the form, and the XPATH mapping in QIF XML that matches the field. The generation of tables describing FAIR fields and corresponding QIF XML depended greatly on the explanation found in (Kramer, 2014). The application of QIF to generate FAIR was demonstrated at the 2014 International Manufacturing Technology Show (IMTS).

Table 1 FAIR Form 1 Field Names, Numbers and Matching QIF XPATH

Form 1 Fields	#	QIF XPATH descriptor
Part Number	1	QIFDocument/MeasurementResults/ActualComponentSet/ActualComponent Or QIFDocument/Product/PartSet/Part[]/
Part Name	2	QIFDocument /Product/PartSet/Part[]/Name
Serial Number	3	QIFDocument /MeasurementsResults/ActualComponentSet[1..n]/ActualComponent[1..n]/SerialNumber
FAI Report Number	4	QIFDocument /PreInspectionTraceability/ReportNumber
Part revision level	5	QIFDocument /Product/PartSet/Part[]/Version
Drawing Number	6	QIFDocument /Product/PartSet/Part[]/DefinitionExternal/ PrintedDrawing[]/PrintedDrawing/ DrawingNumber
Drawing Revision Level	7	QIFDocument /Product/PartSet/Part[]DefinitionExternal/PrintedDrawing[]/Version
Additional Changes	8	QIFDocument/Product/PartSet/Part[]/DefinitionExternal/PrintedDrawing[]/AdditionalChanges
Manufacturing Process Reference	9	N/A
Organization Name	10	QIFDocument /PreInspectionTraceability/InspectingOrganization/Name
Supplier Code	11	QIFDocument /PreInspectionTraceability/SupplierCode
P.O. Number	12	QIFDocument /PreInspectionTraceability/PurchaseOrderNumber
Detail FAI	13	QIFDocument /PreInspectionTraceability/InspectionScope
Full/Partial FAI	14	QIFDocument /PreInspectionTraceability/InspectionMode
If assembly	15 - 18	List of part number, part name, part serial number, FAI report number

Items in fields 15-18 are to be filled in only if the report describes an assembly. One line of the AS9102a form 1 should be filled in for each component of the assembly. Each line has entries for items 15-18. Since item 17 is a serial number, the instructions here assume the existence of a corresponding “MeasurementsResults” section in the “QIFDocument”. Further, the “PrintedDrawing” field is applicable only if there is printed drawing. That is not always the case, since a digital drawing or some form of digital model may be used instead of a printed drawing. In addition, there may be more than one “PrintedDrawing” for a part or assembly.

Using this FAIR form template, we can use the associated XPATH code described to find all the information we need to fill in the form. The FAIR reports produces documentation based on the previously noted QIF example file, “QIF_Results_Sample.xml.

The FAIR generation code was encapsulated in the C++ class CFairReport. Shown earlier was the use of the CFairReport class to generate reports 1 and 3. CFairReport is declared and implemented in the FairUtils.{h,cpp} files. Inside the CFairReport declaration, variables to describe Form1 and Form3 are declared. Since multiple inspection instances can be described within a single form, C++ standard vector of standard ASCII strings is used to store all the XML data required from the QIF document. Below the string vectors are declared that are required of Form 1:

```
std::vector<std::string> part_names;
std::vector<std::string> part_numbers;
std::vector<std::string> serial_numbers;
std::vector<std::string> FAI_report_numbers;
std::vector<std::string> part_revisions;
std::vector<std::string> drawing_numbers;
std::vector<std::string> drawing_revisions;
std::vector<std::string> additional_changes;
std::vector<std::string> organization_names;
std::vector<std::string> supplier_codes;
std::vector<std::string> po_numbers;
std::vector<std::string> detail_fai;
std::vector<std::string> full_fai;
```

Below, the GenerateFAIRReport1 method is further developed, which requires the parsed DOM root node (as a DOM element) be passed as an argument. Among several responsibilities, the C++ class CXercesUtils serves to assist in the XPATH search and retrieval of information using Xerces. Xerces only supports XPATH 1.0, but this level of sophistication suffices for retrieving the information required of FAIR report 1. The class CXercesUtils has a method, GetXpathResults, which takes a starting DOM element, and an XPATH upon which to search the DOM tree beginning with the DOM element. It is beyond the scope of this document to fully explain all the intricacies of XPATH, but suffice to say that the XPATH “//QIFDocument/Product/PartSet/Part/ModelNumber” starts searching for all the matching “QIFDocument” elements in the tree, and for each matching element returns a node list of XML node whose branches match “Product/PartSet/Part/ModelNumber”. By definition, there is only one “QIFDocument” in a QIF XML instance file.

```
std::string CFairReports::GenerateFAIRReport1(xercesc::DOMElement* e )
{
 try
 {
  CXercesUtils utils;

  // Form 1
  //
  part_numbers = utils.GetXpathResults(e, "//QIFDocument/Product/PartSet/Part/ModelNumber");
  part_names =utils. GetXpathResults(e, "//QIFDocument/Product/PartSet/Part/Name");
  serial_numbers = utils.GetXpathResults( e,
            "//QIFDocument/MeasurementsResults/ActualComponentSet/ActualComponent/SerialNumber");
  FAI_report_numbers = utils.GetXpathResults(e, "//QIFDocument/PreInspectionTraceability/ReportNumber");

  part_revisions= utils.GetXpathResults(e, "//QIFDocument/Product/PartSet/Part/Version");
  drawing_numbers= utils.GetXpathResults( e,
            "//QIFDocument/Product/PartSet/Part/DefinitionExternal/PrintedDrawing/DrawingNumber");
  drawing_revisions= utils.GetXpathResults( e,
            "//QIFDocument/Product/PartSet/Part/DefinitionExternal/PrintedDrawing/Version");
  additional_changes= utils.GetXpathResults(e,
            "//QIFDocument/Product/PartSet/Part/DefinitionExternal/PrintedDrawing/AdditionalChanges");

  organization_names= utils.GetXpathResults(e, "//QIFDocument/PreInspectionTraceability/InspectingOrganization/Name");
  supplier_codes= utils.GetXpathResults(e, "//QIFDocument/PreInspectionTraceability/SupplierCode");
  po_numbers= utils.GetXpathResults(e, "//QIFDocument/PreInspectionTraceability/PurchaseOrderNumber");

  detail_fai= utils.GetXpathResults(e, "//QIFDocument/PreInspectionTraceability/InspectionScope");
  full_fai= utils.GetXpathResults(e, "//QIFDocument/PreInspectionTraceability/InspectionMode");

  return Form1();
 }
 catch (const xml_schema::exception& e)
 {
  cerr << e << endl;
 }
 catch (...)
 {
  XERCES_STD_QUALIFIER cerr << "An error occurred parsing/creating the FAIR document" << XERCES_STD_QUALIFIER endl;
 }
 return "";
}
```

Again, exception blocks are used to trap any CodeSynthesis or Xerces errors . It is assumed that the Xerces DOM parser would catch any egregious QIF XML parse errors.

The line “return Form1();” develops a standard HyperText Markup Language (HTML) string describing the FAIR report 1 form using the contents extracted from the QIF XML. Below, the Macro “TEXT” returns either a Unicode or ASCII text string depending on the application implementation. Likewise, the method “CheckEntries()” pushes blank strings into any empty string vectors to prevent memory exceptions from vectors with no elements.

```
std::string CFairReports::Form1()
{
        CheckEntries(); // Clean up any empty arrays

        std::string form;
        form += TEXT("<html>\n");
        form += TEXT("<header>\n");
        form += TEXT("<style type=\"text/css\">\n");
        form += TEXT("<!--\n");
        form += TEXT("@page { size:8.5in 11in; margin: .25in }\n");
        form += TEXT("-->\n");
        form += TEXT("table { table-layout: fixed; }\n");
```

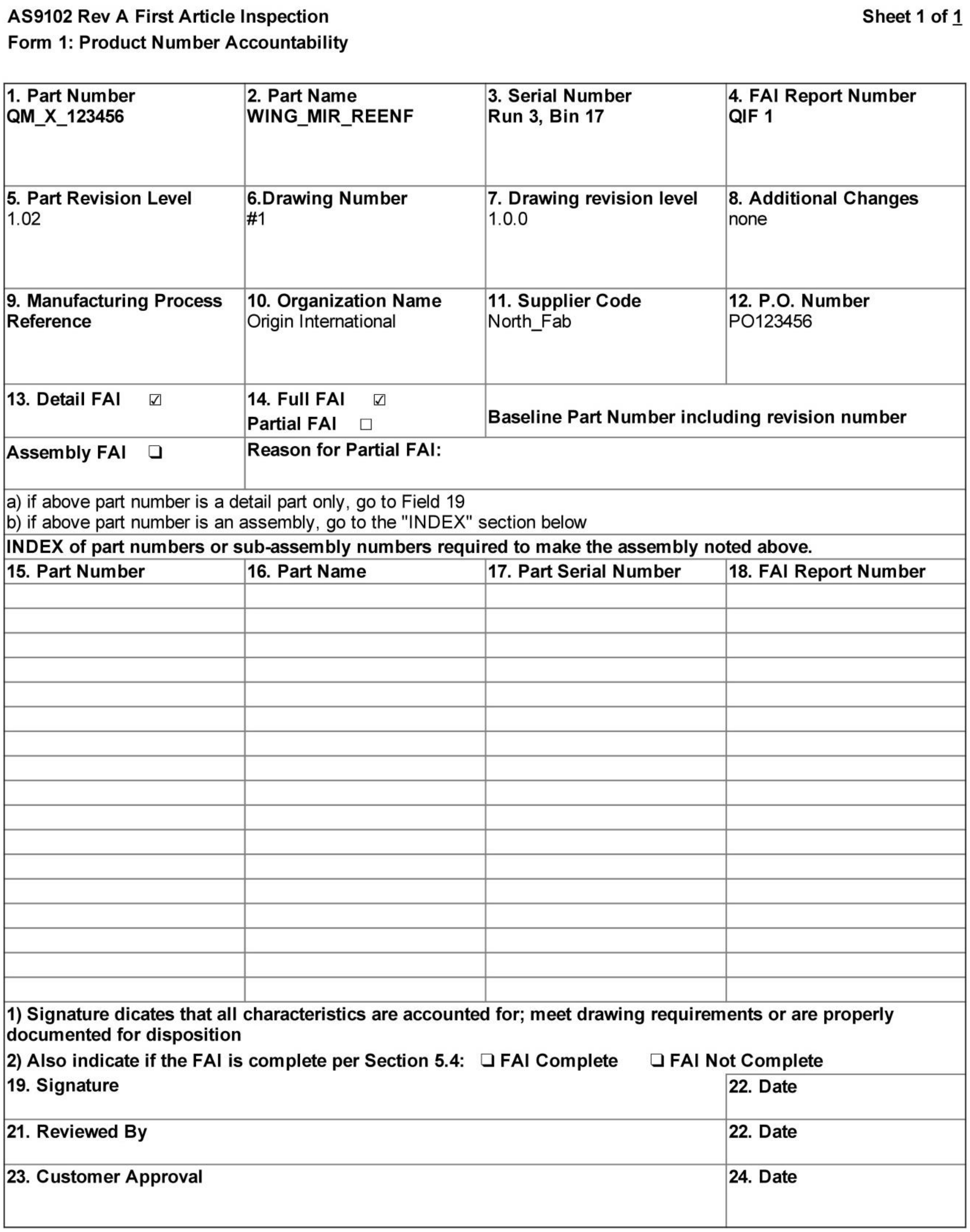
```
            form += TEXT("\n");
            form += TEXT("@media print {\n");
…
            form += TEXT("<tr height=75px valign=\"top\" align=\"left\">\n");
            form += StdStringFormat("<td ><b> 5. Part Revision Level</b><br>%s</td>\n", part_revisions[0].c_str());
            form += StdStringFormat("<td ><b> 6.Drawing Number </b><br>%s</td>\n", drawing_numbers[0].c_str());
            form += StdStringFormat("<td ><b> 7. Drawing revision level</b><br>%s</td>\n", drawing_revisions[0].c_str());
            form += StdStringFormat("<td ><b> 8. Additional Changes </b><br>%s</td>\n", additional_changes[0].c_str());
            form += TEXT("</tr>\n");
…

            form += TEXT("</table>\n");
            form += TEXT("</body>\n");
            form += TEXT("</html>\n");
            return form;
}
```

Now we write the FAIR report form 1 using the “SaveReport” method, shown earlier, which encapsulates the following standard C++ ostream code:

```
std::string report1 = fair.Form1(); // assumes Report 1 XML has already been parsed
std::ofstream out(::ExeDirectory() + "FairReport1.html");
out << report1 << std::endl;
out.close();
```

The file “FairReport1.html” now contains the FAIR inspection report 1.

Of note, it is best if you use Google Chrome web browser to view and then print the form. Google Chrome also has the ability to print to Portable Document Format (PDF), which would allow you to digitally sign the document. Important to note, the html has a separate page style embedded in the html document that will adhere to a portrait 8.5”x11” page and will keep the fields layout as prescribed by the AS9102a standard. For best results, you will want to turn off the Chrome checkbox “Headers and Footers” when printing. Figure 2 shows the FAIR report 1 html output opened in Google Chrome, saved as PDF, and then displayed in Adobe Acrobat from which the screen shot in Figure 2 is taken.



AS9102 Rev A First Article Inspection **Sheet 1 of 1**
Form 1: Product Number Accountability

1. Part Number **QM_X_123456**	**2. Part Name** **WING_MIR_REENF**	**3. Serial Number** **Run 3, Bin 17**	**4. FAI Report Number** **QIF 1**
5. Part Revision Level 1.02	**6.Drawing Number** #1	**7. Drawing revision level** 1.0.0	**8. Additional Changes** none
9. Manufacturing Process Reference	**10. Organization Name** Origin International	**11. Supplier Code** North_Fab	**12. P.O. Number** PO123456
13. Detail FAI ☑	**14. Full FAI** ☑ **Partial FAI** ☐	**Baseline Part Number including revision number**	
Assembly FAI ❑	**Reason for Partial FAI:**		
a) if above part number is a detail part only, go to Field 19 b) if above part number is an assembly, go to the "INDEX" section below			
INDEX of part numbers or sub-assembly numbers required to make the assembly noted above.			
15. Part Number	**16. Part Name**	**17. Part Serial Number**	**18. FAI Report Number**
1) Signature dicates that all characteristics are accounted for; meet drawing requirements or are properly documented for disposition **2) Also indicate if the FAI is complete per Section 5.4:** ❑ **FAI Complete** ❑ **FAI Not Complete**			
19. Signature			**22. Date**
21. Reviewed By			**22. Date**
23. Customer Approval			**24. Date**

Figure 2 FAIR Report Form 1 filled from Example QIF XML

Below the code to search and extract the XML information from the XPATH query is presented. The code uses Xerces XPATH 1.0, so a default namespace resolver is used, then an XPATH expression is created, executed, and the results are individually accessed from Xerces and stored into a string vector.

Namespace resolvers determine external XML resources named by a Uniform Resource Identifier (URI) in the XML. In the code below, the QIF namespaces are used as determined by the DOM root node.

```
std::vector<std::string> CXercesUtils::GetXpathResults(DOMElement* root, std::string querystr)
{
 std::vector<std::string> values;

 DOMDocument* doc (root->getOwnerDocument ());

 // Obtain namespace resolver.
 xsd::cxx::xml::dom::auto_ptr<DOMXPathNSResolver> resolver (
  doc->createNSResolver (root));

 // Create XPath expression.
 xsd::cxx::xml::dom::auto_ptr<DOMXPathExpression> expr (
  doc->createExpression (
  xsd::cxx::xml::string (querystr.c_str()).c_str (),
  resolver.get ()));

 // Execute the query.
 xsd::cxx::xml::dom::auto_ptr<DOMXPathResult> r (
  expr->evaluate (
  root, DOMXPathResult::ORDERED_NODE_SNAPSHOT_TYPE, 0));

 // If no query matches, then return empty vector
 if (!r.get() )
  return values;

 // Iterate over the result and save into string vector
 for (int i=0; i < r->getSnapshotLength(); i++)
 {
  r->snapshotItem(i);
  DOMNode* n (r->getNodeValue ());
  const XMLCh * value = n->getTextContent  (    );
  values.push_back(xsd::cxx::xml::transcode<char> (value));
 }
 return values;
}
```

FAIR Report 3 Generation

FAIR report 3 cannot be exclusively represented using XPATH for navigation of the QIF XML tree to retrieve values. Instead, some of the fields use XML pointers (id attributes) to navigate through the DOM tree. Thus, for FAIR report 3 it is common to find an element that defines an id that corresponds to a branch of the DOM tree found elsewhere. This matching of ids although formal is still rather ad hoc, in that although the id may be verified to be unique, a casual XML reader must know through reading the QIF specification what element the id matches.

For FAIR report form 3, the example QIF XML file noted earlier omitted some Form 3 values, which negated the utility of FAIR report, in that if you have 11 critical features and 8 criticality elements, then you need to refine the process of resolving field information. Instead the following items were added to some of the “CharacteristicItem” XML branches:

```
<Criticality>CRITICAL</Criticality>
```

in the example QIF XML file.

FAIR Form 3 allows numerous part features characteristics to be described, hence an array of these instances is culled from the QIF XML tree. So the strategy to extract the relevant QIF information is to navigate to the branch(es) of interest and then with each branch extract the relevant FAIR report 3 information. Table 2 gives the FAIR Form 3 fields, their column number on the form, and the QIF Document Object Model (DOM) tree navigation mapping to retrieve the QIF field. As mentioned, often the branches have "id" pointers to other DOM branches that must be resolved.

Table 2 FAIR Form 3 Fields, Numbers and QIF XML identification

Form 3 Fields	#	QIF Tree Navigation Description
Characteristic No	5	QIFDocument/Characteristics/CharacteristicItems/CharacteristicItem[]/Name
Reference Location	6	Id from QIFDocument/Characteristics/CharacteristicItems/CharacteristicItem[]/LocationOnDrawing/DrawingId Matching QIFDocument /Product/ PartSet/Part[]/DefinitionExternal/ PrintedDrawing[]/PrintedDrawing/DrawingNumber attribute Id AND QIFDocument/Characteristics/CharacteristicItems/CharacteristicItem[]/LocationOnDrawing/SheetNumber QIFDocument/Characteristics/CharacteristicItems/CharacteristicItem[]/LocationOnDrawing/DrawingZone
Characteristic Designator	7	QIFDocument/Characteristics/CharacteristicItems/CharacteristicItem[]/KeyCharacteristic/Criticality
Requirement	8	QIFDocument/Characteristics/CharacteristicNominals/…
Results	9	QIFDocument/MeasurementResults/MeasuredCharacteristics/CharacteristicActuals[]/Status/CharacteristicStatusEnum (PASS/FAIL)
Designed Tooling	10	Id from QIFDocument/Characteristics/CharacteristicItems/CharacteristicItem[]/MeasurementDeviceIds/Id Matching id in QIFDocument/MeasurementResources/MeasurementDevices
Non-conformance Number	11	QIFDocument/MeasurementResults/MeasuredCharacteristics/CharacteristicActuals/NonConformanceDesignator
Prepared By	12	QIFDocument /MeasurementsResults /InspectionTraceability/ReportPreparer/Name
Date	13	QIFDocument /MeasurementsResults /InspectionTraceability/ReportPreparationDate
Optional Fields	14	

Within the document the "id" pointer mechanism can be reused. The "Requirement" Form 3 field is a recursive field that uses the attribute "id" as a pointer to underlying requirements/definitions in the CharacteristicDefinitions as well as the Datum.

To automate the information requirements of FAIR Form 3, it was necessary to combine the strengths of Xerces DOM object model with the CodeSynthesis C++ object Tree model. First, we needed to access all the elements under a series element. For this, a template method was defined that took a CodeSynthesis type as the template parameter, and then returned a Standard Template Library (STL) (Plauger, 2001) vector of the matching XML nodes matching the XPATH query string.

```
template<typename T>
        std::vector<T *> GetXpathAsStructs(DOMElement* root, std::string querystr)
```

All the queries were simple so they could be done with Xerces XPath 1.0. Below shows the fetch of all the CharacteristicItems that were then cast into CharacteristicItemBaseType for easier manipulation. Again the root DOM Element is passed into the routine.

```
DOMElement* e = static_cast<DOMElement*> ((*qif)._node ());

std::vector<xsd::qif2::CharacteristicItemBaseType*>   cis=     utils.GetXpathAsStructs<xsd::qif2::CharacteristicItemBaseType>(e,
"//QIFDocument/Characteristics/CharacteristicItems/*");

for(size_t k=0; k< cis.size(); k++)
{
. . .
}
```

Next, FAIR form 3 structures were created to hold the information matching the requirements of FAIR Form 3. The requirements match the above XPATH presented in Table 1. Again, the "id" concept as a pointer construct to another part of the QIFDocument is used.

```
        struct CharacteristicInfo
        {
                std::string Name;
                int id;
                std::string characteristic_designator;
                std::string requirement;
                std::string results;
                std::string print_drawing_name;
                std::string sheet_number;
                std::string drawing_zone;
                int designed_toolingId;
                std::vector<int> measurement_ids;
                std::vector<std::string> measurement_names;
                std::string optional;
        };
        std::vector<CharacteristicInfo> char_info;
```

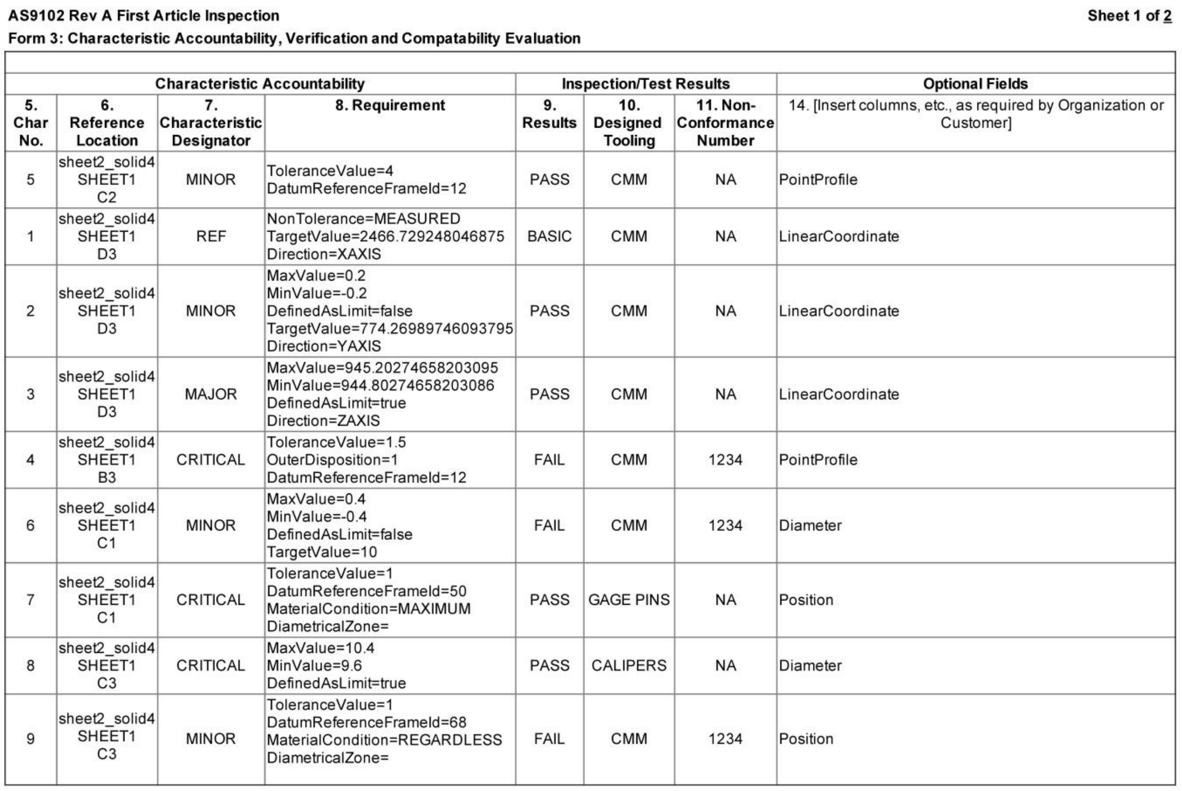
We will show how one of the CharacteristicInfoBaseType was traversed and information extracted from the XML file, in a combination of Xerces and CodeSynthesis. Once the STL vector of CharacteristicInfoBaseType pointers has been collected, each item in the vector is traversed one by one to save the characteristic information and using SAFESTORE to prevent unhandled exceptions if the element does not exist, and so it would be a NULL pointer access. The background on SAFESTORE is discussed later in the document.

```
std::string CFairReports::GenerateFAIRReport3(xercesc::DOMElement* e )
{
 CXercesUtils utils;

 // Get all the XML CharacteristicItems nodes to help fill out form
 //
 std::vector<xsd::qif2::CharacteristicItemBaseType*>   cis=      utils.GetXpathAsStructs<xsd::qif2::CharacteristicItemBaseType>(e,
              "//QIFDocument/Characteristics/CharacteristicItems/*");

 for(size_t k=0; k< cis.size(); k++)
 {
              CFairReports::CharacteristicInfo ci;
              std::string tooling_id;
              std::string ci_id;
              SAFESTORE(ci.Name, cis[k]->Name(),"");
              SAFESTORE(ci.characteristic_designator,*(cis[k]->KeyCharacteristic()->Criticality()),"");
              SAFESTORE(ci.sheet_number,*(cis[k]->LocationOnDrawing()->SheetNumber()),"");
              SAFESTORE(ci.drawing_zone,*(cis[k]->LocationOnDrawing()->DrawingZone()),"");
              SAFESTORE(ci_id,cis[k]->id(),"");
              ci.id = GetIdFromString(ci_id);
              ci.optional= utils.GetElementName(static_cast<DOMElement*> ((*cis[k])._node ()));
              ci.optional=ci.optional.substr(0,ci.optional.find("CharacteristicItem"));
              SAFESTORE(tooling_id, *(cis[k]->LocationOnDrawing()->DrawingId()), "-1");
              ci.designed_toolingId= GetIdFromString(tooling_id);

              int num;
              SAFESTORE(num,cis[k]->MeasurementDeviceIds()->Id().size(),0); // in case no device specified, catch exception
              for(size_t m=0; m < num; m++)
              {
                           std::string id;
                           SAFESTORE(id,cis[k]->MeasurementDeviceIds()->Id().at(m),"-1");
                           ci.measurement_ids.push_back(GetIdFromString(id));
              }

              fair.char_info.push_back(ci);
 }
. . .
```

Ids were complicated because they are stored as char, but can only be saved as string (so far). So a routine was written to transcribe the unsigned char sequence into an “int” that was then used as a STL map header when necessary.

The use of the variable numbers to retrieve the number of Measurement Device ids was used as the access to the id may be null, again we can catch the exception, and store the default “0” into the number variable. The utility GetIdFromString() method was used to extract an integer number from a character array.

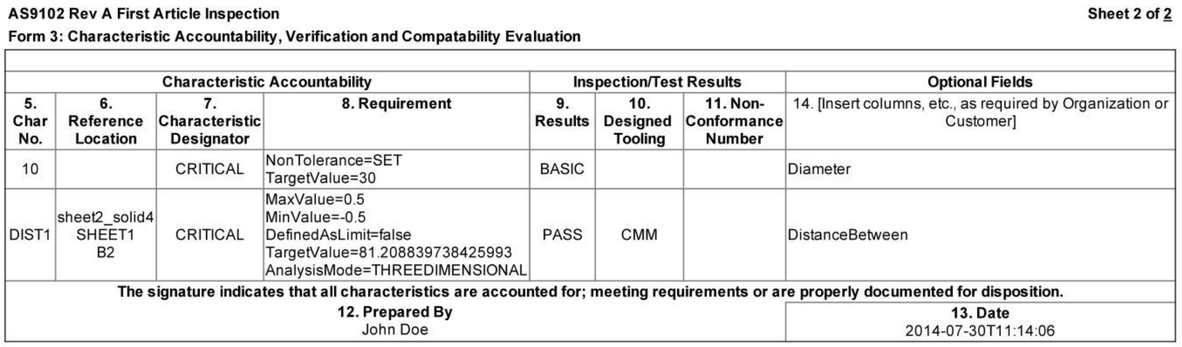
We can write the FAIR report 3 Form using the “SaveReport” method, as described earlier. The file “FairReport3.html” now contains the FAIR inspection report 3. Again, it is best if you use Google Chrome to view and then print the form. Instructions pertaining to displaying and printing the HTML and saving the document as a PDF are described earlier. This is especially important since Form 3 is often a multipage document, and is rendered properly by Google Chrome and Mozilla Firefox browsers, but not Microsoft Internet Explorer. Figure 3 shows the FAIR report 1 html output opened in Google Chrome, saved as PDF, and then displayed in Adobe Acrobat from which the screen shot is extracted.



AS9102 Rev A First Article Inspection — **Sheet 1 of 2**

Form 3: Characteristic Accountability, Verification and Compatability Evaluation

Characteristic Accountability				Inspection/Test Results			Optional Fields
5. Char No.	**6. Reference Location**	**7. Characteristic Designator**	**8. Requirement**	**9. Results**	**10. Designed Tooling**	**11. Non-Conformance Number**	14. [Insert columns, etc., as required by Organization or Customer]
5	sheet2_solid4 SHEET1 C2	MINOR	ToleranceValue=4 DatumReferenceFrameId=12	PASS	CMM	NA	PointProfile
1	sheet2_solid4 SHEET1 D3	REF	NonTolerance=MEASURED TargetValue=2466.729248046875 Direction=XAXIS	BASIC	CMM	NA	LinearCoordinate
2	sheet2_solid4 SHEET1 D3	MINOR	MaxValue=0.2 MinValue=-0.2 DefinedAsLimit=false TargetValue=774.26989746093795 Direction=YAXIS	PASS	CMM	NA	LinearCoordinate
3	sheet2_solid4 SHEET1 D3	MAJOR	MaxValue=945.20274658203095 MinValue=944.80274658203086 DefinedAsLimit=true Direction=ZAXIS	PASS	CMM	NA	LinearCoordinate
4	sheet2_solid4 SHEET1 B3	CRITICAL	ToleranceValue=1.5 OuterDisposition=1 DatumReferenceFrameId=12	FAIL	CMM	1234	PointProfile
6	sheet2_solid4 SHEET1 C1	MINOR	MaxValue=0.4 MinValue=-0.4 DefinedAsLimit=false TargetValue=10	FAIL	CMM	1234	Diameter
7	sheet2_solid4 SHEET1 C1	CRITICAL	ToleranceValue=1 DatumReferenceFrameId=50 MaterialCondition=MAXIMUM DiametricalZone=	PASS	GAGE PINS	NA	Position
8	sheet2_solid4 SHEET1 C3	CRITICAL	MaxValue=10.4 MinValue=9.6 DefinedAsLimit=true	PASS	CALIPERS	NA	Diameter
9	sheet2_solid4 SHEET1 C3	MINOR	ToleranceValue=1 DatumReferenceFrameId=68 MaterialCondition=REGARDLESS DiametricalZone=	FAIL	CMM	1234	Position



AS9102 Rev A First Article Inspection — **Sheet 2 of 2**

Form 3: Characteristic Accountability, Verification and Compatability Evaluation

Characteristic Accountability				Inspection/Test Results			Optional Fields
5. Char No.	**6. Reference Location**	**7. Characteristic Designator**	**8. Requirement**	**9. Results**	**10. Designed Tooling**	**11. Non-Conformance Number**	14. [Insert columns, etc., as required by Organization or Customer]
10		CRITICAL	NonTolerance=SET TargetValue=30	BASIC			Diameter
DIST1	sheet2_solid4 SHEET1 B2	CRITICAL	MaxValue=0.5 MinValue=-0.5 DefinedAsLimit=false TargetValue=81.208839738425993 AnalysisMode=THREEDIMENSIONAL	PASS	CMM		DistanceBetween

The signature indicates that all characteristics are accounted for; meeting requirements or are properly documented for disposition.

12. Prepared By John Doe	**13. Date** 2014-07-30T11:14:06

Figure 3 FAIR Form 3 Pages 1 and 2

Simplifying Null Pointer Access

Xerces and CodeSynthesis rely on the use of pointers to navigate the XML tree. A null pointer has a value reserved for indicating that the pointer does not refer to a valid object. A null pointer can occur often in XML mappings to signify that an optional element branch is not instantiated. Since QIF XSD specifies numerous optional elements, and because these references are represented by a null pointer in C++, it is important to either test every XML element variable access to ensure that it is valid (i.e., non-null) or develop some mechanism to simplify the cascading of pointer variables.

Because of the multitude of pointers and the possibility of a NULL pointer exception, Microsoft Windows and Ubuntu Linux C++ code was developed to trap all signals/exceptions thrown when accessing NULL pointer. One easy way to perform a safe pointer access is with a preprocessor macro, defined below as SAFEFETCH, which encloses all reference chains in a try block, and will catch any exception, and if an exception is thrown, the X variable will be assigned a default Z value.

```
#define SAFEFETCH(X,Y,Z) \
        try { X=Y; } catch(...) { X=Z;}

std::string version;
SAFEFETCH(version, qif->Version()->ThisInstanceQPId(), "");
```

However, this assumes that a zero pointer access will throw an exception, which is not handled in a standard C++ mechanism by either the Microsoft Windows or Linux operating systems.

For Windows and Visual C++ support, the structured exception handling (SEH) was enabled, to catch common hardware and Operating System (OS) signals – such as divide by zero, access a null pointer, etc. Structured exceptions are provided by Windows, with support from the kernel, and were mapped into C++ standard exception. With such an exception, the default assignment in SAFEFETCH could be performed and repetitive zero pointer checks could be skipped.

SEH is not handled by the Gnu C++ compiler, gcc (GNU Project). Instead, C++ code was added to handle signals that are mapped into a C++ exception. Again, this functionality is needed to catch the signal resulting from a (0)->x which triggers an SIGENV interrupt in gcc. Code was added to map signal handling into C++ exceptions, so that (0)->x now causes a C++ std::exception, which allows the SAFEFETCH macro to operate properly.

Summary

This document details an automated approach to generating FAIR reports from QIF XML. The approach should save companies substantial amounts of time and resources in filling out FAIR reports. Much detail was given to the C++ software development used to parse, navigate, and extract relevant FAIR information from the QIF XML. In particular, the software applications and libraries including XSD from CodeSynthesis and Xerces from Mozilla Developer Network were given extensive and detailed explanation in the application of these tools in the FAIR generation. Underlying C++ code that parses the QIF XML and translates the results in the FAIR reports was described. This QIF C++ code is available on the usnistgov github web site found at https://github.com/usnistgov/QIF.

Future work includes QIF document verification that the QIF contains all the quality and measurement results required of the FAIR reports. Additionally, the AS 9102 standard has a new revision “B” that will require minor modifications to the previously described code.

Disclaimer

Commercial equipment and software, many of which are either registered or trademarked, are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

GNU Project. (n.d.). *GCC, the GNU Compiler Collection*. Retrieved 8 22, 2015, from http://www.gnu.org/software/gcc/gcc.html

CodeSynthesis. (2015). Retrieved 6 17, 2015, from http://wiki.codesynthesis.com/XSD

Dimensional Metrology Standards Consortium . (2015). *Download QIF*. Retrieved 7 7, 2015, from http://qifstandards.org/download-qif

Kramer, T. (2014). *Preparing an AS9102a First Article Inspection (FAI) Report from QIF 2.0 XML Instance Files.*

Mozilla. (2015). *Xerces-C++Parser*. Retrieved 6 23, 2015, from http://xerces.apache.org/xerces-c/

Plauger, P. J. (2001). *The C++ Standard Template Library.* Prentice Hall.

Quality Magazine. (2007). *Enhance First Article Inspection*. Retrieved 6 23, 2015, from http://www.qualitymag.com/articles/85036-enhance-first-article-inspection

SME International. (2014, 10 6). *AS9102 - Aerospace First Article Inspection Requirement Revision B*. Retrieved 8 22, 2015, from http://standards.sae.org/as9102b/

United Nations Industrial Development Organization. (2006). *Working paper - product quality: A guide for small and medium-sized enterprises.*

Wikipedia. (2015). *First article inspection*. Retrieved 6 23, 2015, from https://en.wikipedia.org/wiki/First_article_inspection

World Wide Web Consortium. (1999). *XML Path Language (XPath)*. Retrieved 7 7, 2015, from http://www.w3.org/TR/xpath

World Wide Web Consortium. (2004). *Document Object Model (DOM)*. Retrieved 7 7, 2015, from http://www.w3.org/DOM