NOKIA

DITA Specialisation C++ Design Sketch

Creator:	Paul Ross
Function	Senior Technology Architect D RD SD SSS System Doc & Tools GB
Approvers:	Jonathan Harrington, Valentine Ogier-Galland, Robert Shaffer, Abigail Sidford, Paul Ross, Eleanor Weavers of the System Documentation Tools Team D RD SD SSS System Doc & Tools GB

Change History

Version	Status	Date	Handled by	Comments
0.1	Draft	2009-09-05	Paul Ross	Initial Draft
0.2	Draft	2009-11-05	Paul Ross	Updated, removed sections that are irrelevant from the Design Sketch Template.
0.6.1	Draft	2010-05-21	Paul Ross	Brought up to date with version 0.6.0 of the DTDs. Extensively revised.
0.6.2	Draft	2010-06-14	Paul Ross	Minor review changes.
0.6.3	Draft	2010-06-15	Paul Ross	Minor review changes.
0.6.4	Draft	2010-06-15	Paul Ross	Removed irrelevant sections.
0.7.0	Draft	2011-03-11	Paul Ross	Updated to version 0.7.0 of the DTDs

Contents:

1. SUMMARY

This document describes the specialisation of DITA XML 1.1 to describe the interfaces to code written in the C++ language[2]. This work builds on the existing DITA specialisations for software APIs [R10].

This document does not seek to describe every element in the specialisation instead it seeks to identify, clarify and justify various design choices that were made during the course of the specialisation.

This does not describe any DITA specialisation for the C pre-processor language.

1.1 Audience

This is intended for anyone implementing a DITA specialisation for C++ APIs or working on an implementation of anything that generates XML to that specialisation.

The reader is expected to be familiar with the C++ standard [R2] and DITA [R7].

All designs are really just a set of compromises so if readers enjoy controversy they are directed to the following sections:

4.2 “Design Choices” on page 22 – provides some background for why certain decisions were made.

4.4 “Variance from Prior Art” on page 29 – describes why certain decisions contradict existing implementations (notional or real).

1.2 Contributors

This is the work of many minds, in alphabetical order: Jonathan Harrington, Valentine Ogier-Galland, Robert Shaffer, Abigail Sidford, Paul Ross, Eleanor Weavers.

1.3 Implementation Limits

This DITA specialisation provides the means to describe the interfaces to C++ but not any implementation details of those interfaces. There are other standards around such as KDM[1] that claim to describe implementation information.

As such this standard is suitable for tools, such as Doxygen and GCC_XML, that make no claim to comprehend implementation information.

[1] Wikipedia’s page on KDM: http://en.wikipedia.org/wiki/Knowledge_Discovery_Metamodel

1.4 Annotation

Text in italics usually refers to words and phrases that appear in the C++ standard [R2].

Test thus TODO: Clean fridge, put cat out or TBD represent unfinished work in this version of the document.

Text thus describes reasons for rejecting a design:

REJECTED: ...

And this describes which of several designs were chosen:

ACCEPTED: ....

2. DESIGN BACKGROUND

2.1 Design Assumptions

The following assumptions are made throughout this document:

1. The C++0x standard [R3] is not relevant.
2. The C Pre-processing language[2] is not relevant apart from `#define` declarations.

2.2 Design Risks & Issues

The following risks are identified:

1. Any of the above assumptions are false.

2.3 Design Principles

This design is intended to adhere to the principles of *completeness* and *minimalism*.

2.3.1 Design Principle: Completeness

The design shall capture all the significant characteristics of C++ code suitable for producing documentation for developers using that code.

2.3.2 Design Principle: Minimalism

The design should not provide additional information beyond that in the C++ standard and common practice. A consequence is that further inferences can (or must be) be drawn by post-processors.

For example given this code:

```
class Parent
{
        virtual void Virtual();
};

class ChildOne : public Parent
{
        virtual void Virtual();
};

class ChildTwo : public Parent {};
```

So a ‘complete’ representation might say:

1. `ChildOne` inherits from `Parent`

[2] See ISO/IEC 9899:1999 (E) 6.10 Preprocessing directives.

2. `ChildTwo` inherits from `Parent`
3. `ChildOne::Virtual()` re-implements `Parent::Virtual()`
4. `Parent` has derived classes `ChildOne, ChildTwo`
5. `Parent::Virtual()` is re-implemented by `ChildOne` (but not by `ChildTwo`)

However the last two points are not minimal as they duplicate information that can be inferred from the first three. This duplicate information is problematic for the following reasons at least:

- There is no clear way of handling a conflict. For example if the representation of `ChildTwo` claims it derives from `Parent` (point 2 above) but `Parent` fails to claim a relationship with `ChildTwo` (point 4 above) then what is the appropriate way to resolve this?
- A conformant processor might not be aware of all of the relationships. For example the processor might run on a library that defines Parent but subsequently run on code that uses that library. The first run can not know about the second therefore would have incomplete, thus misleading, information.
- In a multi-component world the base class may be seen in many different contexts. If they differ then which is the true one? If they are always the same by being minimal then the base class topics from different components can be allowed to overwrite each other without the fear of losing information.

A example of this principle is that having singly linked lists of information is more minimal than doubly linked lists.

However having *no* duplication is not the goal of minimalism. In some cases duplication might be desirable, for example consider the following code:

```
class Base
{
public:
    inline const char * Func(const char* buf, int index) volatile const;
};
```

And three representations of the function `Func()`:

```
1: GUID-33BEE1B4-20A1-392B-89B3-DA5D4F46418E
2: const char *Func(const char *buf, int index) volatile const
3: Base::Func(const char *int)const volatile
```

There is not necessarily duplication going on here as:

- 1 is potentially a duplicate of 2 or 3, given a few simple rules, but there may be a valid use case to have this presented that way (for example for a specific content management system).
- 2 and 3 are not duplicates of 1 as 1 uses a one-way hash.
- 3 is a duplicate of 2 but only if the information in 2 is parsed according to C++ rules, also note the rearrangement of const/volatile. Parsing C/C++ might not be desirable.

So having all three, whilst not being minimal, is *acceptably* minimal in this design.

2.4 Element and Attribute Naming Conventions

Elements and attributes should follow the following rules:

1. The element name prefix is `‘cxx’`. Note that this is true for `cxxDefine` that actually describes part of the C pre-processing language.
2. If the element represents anything that is a C++ keyword then the element name shall be the prefix followed by the C++ keyword with the first letter capitalised.
3. If the attribute represents an entity that is defined by a C++ keyword then that attribute name shall be the C++ keyword.
4. Attributes and elements that are represented as acronyms and other abbreviations are strongly discouraged. If present however they should follow these rules:
    a. An abbreviation must be unique regardless of the element it appears in.
    b. Abbreviations are in uppercase.
    c. Abbreviations are documented using xsd:annotation in the XML schema and/or a suitable comment in the DTD
    d. The abbreviation, and its documentation should refer to a standard of origin.
5. An element that is a reference to an element that in this specialisation (apart from DITA Map references) is the referent element name but all lowercase.
6. An element that is a reference to an element that in this specialisation from DITA Map is the referent element name followed by `Ref`.
7. Names should be in full.

Examples:

Rule	**Example**
1.	`<cxxFunction ...>`
2.	`<cxxTypedef ...>`
3.	`extern="yes"`
4.	Rather than:

```
<xs:element name="...">
    ...
    <xs:attribute name="storage-class-specifier" use="optional">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="auto"/>
          <xs:enumeration value="register"/>
          <xs:enumeration value="static"/>
          <xs:enumeration value="extern"/>
          <xs:enumeration value="mutable"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
</xs:element>
```

An abbreviated version would be, (differences shown in **bold blue**):

```
<xs:element name="...">
    ...
    <xs:attribute name="SCS" use="optional">
        <xs:annotation>
            <xs:documentation xml:lang=”en”>
```

```
                    Abbreviation for storage-class-specifier. See:
                    ISO/IEC ISO/IEC 14882:1998(E)
                    7.1.1 Storage class specifiers [dcl.stc]
                    </xs:documentation>
                </xs:annotation>
              <xs:simpleType>
                <xs:restriction base="xs:token">
                  <xs:enumeration value="auto"/>
                  <xs:enumeration value="register"/>
                  <xs:enumeration value="static"/>
                  <xs:enumeration value="extern"/>
                  <xs:enumeration value="mutable"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
        </xs:element>
```

5. Given:
`<cxxClass ...>...</cxxClass>`
Then a reference to this element would be:
`<cxxclass ...>...</cxxclass>`

6. Given:
`<cxxClass ...>...</cxxClass>`
Then a map reference to this element would be:
`<cxxClassRef ...>...</cxxClassRef>`

2.5 DITA XML Standard

DITA is an open XML standard for technical documentation[3]. A notable feature if DITA is its ability to specialise from the general standard to one more useful to the task at hand. Specialisation falls into two categories structural and domain. Structural specialisation seeks to limit, expand or rearrange elements to suit a particular authoring environment. Domain (or industry) specialisation seeks to create an interchangeable standard for a particular technology[4].

DITA specialisation is implemented through inheritance. Some of these specialisations already exist in the DITA community; the notable exception is a C++ API reference (although a Java API reference specialisation exists)[5].

The existing API specialisation defines the following generic types to describe API collections in a particular language:

Type	**Description**	**Example when specialised for a language**
apiPackage	Defines a top-level container for the general-	A C++ Library or

[3] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=dita

[4] Examples of *domain* specialisation are: Pharmaceutical Content, Learning and Training Content, Machine Industry documentation and Semiconductor Information.

[5] The existing specialisations date back to 2006 and do not appear to have been maintained since then. There is no existing opensource reference implementation that can convert source code to DITA. There is a proprietary Java source to DITA Eclipse plugin at http://www.ibm.com/developerworks/library/x-DITAdoclet/ . This page also gives the information: “Java API pilot project lead: Mariana Alupului”.

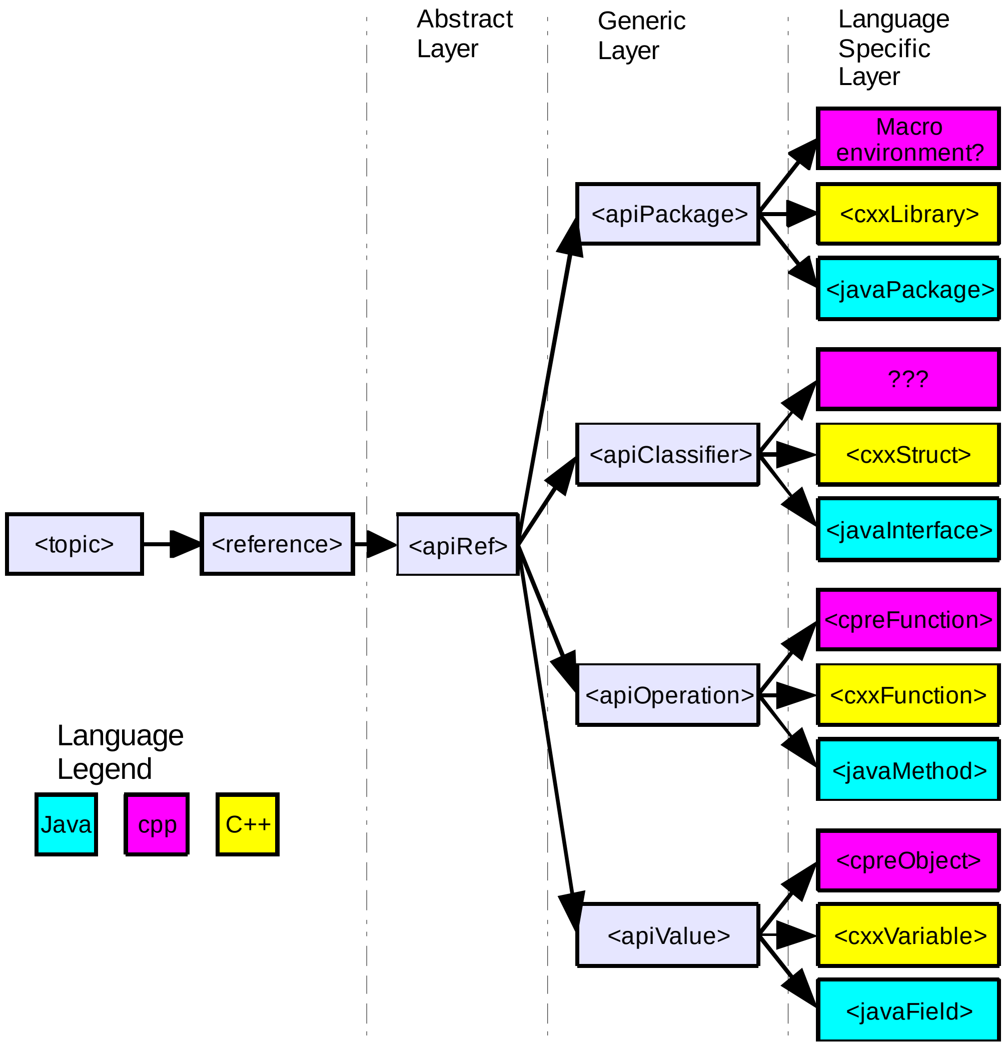
Type	Description	Example when specialised for a language
	purpose topic types for a library, package, header, module, namespace, class members, or other named grouping for the entities provided by the API.	component.
apiClassifier	Defines a top-level container for the generic topic types such as class, interface, typedef, or other named construct for typing values.	A C++ class.
apiOperation	Defines a top-level container for general-purpose topic types for a function, method, subroutine, procedure, event handler, or other named unit of executable code.	A C++ function.
apiValue	Defines a subordinate container to the <apiOperation> element for generic topic types such as constants, variables, enumerations, structures, or other named values.	A C++ variable.

In addition there are the following special purpose types which are especially concerned with linking DITA documents:

Type	Description
api-domain reference	Defines the elements referring to API items within API topics or other DITA topic types, such as concept or task.
apiMap	Provides the relationships among a set of application programming interface (API) topic types, and can be used to create a Table of Contents (TOC), aggregate Java API topics into a PDF document, or to create links between topics in output.

In addition to this *generic layer* there is the *language specific layer*, currently there is a specialisation for the Java language only.

The DITA specialisation by inheritance is illustrated in the following diagram, not all elements are shown but it should give a flavour of the mechanism:

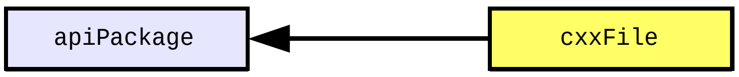


3. DETAILED DESIGN

This does not cover the design or description of every element, for that see the public documentation for the cxxapiref plugin. This sections is really a set of design notes of 'interesting' areas.

3.1 Derivation from apiPackage

The apiPackage module defines a top-level container for the general-purpose topic types for a library, package, header, module or other named grouping for the entities provided by the API.



Derivation path:

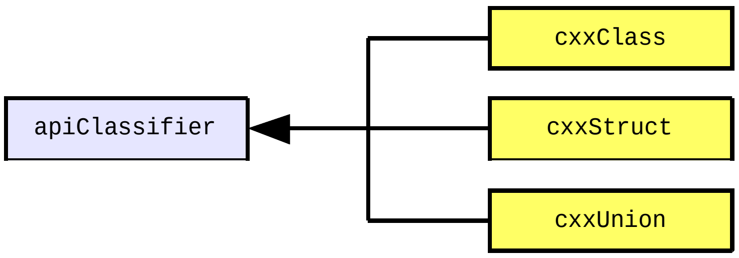
```
topic/topic reference/reference apiRef/apiRef apiPackage/apiPackage
```

3.1.1 cxxFile

The cxxFile element contains all the global APIs of a component.

3.2 Derivation from apiClassifier

The apiClassifier module defines a top-level container for the generic topic types such as class, interface, typedef, or other named construct for typing values.



Derivation path:

```
topic/topic reference/reference apiRef/apiRef apiClassifier/apiClassifier
```

3.2.1 cxxClass

The cxxClass provides the description of a C++ class.

3.2.2 cxxStruct

Element name is `cxxStruct`, content model is near identical to `cxxClass` on page 13 with some obvious prefix changes.

3.2.3 cxxUnion

Element name is `cxxUnion`, content model is nearly identical to `cxxClass` on page 13, the exceptions are that[6]:

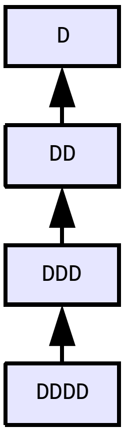
- A union can not have virtual functions.
- A union shall not have base classes.
- A union shall not be used as a base class.
- An object of a class with a non-trivial constructor, a non-trivial copy constructor, a non-trivial destructor, or a non-trivial copy assignment operator cannot be a member of a union, nor can an array of such objects.
- If a union contains a static data member, or a member of reference type, the program is ill-formed.

3.2.4 Elements that Describe Inheritance.

The element names are abridged from the C++standard (see footnotes). Given this example source:

```
class D                  { /* ... */ };
class DD   : public D    { /* ... */ };
class DDD  : public DD   { /* ... */ };
class DDDD : public DDD  { /* ... */ };
```

Or pictorially:



These terms are defined:

Terminology in the standard	Description
direct base class[7]	The immediate superclass(es) of the class. The direct base class(es) appear in the *base-specifier* of the class. In the example `DDD` is the direct base class of `DDDD`, `DD` is the direct base class of `DDD` and `D` is the direct base classes of `DD`.

[6] ISO/IEC 14882:1998(E) Section 9.5 Unions [class.union] Paragraph 1.

[7] ISO/IEC 14882:1998(E) Section 10 Derived classes [class.derived], paragraph 1.

base class[8]	Superclass(es) of the class. A class B is a base class of a class D if it is a direct base class of D or a direct base class of one of D's base classes. In the example D, DD, DDD are base classes of DDDD, both D, DD are base classes of DDD and D is a base class of DD.
indirect base class[9]	An indirect base class is a base class but not a direct base class In the example D and DD are indirect base classes of DDDD. and D is the indirect base class of DDD.
most derived class[10]	Any class that is not a base class but inherits from a base class. In the example DDDD is the most derived class.
None or *most base class*. [Here we are using complimentary terminology to *most derived class*.]	Any class that is a base class but does not inherit from a base class. In the example D is the *most base class*.

[I am trying here to avoid the confusion that is in the existing Java specialisation[11].]

Note that multiple inheritance means that many of these elements can appear once or more. For example, given[12]:

```
class B                                { /* ... */ };
class X : virtual public B             { /* ... */ };
class Y : virtual public B             { /* ... */ };
class Z : public B                     { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

[8] ISO/IEC 14882:1998(E) Section 10 Derived classes [class.derived], paragraph 1.

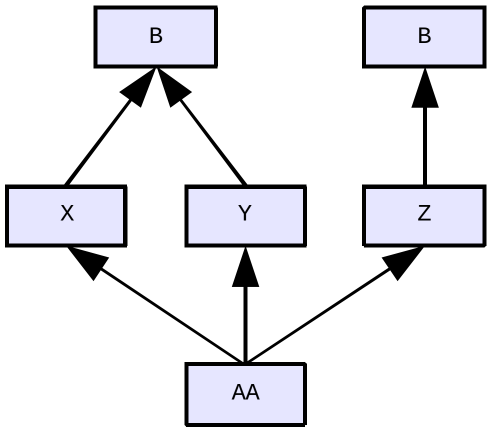
[9] ISO/IEC 14882:1998(E) Section 10 Derived classes [class.derived], paragraph 1.

[10] ISO/IEC 14882:1998(E) Section 10.1 Multiple base classes [class.mi], paragraph 4 and Section 12.6.2 Initializing bases and members [class.base.init], paragraph 5

[11] For example see: http://sourceforge.net/projects/dita-ot/files/Plug-in_%20javaapiref/javaapiref-0.8/javaapiref0.8.zip/download
And the file: javaapiref/doc/html/javaClass/javaBaseClass.html which says in the first sentence: "The <javaBaseClass> element within the Superopt declaration specifies the **direct superclass** of the current class." then in the next sentence "The <javaBaseClass> element is the **root** of the class hierarchy."

[12] ISO/IEC 14882:1998(E) Section 10.1 Multiple base classes [class.mi], paragraph 6.

Or, pictorially:



Class AA has base classes B, X, Y, Z, and direct base classes X, Y, Z and so on.

For reasons of minimalism only the direct base class(s) are referred to, the element being:

```
<cxxClassDerivations>
    <cxxClassDerivation>
        <cxxClassAccessSpecifier value="public"/>
        <cxxClassBaseClass href="class_base">Base</cxxClassBaseClass>
    </cxxClassDerivation>
    ...
</cxxClassDerivations>
```

3.2.5 cxx...Template...

A *template-declaration* can apply to a function or a class[13]. The syntax for a *template-declaration* is:

template-declaration:
  export$_{opt}$ template < template-parameter-list > declaration

template-parameter-list:
  template-parameter
  template-parameter-list , template-parameter

The syntax for template-parameters is given in ISO/IEC 14882:1998(E) 14.1 “Template parameters [temp.param]” but does not need to be decomposed further for the purposes of this specialisation.

So these classes:

```
template<class T> class myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map {
C<K> key;
```

[13] ISO/IEC 14882:1998(E) 14 Templates [temp]

```
C<V> value;
    // ...
};
```

Would result in two topics:

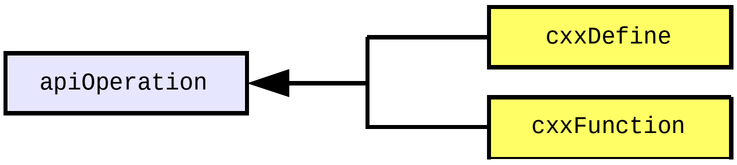
```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<!DOCTYPE cxxClass PUBLIC "-//NOKIA//DTD DITA C++ API Class Reference Type v0.7.0//EN"
"dtd/cxxClass.dtd" >
<cxxClass id="classmyarray">
  <apiName>myarray</apiName>
  <shortdesc/>
  <cxxClassDetail>
    <cxxClassDefinition>
      <cxxClassAccessSpecifier value="public"/>
      <cxxClassTemplateParameters>
        <cxxClassTemplateParameter>
          <cxxClassTemplateParameterType>class T</cxxClassTemplateParameterType>
        </cxxClassTemplateParameter>
      </cxxClassTemplateParameters>
      <cxxClassAPIItemLocation>
        <cxxClassDeclarationFile name="filePath" value="..."/>
        <cxxClassDeclarationFileLineStart name="lineNumber" value="1"/>
        <cxxClassDeclarationFileLineEnd name="lineNumber" value="1"/>
      </cxxClassAPIItemLocation>
    </cxxClassDefinition>
    <apiDesc/>
  </cxxClassDetail>
</cxxClass>
```

And:

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<!DOCTYPE cxxClass PUBLIC "-//NOKIA//DTD DITA C++ API Class Reference Type v0.7.0//EN"
"dtd/cxxClass.dtd" >
<cxxClass id="class_map">
  <apiName>Map</apiName>
  <shortdesc/>
  <cxxClassDetail>
    <cxxClassDefinition>
      <cxxClassAccessSpecifier value="public"/>
      <cxxClassTemplateParameters>
        <cxxClassTemplateParameter>
          <cxxClassTemplateParameterType>class K</cxxClassTemplateParameterType>
        </cxxClassTemplateParameter>
        <cxxClassTemplateParameter>
          <cxxClassTemplateParameterType>class V</cxxClassTemplateParameterType>
        </cxxClassTemplateParameter>
        <cxxClassTemplateParameter>
          <cxxClassTemplateParameterType>template&lt; class T &gt;
class</cxxClassTemplateParameterType>
        </cxxClassTemplateParameter>
      </cxxClassTemplateParameters>
      <cxxClassAPIItemLocation>
        <cxxClassDeclarationFile name="filePath"
value="C:/wip/Doxygen/trunk/test/PaulRo/design_doc/src/src.h"/>
        <cxxClassDeclarationFileLineStart name="lineNumber" value="4"/>
        <cxxClassDeclarationFileLineEnd name="lineNumber" value="8"/>
      </cxxClassAPIItemLocation>
    </cxxClassDefinition>
    <apiDesc/>
  </cxxClassDetail>
</cxxClass>
```

3.3 Derivation from apiOperation

The apiOperation module defines a top-level container for general-purpose topic types for a function, method, subroutine, procedure, event handler, or other named unit of executable code.



Derivation path:

```
topic/topic reference/reference apiRef/apiRef apiOperation/apiOperation
```

3.3.1 cxxFunction

The `cxxFunction` element contains the description of both *member functions* and non-*member functions* (global functions).

3.3.2 Was: cxxSpecialMemberFunction

The default constructor[14], copy constructor and copy assignment operator[15], and destructor[16] (12.4) are *special member functions*[17].

There may be restrictions on implementations that produce DITA to this standard that limit their ability to spot whether a *special member function* will be created.

```
struct X {
    // NOTE: X::X(const X&); not declared
};

void f(const X& aX) {
    X myX(aX); // Causes compiler generated X::X(const X&)
    ...
}
```

REJECTED: This is a language implementation detail.

3.3.3 cxxDefine

The cxxDefine element contains the description of *preprocessing directives*, be they *function like* or *object like* macros. Actually this is part of the C pre-processing language rather than C++ but we still use the `cxx` prefix. They differ from functions in numerous ways such as no types, overloading, re-implementation etc.

[14] ISO/IEC 14882:1998(E) Section 12.1 Constructors [class.ctor]

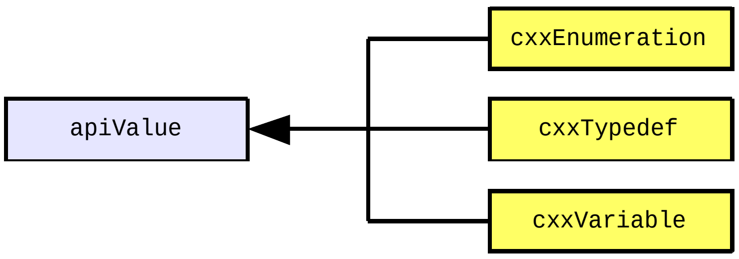
[15] ISO/IEC 14882:1998(E) Section 12.8 Copying class objects [class.copy]

[16] ISO/IEC 14882:1998(E) Section 12.4 Destructors [class.dtor]

[17] ISO/IEC 14882:1998(E) Section 12 Special member functions [special]

3.4 Derivation from apiValue

The apiValue module defines elements for generic topic types such as constants, variables, enumerations, structures, or other named values.



Derivation path:

```
topic/topic reference/reference apiRef/apiRef apiValue/apiValue
```

3.4.1 cxxEnumeration

As enum is a type-specifier[18].

3.4.2 cxxTypedef

Describes typedef'ing.

3.4.3 cxxVariable

The `cxxVariable` element contains the description of both *member variables* and non-*member variables* (globals).

3.5 Derivation from apiDomain Module

The apiDomain module defines the elements referring to API items within API topics or other DITA topic types, such as concept or task. This will be specialised to be the cxxAPIDomain module[19].

Element names are the same as the element name that they refer to but all lower case.

3.5.1 Derivation from apipackage

Derivation path:

```
topic/xref pr-d/xref api-d/apipackage
```

Element	Links to	Notes
`cxxprogram`	`cxxProgram`	Not yet defined.
`cxxfile`	`cxxFile`	

[18] ISO/IEC 14882:1998(E) 7.1.5 [dcl.type]

[19] The naming convention follows the convention used for the Java specialisation.

3.5.2 Derivation from apiclassifier

Derivation path:

```
topic/xref pr-d/xref api-d/apiclassifier
```

Element	Links to	Notes
cxxclass	cxxClass	
cxxdefine	cxxDefine	
cxxenumeration	cxxEnumeration	
cxxfile	cxxFile	
cxxfunction	cxxFunction	
cxxnamespace	cxxNamespace	
cxxstruct	cxxStruct	
cxxtypedef	cxxTypedef	
cxxunion	cxxUnion	
cxxvariable	cxxVariable	

3.5.3 Derivation from apioperation

Derivation path:

```
topic/xref pr-d/xref api-d/apioperation
```

3.5.4 Derivation from apivalue

Derivation path:

```
topic/xref pr-d/xref api-d/apivalue
```

3.6 Derivation from apiMap Module

DITA Maps provide the highest level structure to the documentation and they consist of a set of references to topics, this is the cxxAPIMap[20] module.

3.6.1 cxxAPIMap

Derivation path:

```
map/map apiMap/apiMap
```

Element	Links to	Notes
cxxAPIMap	cxxAPIMap	

[20] The naming convention follows the convention used for the Java specialisation.

3.6.2 cxxAPIMapRef

This describes map references that reference `cxx` topics.

Derivation path:

```
map/topicref apiMap/apiItemRef
```

Element	Links to	Notes
`cxxClassRef`	`cxxClass`	
`cxxDefineRef`	`cxxDefine`	
`cxxEnumerationRef`	`cxxEnumeration`	
`cxxFileRef`	`cxxFile`	
`cxxFunctionRef`	`cxxFunction`	
`cxxStructRef`	`cxxStruct`	
`cxxTypedefRef`	`cxxTypedef`	
`cxxUnionRef`	`cxxUnion`	
`cxxVariableRef`	`cxxVariable`	

4. EXTENDED DESIGN INFORMATION

This section describes additional design information that does not readily fit elsewhere.

4.1 Design Lacunae

This section describes current omissions in the design.

4.1.1 cxxProgram

This version of the design (version 0.7.0) does cover the `cxxProgram` element (or equivalent).

A reasonable definition of a program consists of one or more translation units (clause 2) linked together. A translation unit consists of a sequence of declarations[21].

Another thing to consider is the notion of a “program” as an executable or library (dynamic or static).

4.1.2 Representing Source Code Verbatim

This version of the design (version 0.7.0) does not cover the verbatim representation of source code. There are many other tool that can do this.

[21] ISO/IEC 14882:1998(E) Section 3.5 Program and linkage [basic.link], paragraph 1.

If it was desired to represent source code in DITA it is worth thinking about this in a language independent way. Some possibilities:

- Reproducing it all in a `<codeblock>` element.
- Using the DITA Programming Elements[22]
- Most if not all languages can be decomposed into tokens, for example both the C[R1] and the C++[R2] standard describe the lexical decomposition to tokens thus:

-
  ```
  token:
      keyword
      identifier
      constant
      string-literal
      punctuator
  ```

4.2 Design Choices

This section describes some of the alternatives that were explored in the development of this design sketch and why those alternatives were discarded in favour of the current design.

4.2.1 cxxProgram rather than cxxLibrary

ISO/IEC 14882:1998(E) Section 3.5 Program and linkage [basic.link], paragraph 1 states "A *program* consists of one or more *translation units* (clause 2) linked together. A translation unit consists of a sequence of declarations." The standard uses *library* to refer to the *C++ Standard Library*.

The same standard also defines a *well-formed program* as "a C + + program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule"[23].

4.2.2 Choice of 'cxx' Prefix

The requirement is, as a minimum to support C++ but this, effectively, means supporting the C pre-processor. Supporting multiple languages means that a shrewd choice of element naming conventions has to be made. Judging by the proposal in the apiref documentation[24] the suggestion is that a language prefix be used for each language, for example `<javaMethod>` `<perlMethod>` `<vbMethod>` etc. The suggestion for C++ appears to be '`cpp`' but this is regarded as a confusing choice for the following reasons.

1. Existing C++ standards[25] uses 'cpp' as section identifiers for the *pre-processing* part of the standard.

[22] DITA DITA Version 1.1 Language Specification, chapter 14.

[23] ISO/IEC 14882:1998(E) Section 1.3.14 [defns.well.formed]

[24] See: http://sourceforge.net/projects/dita-ot/files/Plug-in_%20apiref/apiref-0.8/apiref0.8.zip/download

And the file: apiref/doc/html/guidelineSpecialization/language-specificapireference.html

[25] 'C++ 98' i.e. ISO/IEC 14882:1998(E) and 'C++0x' i.e. ISO/IEC IS 14882.

2. The GNU executable `cpp` is widely used as a *pre-processor*.

To avoid confusion between C++ and C/C++ pre-processing (an entirely separate language) the following prefixes are to be used:

Language	**Element Prefix**	**Example**	**Rationale**
Java	`java`	`javaPackage`	Prior art.
C	`c`	`cFunction`	Unambiguous and it is the complete language name.
C Preprocessor	`cpre`	`cpreMacro`	Unambiguous and not likely to be confused with C or C++.
C++	`cxx`	`cxxClass`	Distinct from C and C pre-processor, .cxx is a common file extension for C++ source code (in the wider world at least). NOTE: “c++” is not allowed in an element name by the XML specification.

4.2.3 Class, Struct and Union

It is commonly accepted that a struct is the primary record in C++ with a class as syntactic sugar for a struct with its members private by default. The temptation was to follow this in this design but, since the C++ standard [R2] Section 9 states ‘A class is a type’ with a *class-key* of `class | struct | union`. This design follows that convention so cxxClass, cxxStruct and cxxUnion are peer records and do not derive from one another.

An alternate design was considered as follows[26]:

```
<cxxClass class-key="class" ...>...<cxxClass>
<cxxClass class-key="struct" ...>...<cxxClass>
<cxxClass class-key="union" ...>...<cxxClass>
```

But this was thought to be too confusing.

See also section 4.2.7 “Why do cxxClass, cxxStruct, cxxUnion do not derive from, say cxxRecord” on page 25 for some discussion of this.

4.2.4 Which Members are Included in Class Topics

Presumably it is fairly non-controversial that, given the following source code where class B is in global namespace:

```
class B {
public:
    void f();
};
```

[26] The attribute name ‘class-key’ comes from ISO/IEC 14882:1998(E) 9 Classes [class], paragraph 1.

That the following would happen:

- The class ‘B’ would have a single topic describing it.
- That topic would contain the description of ‘f’.

In other words something like this:

```
<cxxClass id=”B”>
  <apiName>B</apiName>
  <shortdesc>...</shortdesc>
  <cxxClassDetail>...</cxxClassDetail>
  <cxxMemberFunction id="B::f()">
    <apiName>f</apiName>
    <shortdesc>...</shortdesc>
    <cxxFunctionDetail >...</cxxFunctionDetail >
  </cxxMemberFunction>
</cxxClass>
```

However given nested classes[27] in this code:

```
class enclose {
public:
  class inner {
    void f(int i);
  };
};
```

Does the class ‘inner’ get is own topic that is referenced by ‘enclose’, for example:

```
<cxxClass id=”...”>
  <apiName>enclose</apiName>
  <shortdesc>...</shortdesc>
  <cxxClassDetail>...</cxxClassDetail>
  <cxxclass id="enclose::inner"></ cxxclass>
</cxxClass>
```

Or is it included in the enclosing class?

ACCEPTED: Each class is in a separate topic.

4.2.5 Members in Namespaces

A single namespace can contain a huge number of members, consider the namespace `std` for example. A single topic for `std` containing the descriptions of all members would be too unwieldy. Instead a single namespace topic will contain references to all members, each member being a topic.

4.2.6 Member Functions and Non-Member Functions

From a standards perspective the only difference between a member function and a non-member function is that the former has a prefix in its mangled name and can be virtual[28].

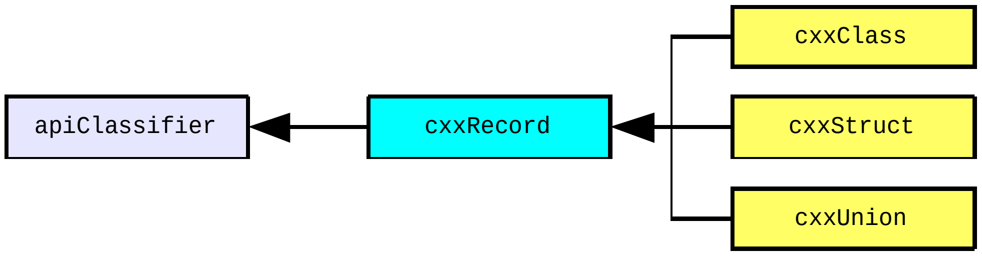
[27] ISO/IEC 14882:1998(E) 9.7 Nested class declarations [class.nest]

[28] ISO/IEC 14882:1998(E) Section 7.1.2 Function specifiers [dcl.fct.spec]

See also the section 4.2.8 “Why do cxxNonMemberFunction and cxxMemberFunction not derive from, say, cxxFunction” on page 25.

4.2.7 Why do cxxClass, cxxStruct, cxxUnion do not derive from, say cxxRecord

class/struct/union are have, from a C++ standard perspective, similar content. Since all three have a complex content model would it make sense fro them all to derive from a common element? Like this:



REJECTED: Excess complexity.

4.2.8 Why do cxxNonMemberFunction and cxxMemberFunction not derive from, say, cxxFunction

Similarly is it worth making two elements `cxxNonMemberFunction` and `cxxMemberFunction` that subclass `cxxFunction`? Probably not.

REJECTED: Only cxxFunction is used.

4.2.9 Do API IDs need some sort of namespace?

A namespace that would be unique to this standard could be used as a prefix and reduce (but not remove) the likelihood of broken links.

What are the pros and cons of this?

REJECTED: Until proven necessary. Note that the domain specific referencing mechanisim means that to link from outside the API reference to a Java class use <javaclass> and a C++ class use <cxxclass>. This will go some way to eliminating collisions.

4.2.10 How Should cxxClass, cxxStruct, cxxUnion Represent Inherited Members?

There seem to be the following techniques of representing an inherited member:

1. Reproduce inherited members in the derived class.
2. Link from the derived class to the inherited member in the base class.
3. Do not reproduce inherited members i.e. leave it implicit.

So given:

```
struct B {
    virtual void f;
};

struct D : public B {
    // Inherits void f();
};
```

Each technique produces two topics, the first topic being the `struct B`, this being identical for each techniques:

```
<cxxStruct id="B">
  <apiName>B</apiName>
  <shortdesc>...</shortdesc>
  <cxxClassDetail>...</cxxClassDetail>
  <cxxMemberFunction id="B::f()">
    <apiName>f</apiName>
    <shortdesc>...</shortdesc>
    <cxxMemberFunctionDetail>
      <cxxMemberFunctionDef>
        <cxxMemberFunctionAccess value="public"/>
        <cxxMemberFunctionVirtual/>
      </cxxMemberFunctionDef>
    </cxxMemberFunctionDetail>
  </cxxMemberFunction>
</cxxStruct>
```

Technique 1 reproduces the inherited function and would produce this for the `struct D`:

```
<cxxStruct id="D">
  <apiName>D</apiName>
  <shortdesc>...</shortdesc>
  <cxxClassDetail>
    <cxxClassDefinition>
      <cxxBaseClass>
        <cxxclass keyref="B"/>
      </cxxBaseClass>
    </cxxClassDefinition>
  </cxxClassDetail>
  <cxxMemberFunction id="D::f()">
    <apiName>f</apiName>
    <shortdesc>...</shortdesc>
    <cxxMemberFunctionDetail>
      <cxxMemberFunctionDef>
        <cxxMemberFunctionAccess value="public"/>
        <cxxMemberFunctionVirtual/>
      </cxxMemberFunctionDef>
    </cxxMemberFunctionDetail>
  </cxxMemberFunction>
</cxxStruct>
```

REJECTED: On the principle of minimalism.

Technique 2 links to inherited functions and would produce this for the `struct D`. I am assuming here that these references will be put in a special element:

```
<cxxStruct id="D">
  <apiName>D</apiName>
  <shortdesc>...</shortdesc>
  <cxxClassDetail>
    <cxxClassDefinition>
      <cxxBaseClass>
        <cxxclass keyref="B"/>
      </cxxBaseClass>
      <cxxClassInheritedMembers>
        <cxxmemberfunction keyref="B::f()"/>
      </cxxClassInheritedMembers>
    </cxxClassDefinition>
  </cxxClassDetail>
</cxxStruct>
```

The base class from which the member was inherited is implicit in the id attribute.

ACCEPTED: A reasonable compromise.

Technique 3 does not include inherited members and would produce this for the `struct D`:

```
<cxxStruct id="D">
  <apiName>D</apiName>
  <shortdesc>...</shortdesc>
  <cxxClassDetail>
    <cxxClassDefinition>
      <cxxBaseClass>
        <cxxclass keyref="B"/>
      </cxxBaseClass>
    </cxxClassDefinition>
  </cxxClassDetail>
</cxxStruct>
```

REJECTED: Valuable information is not represented.

Then there are similar issues with pure virtual functions, for example given this:

```
struct B {
    virtual void f() = 0;
};

struct D : B {
    virtual void f();
};

void D::f() { /* Implementation makes D non-abstract ... */ }

struct DD : D {
    void f();
};
```

The following facts are known from this code:

```
B aB;               // B can not be instantiated as it is not complete
D aD;               // OK
DD aDD;             // OK
B* pB = new B();    // B can not be instantiated as it is not complete
B* pB = new D();    // OK
pb->f();            // Invokes D::f()
delete pb
B* pB = new DD();   // OK
pb->f();            // Invokes DD::f()
```

How much of that do we want to represent in the documentation?

Also consider that post-processing tools can be imagined that take this DITA and could generate, for example useful class interaction diagrams, their job becomes easier if these relationships are explicit rather than implicit.

There is a twist when a class declares a pure virtual function and provides an implementation of that function. For example:

```
struct B {
    virtual void f() = 0;
};

void B::f() { /* “Default” implementation ... */ }

struct D : B {
    void f();
};

void D::f() {
    /* This is actually an implementation detail... */
    return B::f();
}
```

4.2.11 Identifying exported functions?

Is this necessary? If so how? For example given in Symbian land:

```
class C {
public:
  IMPORT_C f();
};
```

Is it necessary to identify `C:f()` as exported, and what element/attribute combination could be used? The word `export` is a C++ keyword (ISO/IEC 14882:1998(E) 2.11 Keywords [lex.key]) although there is no common implementation so a reasonable choice could be to have an element that derives from apiQualifier:

```
<cxxMemberFunction id="C::f()"
  <cxxFunctionExport name="export" value="true"/>
</cxxMemberFunction>
```

Where the element `cxxFunctionExport` inherits from:

`topic/state reference/state apiRef/apiQualifier apiClassifier/apiQualifier`

REJECTED: Lack of platform independent or standards based way of expressing this.

4.2.12 Representing file and line numbers?

Do we want to represent the declaration and definition filename and line numbers?

ACCEPTED: Yes, see Section 4.4.7 “Representing file and line numbers” on page 30.

4.2.13 Does cxxSpecialMemberFunction have any value?

Well does it? If not it should be dropped.

REJECTED: Language implementation issue.

4.3 Open Design Questions

This section describes some of the alternatives that were explored in the development of this design sketch without a conclusion yet.

4.4 Variance from Prior Art

There currently exists a generic API specialisation that hints at specific language specialisations[29]. This section describes how this design deviates from that one.

4.4.1 Element Prefix

As argued in the parent design sketch the element prefix is cxx, not cpp and this is to avoid confusion with CPP, the C pre-processor.

4.4.2 Namespace is an apiClassifier not an apiPackage

The definition of a namespace can be split over several parts of one or more translation units[30] so it is less of a packaging mechanism than a means of classifying APIs.

4.4.3 cxxFunction rather than cxxMethod

'Method' is more an OO term than a language term, indeed the C++ standard does not use the term 'method' in connection with functions declared in the definition of a class, they are called *member functions*[31].

4.4.4 cxxBitField

Added, as this is assumed to be an omission.

4.4.5 cxxSpecialMemberFunction

This something of an implementation issue as these functions are automatically generated by a compiler if required. As such perhaps this should be removed from this design.

REJECTED: Removed as implementation issue.

[29] See: http://sourceforge.net/projects/dita-ot/files/Plug-in_%20apiref/apiref-0.8/apiref0.8.zip/download
And the file: apiref/doc/html/guidelineSpecialization/language-specificapireference.html

[30] ISO/IEC 14882:1998(E) Section 7.3 Namespaces [basic.namespace]

[31] ISO/IEC 14882:1998(E) Section 9.3 Member functions [class.mfct]

4.4.6 cxxClassDefinition not cxxClassDef

The current Java API reference specialisation[32] seems to conflate the meaning of declaration and definition, perhaps not surprising as there is no difference in Java. Here in C++ land we have to be more specific[33] and there seems to be no need to make abbreviations.

4.4.7 Representing file and line numbers

The API specialisation has nothing on this. DITA itself has a `<filepath>` element but there is no direct means of representing line numbers.

Note: This section describes only *how* this information is emitted not *if* they should be emitted at all. For the latter see section 4.2.12 "Representing file and line numbers?" on page 28.

There appear to be several ways to do this:

1. Introduce semantics into the `<filepath>` contents:

```
<filepath>/epoc32/include/e32std.h#172</filepath>
```

REJECTED: Wrong domain.

2. Use the `<filepath>` and specialise a `<fileLine>` element from the DITA `<state>` element:

```
<filepath>/epoc32/include/e32std.h</filepath>
<fileline name="line" value="172"/>
```

REJECTED: Mixed domains.

3. Specialise a set of elements for each API type:
For example, for a function:

```
<fileName name="path" ="/epoc32/include/e32std.h”/ >
<fileLine name="line" value="172"/>

<cxxFunctionAPIItemLocation>
    <cxxFunctionDeclarationFile name="filePath"
value="W:/sf/os/base/kernel/eka/include/e32base.h" />
    <cxxFunctionDeclarationFileLine name="lineNumber" value="100" />
    <cxxFunctionDefinitionFile name="filePath"
value="W:/sf/os/base/kernel/eka/euser/cbase/ub_buf.cpp" />
    <cxxFunctionDefinitionFileLineStart name="lineNumber" value="34" />
    <cxxFunctionDefinitionFileLineEnd name="lineNumber" value="39" />
</cxxFunctionAPIItemLocation>
```

ACCEPTED: Note DTD can default "name" as FIXED for each element.

4. Specialise a `<fileLine>` element from the DITA `<state>` element and have two entries:

```
<fileLine name="path" ="/epoc32/include/e32std.h”/ >
<fileLine name="line" value="172"/>
```

REJECTED: mixed element usage.

---

[32] See: http://sourceforge.net/projects/dita-ot/files/Plug-in_%20javaapiref/javaapiref-0.8/javaapiref0.8.zip/download
And the file: javaapiref/doc/html/javaClass/javaClassDef.html which states: "The <javaClass*Def*> element represents the Java class *declaration*." (Authors italics).

[33] ISO/IEC 14882:1998(E) Section 3.1 Declarations and definitions [basic.def]

5. ELEMENT INHERITENCE[34]

This describes the DITA element inheritance.

5.1 topic/xref

```
|   \---topic/xref
|       \---pr-d/xref
|           \---api-d/apipackage
|               +---cxxapi-d/cxxclass
|               +---cxxapi-d/cxxdefine
|               +---cxxapi-d/cxxenumeration
|               +---cxxapi-d/cxxfile
|               +---cxxapi-d/cxxfunction
|               +---cxxapi-d/cxxstruct
|               +---cxxapi-d/cxxtypedef
|               +---cxxapi-d/cxxunion
|               \---cxxapi-d/cxxvariable
```

5.2 map

5.2.1 map/map

```
+---map/map
|   \---apiMap/apiMap
|       \---cxxAPIMap/cxxAPIMap
```

5.2.2 map/topicref

```
+---map/topicref
|   \---apiMap/apiItemRef
|       +---cxxAPIMap/cxxClassRef
|       +---cxxAPIMap/cxxDefineRef
|       +---cxxAPIMap/cxxEnumerationRef
|       +---cxxAPIMap/cxxFileRef
|       +---cxxAPIMap/cxxFunctionRef
|       +---cxxAPIMap/cxxStructRef
|       +---cxxAPIMap/cxxTypedefRef
|       +---cxxAPIMap/cxxUnionRef
|       \---cxxAPIMap/cxxVariableRef
```

5.3 topic

5.3.1 topic/body

```
+---topic/body
|   \---reference/refbody
|       \---apiRef/apiDetail
|           +---apiClassifier/apiClassifierDetail
|           |   +---cxxClass/cxxClassDetail
|           |   +---cxxStruct/cxxStructDetail
|           |   \---cxxUnion/cxxUnionDetail
|           +---apiClassifier/apiDetail
|           |   +---cxxClass/cxxClassInheritsDetail
|           |   +---cxxClass/cxxClassNestedDetail
|           |   +---cxxStruct/cxxStructInheritsDetail
|           |   +---cxxStruct/cxxStructNestedDetail
|           |   \---cxxUnion/cxxUnionNestedDetail
```

[34] Many thank to Robert Shaffer for the tool to generate this directly from the DTDs.

```
|           +---apiOperation/apiOperationDetail
|           |   +---cxxDefine/cxxDefineDetail
|           |   \---cxxFunction/cxxFunctionDetail
|           \---apiValue/apiValueDetail
|               +---cxxEnumeration/cxxEnumerationDetail
|               +---cxxTypedef/cxxTypedefDetail
|               \---cxxVariable/cxxVariableDetail
```

5.3.2 topic/keyword

```
+---topic/keyword
|   \---reference/keyword
|       \---apiRef/apiItemName
|           +---apiOperation/apiItemName
|           |   +---cxxDefine/cxxDefineParameterDeclarationName
|           |   +---cxxFunction/cxxFunctionParameterDeclarationName
|           |   +---cxxFunction/cxxFunctionParameterDefinitionName
|           |   \---cxxFunction/cxxFunctionScopedName
|           \---apiValue/apiItemName
|               +---cxxEnumeration/cxxEnumerationScopedName
|               +---cxxEnumeration/cxxEnumeratorScopedName
|               +---cxxTypedef/cxxTypedefScopedName
|               \---cxxVariable/cxxVariableScopedName
```

5.3.3 topic/ph

```
+---topic/ph
|   \---reference/ph
|       +---apiRef/apiData
|       |   +---apiOperation/apiData
|       |   |   \---cxxFunction/cxxFunctionParameterDefaultValue
|       |   \---apiValue/apiData
|       |       \---cxxEnumeration/cxxEnumeratorInitialiser
|       \---apiRef/apiDefItem
|           +---apiClassifier/apiDefItem
|           |   +---cxxClass/cxxClassAPIItemLocation
|           |   +---cxxClass/cxxClassDerivation
|           |   +---cxxClass/cxxClassDerivations
|           |   +---cxxClass/cxxClassTemplateParameter
|           |   +---cxxClass/cxxClassTemplateParameters
|           |   +---cxxClass/cxxClassTemplateParameterType
|           |   +---cxxStruct/cxxStructAPIItemLocation
|           |   +---cxxStruct/cxxStructDerivation
|           |   +---cxxStruct/cxxStructDerivations
|           |   +---cxxStruct/cxxStructTemplateParameter
|           |   +---cxxStruct/cxxStructTemplateParameters
|           |   +---cxxStruct/cxxStructTemplateParameterType
|           |   +---cxxUnion/cxxUnionAPIItemLocation
|           |   +---cxxUnion/cxxUnionTemplateParameter
|           |   +---cxxUnion/cxxUnionTemplateParameters
|           |   \---cxxUnion/cxxUnionTemplateParameterType
|           +---apiOperation/apiDefItem
|           |   +---cxxDefine/cxxDefineAPIItemLocation
|           |   +---cxxDefine/cxxDefineNameLookup
|           |   +---cxxDefine/cxxDefineParameter
|           |   +---cxxDefine/cxxDefineParameters
|           |   +---cxxDefine/cxxDefinePrototype
|           |   +---cxxFunction/cxxFunctionAPIItemLocation
|           |   +---cxxFunction/cxxFunctionDeclaredType
|           |   +---cxxFunction/cxxFunctionNameLookup
|           |   +---cxxFunction/cxxFunctionParameter
|           |   +---cxxFunction/cxxFunctionParameterDeclaredType
|           |   +---cxxFunction/cxxFunctionParameters
|           |   +---cxxFunction/cxxFunctionPrototype
|           |   +---cxxFunction/cxxFunctionReturnType
|           |   +---cxxFunction/cxxFunctionTemplateParameter
|           |   +---cxxFunction/cxxFunctionTemplateParameters
|           |   \---cxxFunction/cxxFunctionTemplateParameterType
|           +---apiPackage/apiDefItem
|           |   \---cxxFile/cxxFileAPIItemLocation
|           \---apiValue/apiDefItem
|               +---cxxEnumeration/cxxEnumerationAPIItemLocation
|               +---cxxEnumeration/cxxEnumerationNameLookup
|               +---cxxEnumeration/cxxEnumerationPrototype
|               +---cxxEnumeration/cxxEnumerator
|               +---cxxEnumeration/cxxEnumeratorAPIItemLocation
|               +---cxxEnumeration/cxxEnumeratorNameLookup
|               +---cxxEnumeration/cxxEnumeratorPrototype
|               +---cxxEnumeration/cxxEnumerators
|               +---cxxTypedef/cxxTypedefAPIItemLocation
|               +---cxxTypedef/cxxTypedefDeclaredType
|               +---cxxTypedef/cxxTypedefNameLookup
|               +---cxxTypedef/cxxTypedefPrototype
|               +---cxxVariable/cxxVariableAPIItemLocation
|               +---cxxVariable/cxxVariableDeclaredType
|               +---cxxVariable/cxxVariableNameLookup
|               \---cxxVariable/cxxVariablePrototype
```

5.3.4 topic/section

```
+---topic/section
|   \---reference/section
|       \---apiRef/apiDef
|           +---apiClassifier/apiClassifierDef
|           |   +---cxxClass/cxxClassDefinition
|           |   +---cxxStruct/cxxStructDefinition
|           |   \---cxxUnion/cxxUnionDefinition
|           +---apiOperation/apiOperationDef
|           |   +---cxxDefine/cxxDefineDefinition
|           |   \---cxxFunction/cxxFunctionDefinition
|           \---apiValue/apiValueDef
|               +---cxxEnumeration/cxxEnumerationDefinition
|               +---cxxTypedef/cxxTypedefDefinition
|               \---cxxVariable/cxxVariableDefinition
```

5.3.5 topic/state

```
+---topic/state
|   \---reference/state
|       \---apiRef/apiQualifier
|           +---apiClassifier/apiQualifier
|           |   +---cxxClass/cxxClassAbstract
|           |   +---cxxClass/cxxClassAccessSpecifier
|           |   +---cxxClass/cxxClassDeclarationFile
|           |   +---cxxClass/cxxClassDeclarationFileLine
|           |   +---cxxClass/cxxClassDefinitionFile
|           |   +---cxxClass/cxxClassDefinitionFileLineEnd
|           |   +---cxxClass/cxxClassDefinitionFileLineStart
|           |   +---cxxClass/cxxClassDerivationAccessSpecifier
|           |   +---cxxClass/cxxClassDerivationVirtual
|           |   +---cxxStruct/cxxStructAbstract
|           |   +---cxxStruct/cxxStructAccessSpecifier
|           |   +---cxxStruct/cxxStructDeclarationFile
|           |   +---cxxStruct/cxxStructDeclarationFileLine
|           |   +---cxxStruct/cxxStructDefinitionFile
|           |   +---cxxStruct/cxxStructDefinitionFileLineEnd
|           |   +---cxxStruct/cxxStructDefinitionFileLineStart
|           |   +---cxxStruct/cxxStructDerivationAccessSpecifier
|           |   +---cxxStruct/cxxStructDerivationVirtual
|           |   +---cxxUnion/cxxUnionAbstract
|           |   +---cxxUnion/cxxUnionAccessSpecifier
|           |   +---cxxUnion/cxxUnionDeclarationFile
|           |   +---cxxUnion/cxxUnionDeclarationFileLine
|           |   +---cxxUnion/cxxUnionDefinitionFile
|           |   +---cxxUnion/cxxUnionDefinitionFileLineEnd
|           |   \---cxxUnion/cxxUnionDefinitionFileLineStart
|           +---apiOperation/apiQualifier
|           |   +---cxxDefine/cxxDefineAccessSpecifier
|           |   +---cxxDefine/cxxDefineDeclarationFile
|           |   +---cxxDefine/cxxDefineDeclarationFileLine
|           |   +---cxxFunction/cxxFunctionAccessSpecifier
|           |   +---cxxFunction/cxxFunctionConst
|           |   +---cxxFunction/cxxFunctionConstructor
|           |   +---cxxFunction/cxxFunctionDeclarationFile
|           |   +---cxxFunction/cxxFunctionDeclarationFileLine
|           |   +---cxxFunction/cxxFunctionDefinitionFile
|           |   +---cxxFunction/cxxFunctionDefinitionFileLineEnd
|           |   +---cxxFunction/cxxFunctionDefinitionFileLineStart
|           |   +---cxxFunction/cxxFunctionDestructor
|           |   +---cxxFunction/cxxFunctionExplicit
|           |   +---cxxFunction/cxxFunctionInline
|           |   +---cxxFunction/cxxFunctionPureVirtual
|           |   +---cxxFunction/cxxFunctionStorageClassSpecifierExtern
|           |   +---cxxFunction/cxxFunctionStorageClassSpecifierMutable
|           |   +---cxxFunction/cxxFunctionStorageClassSpecifierStatic
|           |   +---cxxFunction/cxxFunctionVirtual
|           |   \---cxxFunction/cxxFunctionVolatile
|           +---apiPackage/apiQualifier
|           |   \---cxxFile/cxxFileDeclarationFile
|           \---apiValue/apiQualifier
|               +---cxxEnumeration/cxxEnumerationAccessSpecifier
|               +---cxxEnumeration/cxxEnumerationDeclarationFile
|               +---cxxEnumeration/cxxEnumerationDeclarationFileLine
|               +---cxxEnumeration/cxxEnumerationDefinitionFile
|               +---cxxEnumeration/cxxEnumerationDefinitionFileLineEnd
|               +---cxxEnumeration/cxxEnumerationDefinitionFileLineStart
|               +---cxxEnumeration/cxxEnumeratorDeclarationFile
|               +---cxxEnumeration/cxxEnumeratorDeclarationFileLine
|               +---cxxTypedef/cxxTypedefAccessSpecifier
|               +---cxxTypedef/cxxTypedefDeclarationFile
|               +---cxxTypedef/cxxTypedefDeclarationFileLine
|               +---cxxVariable/cxxVariableAccessSpecifier
|               +---cxxVariable/cxxVariableConst
|               +---cxxVariable/cxxVariableDeclarationFile
|               +---cxxVariable/cxxVariableDeclarationFileLine
|               +---cxxVariable/cxxVariableStorageClassSpecifierExtern
|               +---cxxVariable/cxxVariableStorageClassSpecifierMutable
|               +---cxxVariable/cxxVariableStorageClassSpecifierStatic
|               \---cxxVariable/cxxVariableVolatile
```

5.3.6 topic/topic

```
+---topic/topic
|   \---reference/reference
|       \---apiRef/apiRef
|           +---apiClassifier/apiClassifier
|           |   +---cxxClass/cxxClass
|           |   +---cxxClass/cxxClassInherits
|           |   +---cxxClass/cxxClassNested
|           |   +---cxxStruct/cxxStruct
|           |   +---cxxStruct/cxxStructInherits
|           |   +---cxxStruct/cxxStructNested
|           |   +---cxxUnion/cxxUnion
|           |   \---cxxUnion/cxxUnionNested
|           +---apiOperation/apiOperation
|           |   +---cxxDefine/cxxDefine
|            |   \---cxxFunction/cxxFunction
|            +---apiPackage/apiPackage
|            |   \---cxxFile/cxxFile
|            \---apiValue/apiValue
|                +---cxxEnumeration/cxxEnumeration
|                +---cxxTypedef/cxxTypedef
|                \---cxxVariable/cxxVariable
```

5.3.7 topic/xref

```
\---topic/xref
    \---reference/xref
        \---apiRef/apiRelation
            +---apiClassifier/apiBaseClassifier
            |   +---cxxClass/cxxClassBaseClass
            |   +---cxxClass/cxxClassBaseStruct
            |   +---cxxClass/cxxClassBaseUnion
            |   +---cxxStruct/cxxStructBaseClass
            |   +---cxxStruct/cxxStructBaseStruct
            |   \---cxxStruct/cxxStructBaseUnion
            +---apiClassifier/apiRelation
            |   +---cxxClass/cxxClassEnumerationInherited
            |   +---cxxClass/cxxClassEnumeratorInherited
            |   +---cxxClass/cxxClassFunctionInherited
            |   +---cxxClass/cxxClassNestedClass
            |   +---cxxClass/cxxClassNestedStruct
            |   +---cxxClass/cxxClassNestedUnion
            |   +---cxxClass/cxxClassVariableInherited
            |   +---cxxStruct/cxxStructEnumerationInherited
            |   +---cxxStruct/cxxStructEnumeratorInherited
            |   +---cxxStruct/cxxStructFunctionInherited
            |   +---cxxStruct/cxxStructNestedClass
            |   +---cxxStruct/cxxStructNestedStruct
            |   +---cxxStruct/cxxStructNestedUnion
            |   +---cxxStruct/cxxStructVariableInherited
            |   +---cxxUnion/cxxUnionNestedClass
            |   +---cxxUnion/cxxUnionNestedStruct
            |   \---cxxUnion/cxxUnionNestedUnion
            +---apiOperation/apiRelation
            |   +---cxxDefine/cxxDefineReimplemented
            |   \---cxxFunction/cxxFunctionReimplemented
            \---apiValue/apiRelation
                +---cxxEnumeration/cxxEnumerationReimplemented
                +---cxxTypedef/cxxTypedefReimplemented
                \---cxxVariable/cxxVariableReimplemented
```

6. FURTHER INFORMATION

6.1 People

Role	Person / People
Reviewers	System Documentation, S60, SDA.
Contributors	Jonathan Harrington, Valentine Ogier-Galland, Robert Shaffer, Abigail Sidford, Paul Ross, Eleanor Weavers.
Distribution	Nokia.

6.2 References

No.	Document Reference	Version	Description
[R1]	ISO/IEC 9899:1999 (E)	First edition 1990	Programming languages C
[R2]	ISO/IEC 14882:1998(E)	First edition 1998-09-01	Programming languages C++
[R3]	ISO/IEC JTC 1/SC22/WG21 N2800	N2914=09-0104 Date: 2009-06-22	Programming Languages C++ http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2914.pdf
[R4]	RedFist Design Sketch [RM-RIM: 417-53493]		Design Sketch of the Doxygen based in-source comment to DITA XML tool.
[R5]	http://en.wikipedia.org/wiki/Name_mangling	Current	Introduction to name mangling.
[R6]	http://www.codesourcery.com/public/cxx-abi/abi.html#mangling	$Revision: 1.86 $	Itanium C++ Name mangling sheme.
[R7]	http://en.wikipedia.org/wiki/Darwin_Information_Typing_Architecture	Current	The DITA page on Wikipedia.
[R8]	//EPOC/DV3/team/2005/sysdoc/tools/buildrefdoc/documentation/rel/SGL.PR0098.xxx_v0.3.XML_NAME_encoding.doc	0.2	XML Name encoding.
[R9]	ISO/IEC 14882:2003(E)	Second edition 2003-10-15	Available from: http://openassist.googlecode.com/files/C%2B%2B%20Standard%20-%20ANSI%20ISO%20IEC%2014882%202003.pdf
[R10]	Existing DITA specialisations for APIs	Generic API specialisation version 0.8 dated Thu Feb 23 2006 06:24 Java API specialisation version 0.8 dated Thu Feb 23 2006 06:26	API specialisation from: http://sourceforge.net/projects/dita-ot/files/Plug-in_%20apiref/apiref-0.8/apiref0.8.zip/download Java specialisation from: http://sourceforge.net/projects/dita-ot/files/Plug-in_%20javaapiref/javaapiref-0.8/javaapiref0.8.zip/download
[R11]	Orb Architecture [RM-RIM: 417-53493]		Architecture of the Doxygen based in-source comment to DITA XML tool.

6.3 Open Issues

6.4 Glossary

Term	Definition
CMS	Content Management System.
SDL	Symbian Developer Library.
SF	Symbian Foundation.
SFDL	Symbian Foundation Developer Library.