swiftML design rules

Technical Specification

Table of contents

1. Introduction

Last year, the SWIFT Board approved the adoption of XML as the preferred syntax for SWIFTStandards. XML is a technical standard defined by W3C (the World Wode Web Consortium) and leaves a lot of freedom for the exact way it is used in a particular application. Therefore, merely stating that we are going to use XML is not sufficient, we must also explain HOW we will use XML.
The use of XML by SWIFT is part of the overall approach for the development of SWIFTStandards. This development focuses on the correct definition of a business standard using modelling techniques[1]. The resulting business standard is captured in UML (Unified Modelling Language[2]) and is stored in an electronic repository, the "SWIFTStandards Repository"[3]. Business messages are defined in UML class diagrams and XML is then used as a physical representation (i.e. the syntax) of the defined business messages. A set of **XML design rules**, called **swiftML**, define in a very detailed and strict way how this physical XML representation is derived from the business message in the UML class diagram.
This document explains these XML design rules.

2. Mapping rules from UML to swiftML

2.1 General mapping rules

Mapping rules from UML to swiftML are governed by the following design choices:

- swiftML representation to be as structured as possible:
  - Business information is expressed as XML elements/values;
  - Metadata information is expressed as XML attributes. XML attributes are not to be conveyed 'on the wire' in the XML instance, unless required to remove ambiguity.
- Even though we have XML-schemas in mind, the design rules currently focus on the generation of traditional DTDs.
- swiftML element tag names should be readable.
- Names in UML should be reused in swiftML as much as possible.
- swiftML elements are derived from the UML representation of a business message. They can only be derived from UML-classes, UML-roles or UML-attributes.
- Each swiftML element must be traceable to the corresponding UML model element.

2.2 swiftML elements

Any swiftML element shall have the following structure:

<swiftMLTag elementID="xxx" [roleID="yyy"] type="zzz" version="aaa" >

2.2.1 swiftMLTag

swiftMLTag is assigned according to following rules:
For a swiftML element derived from a class:

- the name of the class

For a swiftML element derived from a role:

- the name of the role

---

[1] This approach is called SWIFTStandards Modelling. You can find more information on this approach in the SWIFTStandards White Paper on www.swift.com.

[2] You can find more information about UML on the Object Management Group website at: http://www.omg.org/uml

[3] You can find more information on this repository in the SWIFTStandards White Paper on www.swift.com.

For a swiftML element derived from an attribute:
- the name of the attribute

2.2.2 elementID

elementID links the swiftML element to its corresponding business meaning in the SWIFTStandards Repository.
This approach facilitates implementation and testing since it offers an easier way to implement / check / validate the same classes cross-message/scenario.
elementID is assigned according to following rules:
For a swiftML element derived from a class:
- the swiftID of the class if this class does not have a refinement Traceability link;
- the swiftID of the element that it is found by following the complete chain of refinement Traceability links starting from the class, if this class has a refinement Traceability link.
  Refinement Traceability link, for a class:
- is modeled as a dependency relationship from this class toward the class which is refined;
- with stereotype[4] set as <<refine>>.

For a swiftML element derived from a role:
- the swiftID of the class which is the end of the aggregation if this class does not have a refinement Traceability link;
- the swiftID of the element that it is found by following the complete chain of refinement Traceability links starting from the class which is the end of the aggregation if this class has a refinement Traceability link.
  Remark:
- If the class at the end of the aggregation is specialised the DTD will specify that the elementID can be the swiftID of the base class or the swiftID of one of the specialised classes (taking into account the refinement Traceability links for all considered base and specialised classes).

For a swiftML element derived from an attribute:
- the swiftID of the attribute if this attribute does not have a refinement Traceability link;
- the swiftID of the element that it is found by following the complete chain of refinement Traceability links starting from the attribute if this attribute has a refinement Traceability link.
  Refinement Traceability link, for an attribute:
- is modeled as a dependency relationship from this attribute toward the attribute which is refined;
- with stereotype set as <<refine>>.

ElementID is:
- S.W.I.F.T. owned, maintained and issued
- unique
- opaque
- fixed length (8 characters), alphanumeric (lower case and upper case)
- machine generated, but machine independent

Since this code is S.W.I.F.T. proprietary, it clearly distinguishes a swiftML element from any other XML repository element. Hence it is logical that a third party cannot assign a elementID to a proprietary created XML element.
As two roles with the same class will have the same elementID, a **roleID** is introduced.

2.2.3 roleID

roleID links the swiftML element to the UML-role of the corresponding business element in the business message defined in the SWIFTStandards Repository.
roleID is assigned according to following rules:
For a swiftML element derived from a class:
  This swiftML element has no roleID.
For a swiftML element derived from a role:
- the swiftID of the role if this role does not have a refinement Traceability link;

[4] Stereotypes are one of the extension mechanisms used to extend the semantics of the metamodel.

- the swiftID of the refined role if this role has a refinement Traceability link.
  Refinement Traceability link, for a role:
- is modeled as a dependency relationship from this aggregation toward the role which is refined;
- with stereotype set as <<refine>>.

For a swiftML element derived from an attribute:
  This swiftML element has no roleID.

RoleID is
- S.W.I.F.T. owned, maintained and issued
- unique
- opaque
- fixed length (8 characters), alphanumeric (lower case and upper case)
- machine generated, but machine independent

2.2.4 type

type links the swiftML element to the UML-type of the corresponding business element in the business message defined in the SWIFTStandards Repository.

In the DTD

In the DTD, type is assigned according to following rules:
For a swiftML element derived from a class:
- the swiftID of the class

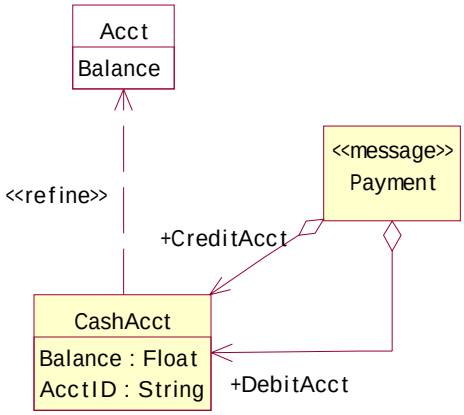
For a swiftML element derived from a role:
- the swiftID of the class which is the end of the aggregation

For a swiftML element derived from an attribute:
- the name of the type of the attribute if this type is a primitive type
- the swiftID of the class by which the type of the attribute is defined if this type is no primitive type

In the corresponding instance

In the swiftML instance, in case of ambiguity, the selected “type” MUST also be specified.
There are occasions whereby an elementID together with its roleID still do not offer in swiftML an unambiguous interpretation in the corresponding XML instance. This is due to the fact that inheritance can create ambiguity in the swiftML instance since not all elements preceding or following in a hierarchy of elements are mentioned in swiftML. In other words, the type swiftML attribute is required only to remove ambiguity on a swiftML instance, in those cases whereby there are multiple usages of a same swiftML element defined in the corresponding DTD.
This can for instance be the case when in UML a class has several specialisations, or an attribute has several primitive datatypes, or with XOR constructs (see further). In those cases, it is necessary to explicitly indicate which usage of that swiftML element is meant. This is achieved by introducing this ‘type’ attribute that contains the elementID of the logical class to which this swiftML element applies.

Example of the usage of the ID:




```
class  Acct          ID=ABCD0001
role   Balance       ID=ABCD0002

class  Payment       ID=ABCD0003
class  CashAcct      ID=ABCD0004
role   CreditAcct    ID=ABCD0005
role   DebitAcct     ID=ABCD0006
role   Balance       ID=ABCD0007
role   AcctID        ID=ABCD0008
```

```
<Payment elementID="ABCD0003" type="ABCD0003">
 <CreditAcct elementID="ABCD0001" roleID="ABCD0005" type="ABCD0004">
  <Balance elementID="ABCD0002" type="float">1000</Balance>
  <AcctID elementID="ABCD0008" type="string">124-56789-1</AcctID>
 </CreditAcct>
 <DebitAcct elementID="ABCD0001" roleID="ABCD0006" type="ABCD0004">
  <Balance elementID="ABCD0002" type="float">1550</Balance>
  <AcctID elementID="ABCD0008" type="string">125-56789-1</AcctID>
 </DebitAcct>
</Payment>
```

DTD:

```
<!ELEMENT NetProceed (Quantity, Allocated, ProcessedDate>
   <!ATTLIST NetProceed elementID CDATA #FIXED “1”
          type CDATA #FIXED “ABCD0008”
          version CDATA #FIXED “1.0”>

<!ELEMENT Quantity (#PCDATA)>
<!ATTLIST Quantity elementID CDATA #FIXED “ABCD0002”
          type CDATA #FIXED “integer”
          version CDATA #FIXED “1.0”>

<!ELEMENT Allocated (#PCDATA)>
<!ATTLIST Allocated elementID CDATA #FIXED “ABCD0003”
          type CDATA #FIXED “boolean”
          version CDATA #FIXED “1.0”>

<!ELEMENT ProcessedDate (#PCDATA)>
<!ATTLIST ProcessedDate elementID CDATA #FIXED “ABCD0004”
          type CDATA #FIXED “date”
          version CDATA #FIXED “1.0”>
```

Although it does not appear on the drawing, there is a “refine” traceability link between the attribute Balance of class CashAcct and the attribute Balance of class Acct.
For more information on traceability and how it is applied with the elementID, roleID and type, see traceability .

2.2.5 version

version contains the public version number of the swiftML tag. See versioning
version is assigned according to following rules:
For a swiftML element derived from a class:

- the public version of the element that is used to define the elementID of the class

For a swiftML element derived from a role:

- the public version of the element that is used to define the elementID of the role

For a swiftML element derived from an attribute:

- the public version of the element that is used to define the elementID of the attribute

2.3 Specific mapping rules

All model elements, defined accordingly to the SWIFTStandards methodology, are based on following UML structures. Hence, by defining the conversion rules from those structures into swiftML we can convert any UML model into its corresponding swiftML DTD and instance.
Roles and attributes in UML are considered the same in swiftML; they map into XML elements.
Classes do not map into XML elements, except for when they become the root element of the DTD.

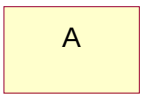
2.3.1 Primitive data types

- swiftML primitive data types are encoded as defined by W3C, defined at http://www.w3.org/TR/xmlschema-2/#dt-encoding. If roles are polymorphic because they have multiple primitive datatypes, use the "type" attribute for those XML elements that need to specify the primitive datatype. We will then use the same codes as defined for XML-schema primitive datatypes. This means that the primitive datatypes do not need to be defined in the swiftStandards Repository (SSR).

UML Name	XML Name	Description
`String`	`string`	Set of finite sequences of UTF-8 characters
`Boolean`	`boolean`	Has the value space of boolean constants "True" or "False"
`Integer`	`integer`	Corresponds to 32 bits integer type
`BigDecimal`	`decimal`	Arbitrary precision decimal numbers
`Float`	`float`	Corresponds to IEEE single-precision 32 bits floating point type
`Double`	`double`	Corresponds to IEEE double-precision 64 bits floating point type
`Long`	`long`	Corresponds to 64 bits integer type
`Date`	`date`	Corresponds to a date. See ISO 8601.

2.3.2 Class

UML	XML instance
Class name with a role	Role becomes an element
Class name without a role	When it is the root element, then class name becomes the element name. When it is NOT the root element, then the class name becomes the element name AND "_**role**" is added.



Instance:

```
<A>
</A>
```

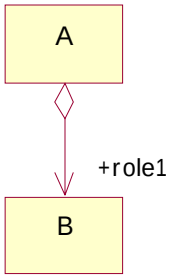
DTD:

```
<!ELEMENT A>
<!ATTLIST A elementID CDATA #FIXED "ID of class A"
            type CDATA #FIXED "ID of class A"
            version CDATA #FIXED "1.0"
>
```

2.3.3 Simple composition

- A parent-child relationship between two classes is expressed by a role;
- The parent-class maps to a swiftML element with its name as the tag;
- The child-class maps to a swiftML element with its role as the tag.

UML	swiftML instance
Parent class	swiftML element with class name as tag
Child class	swiftML element with role name as tag. This element is contained within the parent element



Instance:

```
<A>
   <role1>
   </role1>
</A>
```

DTD:

```
<!ELEMENT A (role1)>
<!ATTLIST A elementID CDATA #FIXED "ID of class A"
            type CDATA #FIXED "ID of class A"
            version CDATA #FIXED "1.0"
>
<!ELEMENT role1 (#PCDATA)>
<!ATTLIST role1 elementID CDATA #FIXED "ID of class B"
            roleID CDATA #FIXED "ID of role1"
            type CDATA #FIXED "ID of class B"
            version CDATA #FIXED "1.0"
>
```

2.3.4 Class attributes

- A class can also contain attributes;
- A class attribute is described using a name and a type;
- The derived swiftML instance only considers UML roles and UML attributes equivalent as far as the swiftML tags are concerned. The XML attributes are different;
- The first swiftML child elements within its parents are the attributes, followed by the roles.

UML	**swiftML instance**
Class containing attributes	Attributes become nested swiftML elements.


```
<B>
  <att1>data</att1>
  <att2>data</att2>
</B>
```

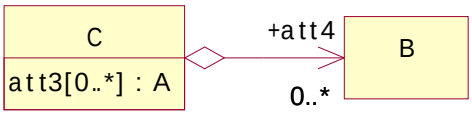
DTD:

```
<!ELEMENT B (att1,att2)>
<!ATTLIST B elementID CDATA #FIXED "ID of class B"
            type CDATA #FIXED “ID of class B”
            version CDATA #FIXED "1.0"
>
<!ELEMENT att1 (#PCDATA)>
<!ATTLIST att1 elementID CDATA #FIXED "ID of att1"
               type CDATA #FIXED “ID of class C”
               version CDATA #FIXED "1.0"
>
<!ELEMENT att2 (#PCDATA)>
<!ATTLIST att2 elementID CDATA #FIXED "ID of class A"
               roleID CDATA #FIXED "ID of att2"
               type CDATA #FIXED “ID of class A”
               version CDATA #FIXED "1.0"
>
```

2.3.5 Composition of vectorial attributes (Collections)

- The cardinality expresses the number of occurrences of elements. In most cases the cardinality is 1, in which case it is omitted; else it is represented as a **range** e.g **`0..*`** .
- Use a range-cardinality to express a collection of elements, which can be represented either as a collection of attributes or roles. In the example below, C contains a collection of As expressed as attributes (att3) and a collection of Bs expressed as roles (att4).
- A container element is added. It wraps each collection. Its name is derived from the name of the role. The rule is to append an underscore character to the name of the role (e.g. **att3**_ and **att4**_). The container does not have any existence in the data dictionary (it will not have any elementID associated with it), and its name is derived from the name of the role or the attribute.
- To a certain extent XML DTDs can validate the cardinality.

Cardinality	Description	DTD representation
1	Exactly one	A
0..1	Optional	A?
0..*	Any number of occurrences	A*
1..*	At least one	A+
1..4	From 1 to 4	Not directly supported with DTD.




```
<C>
   <att3_>
      <att3>data</att3>
      <att3>data</att3>
   </att3_>
   <att4_>
      <att4>data</att4>
      <att4>data</att4>
   </att4_>
</C>
```

DTD:

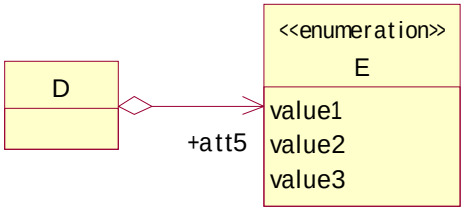
```
<!ELEMENT C (att3_,att4_)>
<!ATTLIST C elementID CDATA #FIXED "ID of class C"
            type CDATA #FIXED "ID of class C"
            version CDATA #FIXED "1.0"
>
<!ELEMENT att3_ (att3*)>
<!ELEMENT att4_ (att4*)>
<!ELEMENT att3 (#PCDATA)>
<!ATTLIST att3 elementID CDATA #FIXED "ID of att3"
               type CDATA #FIXED "ID of A"
               version CDATA #FIXED "1.0"
>
<!ELEMENT att4 (#PCDATA)>
<!ATTLIST att4 elementID CDATA #FIXED "ID of class B"
               roleID CDATA #FIXED "ID of att4"
               type CDATA #FIXED "ID of class B"
```

```
                version CDATA #FIXED “1.0”
>
```

2.3.6 Enumerations

- An enumerated value is constrained within a list of possible values.
- Enumerations are represented in the model as a type (e.g. E) with all the possible values listed as attributes (e.g. value1, value2, value3).
- The swiftML representation of an enumerated value contains the chosen value.
- The DTD cannot validate whether one of the enumerated values in the model has actually been used in the instance since DTD do not allow for validation of the actual data[5].

UML	swiftML instance
Class contains an enumeration of possible values	swiftML element contains the chosen value




```
<D>
  <att5>
     value2
  </att5>
</D>
```

DTD:

```
<!ELEMENT D (att5)>
<!ATTLIST D elementID CDATA #FIXED "ID of class D"
            type CDATA #FIXED "ID of class D"
            version CDATA #FIXED "1.0"
>
<!ELEMENT att5 (#PCDATA)>
<!ATTLIST att5 elementID CDATA #FIXED "ID of class E"
            type CDATA #FIXED "ID of class E"
            roleID CDATA #FIXED "ID of att5"
            version CDATA #FIXED "1.0">
```

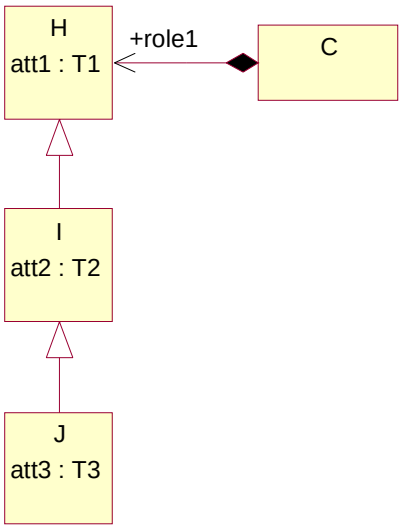
[5] An alternative could have been to represent enumerations as XML attributes, which DTD can validate (i.e.ATTLIST); however this is against one of the design principles, which is to separate business data from meta-data.

2.3.7 Inheritance

It is possible to re-use business elements by specializing existing elements. This process has impacts on element order and on generated DTDs. Both issues are described below.

Element order

- In the example below the business element H contains an attribute att1. The business element I, which re-uses H, contains att2 and att1; the latter attribute is inherited from H. The business element J, which re-uses I, contains att3, att2 and att1; the last two attributes being inherited from I respectively H.




```
<C>
   <role1 elementID='ID_of_H' type='ID_of_H'>
      <att1>data</att1>
   </role1>
</C>
```

or

```
<C>
   <role1 elementID='ID_of_I' type='ID_of_I'>
      <att1>data</att1>
      <att2>data</att2>
   </role1>
</C>
```

or

```
<C>
   <role1 elementID='ID_of_J' type='ID_of_J'>
      <att1>data</att1>
      <att2>data</att2>
      <att3>data</att3>
   </role1>
</C>
```

DTD:

```
<!ELEMENT C (role1)>
```

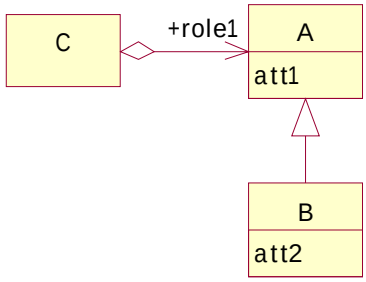
```
<!ATTLIST C elementID CDATA #FIXED "ID_of_C"
            type CDATA #FIXED "ID_of_C"
            version CDATA #FIXED "1.0"
>
<!ELEMENT role1 (att1,(att2,att3?)?)>
<!ATTLIST role1 elementID (ID_of_I|ID_of_J|ID_of_H) #REQUIRED
          roleID CDATA #FIXED "ID_of_role1"
          type (ID_of_I|ID_of_J|ID_of_H) #REQUIRED
          version CDATA #FIXED "1.0"
>
<!ELEMENT att3 (#PCDATA)>
<!ATTLIST att3 elementID CDATA #FIXED "ID_of_att3"
            type CDATA #FIXED "ID_of_T3"
            version CDATA #FIXED "1.0"
>
<!ELEMENT att2 (#PCDATA)>
<!ATTLIST att2 elementID CDATA #FIXED "ID_of_att2"
            type CDATA #FIXED "ID_of_T2"
            version CDATA #FIXED "1.0"
>
<!ELEMENT att1 (#PCDATA)>
<!ATTLIST att1 elementID CDATA #FIXED "ID_of_att1"
            type CDATA #FIXED "ID_of_T1"
            version CDATA #FIXED "1.0"
>
```

Notes:

- Inherited attributes appear first;
- Inheritance is cumulative: always add attributes, never remove them;
- It is an error to redefine an attribute that already exists in a base class.

Virtual containment

- Suppose an element A has been specialized as an element B;
- This means that a container C, containing A, can also contain B, as B "is-a" A;




```
<C>
   <role1 elementID="ID_of_B" type="ID_of_B">
      <att1>data</att1>
      <att2>data</att2>
   </role1>
</C>
```

DTD:

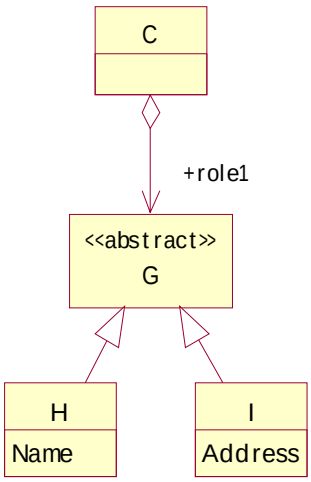
```
<!ELEMENT C (role1)>
<!ATTLIST C elementID CDATA #FIXED "ID_of_C"
            type CDATA #FIXED "ID of class C"
            version CDATA #FIXED "1.0"
>
```

```
<!ELEMENT role1 (att1,att2?)>
<!ATTLIST role1 elementID (ID_of_A|ID_of_B) #REQUIRED       roleID CDATA #FIXED
“ID_of_role1”
            type (ID_of_A|ID_of_B) #REQUIRED
            version CDATA #FIXED "1.0"
>
<!ELEMENT att1 (#PCDATA)>
<!ATTLIST att1 elementID CDATA #FIXED "ID_of_A"
            type CDATA #FIXED “ID_of_the_type_of_att1”
            version CDATA #FIXED "1.0"
>
<!ELEMENT att2 (#PCDATA)>
<!ATTLIST att2 elementID CDATA #FIXED "ID_of_B"
            type CDATA #FIXED “ID_of_the_type_of_att2”
            version CDATA #FIXED "1.0"
>
```

2.3.8 Enumerated types

Basic pattern

- In most cases a role is played by an element of a given type. However there are business needs to allow for a role to be played by more than one type.
- In the example below, two different types can play role1: either Name or Address.
- In the swiftML representation, an swiftML attribute is introduced to express the actual type.
- The DTD enumerates all the possible values this type can take.




```
<C>
  <role1 elementID="ID_of_H" type="ID_of_H">
    <Name>data</Name>
  </role1>
</C>
```

or

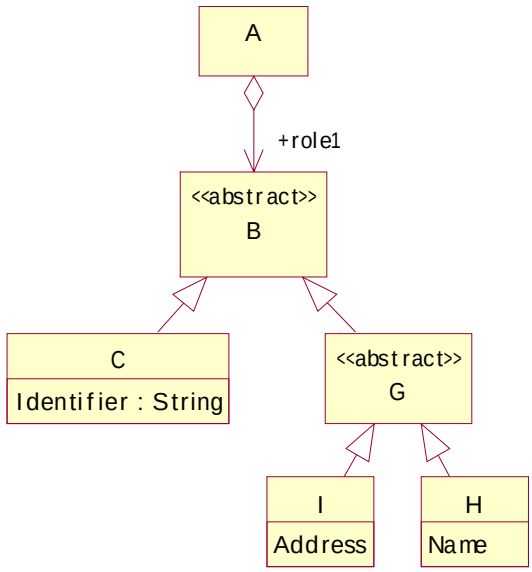
```
<C>
  <role1 elementID="ID_of_I" type="ID_of_I">
    <Address>data</Address>
  </role1>
</C>
```

DTD:

```
<!ELEMENT C (role1)>
<!ATTLIST C elementID CDATA #FIXED "ID of C"
            type CDATA #FIXED "ID of C"
            version CDATA #FIXED "1.0"
>
<!ELEMENT role1 (Name|Address)>
<!ATTLIST role1 elementID (ID_of_I|ID_of_H) #REQUIRED
          roleID CDATA #FIXED "ID of role1"
          type (ID_of_I|ID_of_H) #REQUIRED
          version CDATA #FIXED "1.0"
>
<!ELEMENT Name (#PCDATA)>
<!ATTLIST Name elementID CDATA #FIXED "ID of Name"
```

```
            type CDATA #FIXED “ID of type of Name”
            version CDATA #FIXED "1.0"
<!ELEMENT Address (#PCDATA)>
<!ATTLIST Address elementID CDATA #FIXED "ID of Address"
            type CDATA #FIXED “ID of the type of Address”
            version CDATA #FIXED "1.0"
```

Re-use pattern




```
<A>
  <role1 elementID="ID_of_C" type="ID_of_C">
    <Identifier>data</Identifier>
  </role1>
</A>
```

or

```
<A>
  <role1 elementID="ID_of_I" type="ID_of_I">
    <Address>data</Address>
  </role1>
</A>
```

or

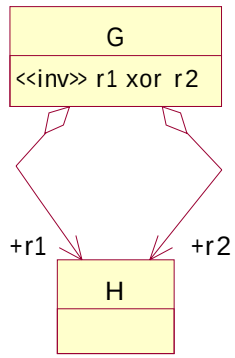
```
<A>
  <role1 elementID="ID_of_H" type="ID_of_H">
    <Name>data</Name>
  </role1>
</A>
```

DTD

```
<!ELEMENT A (role1)>
<!ATTLIST A elementID CDATA #FIXED "ID of A"
            type CDATA #FIXED “ID of A”
            version CDATA #FIXED "1.0"
>
<!ELEMENT role1 (Identifier|Address|Name)>
<!ATTLIST role1 elementID (ID_of_C|ID_of_I|ID_of_H) #REQUIRED
          roleID CDATA #FIXED "ID of role1"
```

```
          type (ID_of_C|ID_of_I|ID_of_H) #REQUIRED
          version CDATA #FIXED "1.0"
>
<!ELEMENT Identifier (#PCDATA)>
<!ATTLIST Identifier elementID CDATA #FIXED "ID of Identifier"
            type CDATA #FIXED “String”
            version CDATA #FIXED "1.0"
>
<!ELEMENT Name (#PCDATA)>
<!ATTLIST Name elementID CDATA #FIXED "ID of Name"
            type CDATA #FIXED “ID of the type of Name”
            version CDATA #FIXED "1.0"
>
<!ELEMENT Address (#PCDATA)>
<!ATTLIST Address elementID CDATA #FIXED "ID of Address"
            type CDATA #FIXED “ID of the type of Address”
             version CDATA #FIXED "1.0"
>
```

2.3.9 Enumerated roles




```
<G>
  <r1>data</r1>
</G>
```

or

```
<G>
  <r2>data</r2>
</G>
```

DTD:

```
<!ELEMENT G (r1|r2)>
<!ATTLIST G elementID CDATA #FIXED “ID of class G”
            type CDATA #FIXED “ID of class G”
            version CDATA #FIXED “1.0”
>
<!ELEMENT r1 (#PCDATA)>
<!ATTLIST r1 elementID CDATA #FIXED “ID of class H”
             roleID CDATA #FIXED “ID of r1”
             type CDATA #FIXED “ID of class H”
             version CDATA #FIXED “1.0”
```

```
>
<!ELEMENT r2 (#PCDATA)>
<!ATTLIST r2 elementID CDATA #FIXED “ID of class H”
             roleID CDATA #FIXED “ID of r2”
             type CDATA #FIXED “ID of class H”
             version CDATA #FIXED “1.0”
>
```

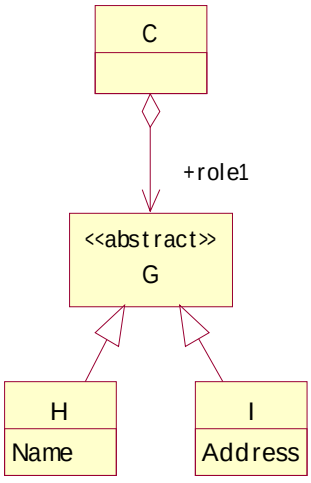
3. DTD and Schema design rules

3.1 Common design rules and usage

- Should only be used to validate the message (though this validation is limited if we compare with pure software validation)
- Should not be used to document the message
- Should not replace the UML model.
- Should provide meta-information that enriches each instance in memory in order to be used by the customer/programmer.

3.2 DTD design rules

- DTD do not make a distinction between Type and Role => Role names must be unique in a message.
- Enumerations as attributes cannot be used.
- In case of a name clash in a DTD (for instance 2 UML roles have the same name but for different classes), the only way to resolve this with DTD is to decrease the level of validation in the DTD. In other words, the DTD in such a case will not describe the properties but instead use the “ANY” code word with ELEMENT in the DTD. Rationale being that a DTD constraint should be solved at DTD level, and thus decoupled from the model.
- Since swiftML attributes hold meta-information, the DTD describes all the possible attributes using an attribute list, noted **`ATTLIST`**.
- **`ENTITIES`** could be used to put some additional information about the structure of the message. In some of the patterns, part of the structure can be lost when converting into swiftML. Suppose following pattern:



The above pattern would normally result in the generation of a DTD that does not contain ENTITIES.

Without entities (see also pattern “basic pattern of enumerated types”):

```
<!ELEMENT C (role1)>
<!ATTLIST C elementID CDATA #FIXED "ID of class C"
            type CDATA #FIXED “ID of class C”
            version CDATA #FIXED "1.0"
>
<!ELEMENT role1 (Name|Address)>
<!ATTLIST role1 elementID CDATA ID_of_I|ID_of_H) #REQUIRED
```

```
          roleID CDATA #FIXED “ID of role1”
          type (ID_of_I|ID_of_H) #REQUIRED
          version CDATA #FIXED "1.0"
>
<!ELEMENT Name (#PCDATA)>
<!ATTLIST Name elementID CDATA #FIXED "ID of Name"
            type CDATA #FIXED “ID of the type of Name”
            version CDATA #FIXED "1.0"
<!ELEMENT Address (#PCDATA)>
<!ATTLIST Address elementID CDATA #FIXED "ID of Address"
            type CDATA #FIXED “ID of the type of Address”
            version CDATA #FIXED "1.0"
```

With ENTITIES, it would look as follows:

With entities:

```
<!ELEMENT C (role1)>
<!ATTLIST C elementID CDATA #FIXED "ID of class C"
            type CDATA #FIXED “ID of class C”
            version CDATA #FIXED "1.0"
>
<!ENTITY % _H_ "Name">
<!ENTITY % _I_ "Address">

<!ELEMENT role1 (%_H_;|%_I_;)>
<!ATTLIST role1 elementID (ID_of_I|ID_of_H) #REQUIRED roleID CDATA #FIXED “ID
of role1”
          type (ID_of_I|ID_of_H) #REQUIRED
          version CDATA #FIXED "1.0"
>
<!ELEMENT H (%_H_)>
<!ATTLIST H elementID CDATA #FIXED "ID of class H"
            type CDATA #FIXED “ID of class H”
            version CDATA #FIXED "1.0"
>

<!ELEMENT I (%_I_)>
<!ATTLIST I elementID CDATA #FIXED "ID of class I"
            type CDATA #FIXED “ID of class I”
            version CDATA #FIXED "1.0"
>

<!ELEMENT Name (#PCDATA)>
<!ATTLIST Name elementID CDATA #FIXED "ID of Name"
            type CDATA #FIXED “ID of the type of Name”
            version CDATA #FIXED "1.0"
<!ELEMENT Address (#PCDATA)>
<!ATTLIST Address elementID CDATA #FIXED "ID of Address"
            type CDATA #FIXED “ID of the type of Address”
            version CDATA #FIXED "1.0"
```

It is important to note that the swiftML instance will not differ whether the DTD uses ENTITIES or not.

Advantages of using ENTITIES:

- The complete pattern is shown in the DTD.
- Demonstrates reusable elements.

Disadvantages of using ENTITIES:

- *Complexifies* the DTD without adding any validation restrictions.
- *Duplication of information*. This information is already in the model. The elementID and roleID reference it. One of the design rules of these ID's was to demonstrate reusability of the elements. Hence there is no need to repeat that information in the DTD.

- **DTD do not allow combining #PCDATA in a choice production**.
  (per http://www.w3.org/TR/REC-xml#dt-parentchild). Hence the following is not allowed:
  <!ELEMENT A (#PCDATA|B)>
  This is solved by describing a mixed contents (http://www.w3.org/TR/REC-xml#NT-Mixed) using the Keen closure, which is allowed:
  <!ELEMENT A (#PCDATA|B)*>
  Obviously using the Keen closure relaxes validation; but this better than the worst alternative:
  <!ELEMENT A (ANY)>
  This problem occurs when intrinsic business types are used in enumerated types.

  Suppose following example:
  <!ELEMENT Buyer (#PCDATA)>
  <!ELEMENT Seller (#PCDATA)>
  <!ELEMENT Other (Name, Address)>
  <!ELEMENT Party (#PCDATA|#PCDATA|(Name, Address))
  However, the last line is not allowed. Instead we must generate:
  <!ELEMENT Party (#PCDATA|(Name, Address))*>

- **Resolving ambiguous content models.**
  Consider the following:
  <!ELEMENT role1 ((attr1, attr2) | (attr1, att3))>.
  This is an ambiguous representation. Indeed, when attr1 is encountered, the parser still does not know whether the first or the second choice was taken.
  This is resolved by externalising 'attr1' of the XOR.
  <!ELEMENT role1 (attr1, (attr2 | attr3))>

  A more specific case:
  <!ELEMENT role1 (attr1 | (attr1, attr2) | (attr1, attr2, attr3))
  This is resolved as follows:
  <!ELEMENT role1 (att1, (att2, att3?)?)>
  meaning att1 is optionally followed by att2. If att2 is present, it is optionally followed by att3.

3.3 swiftML Schema Design rules

3.3.1 General constraints

- *Equivalence* cannot be used (DTD cannot handle multiple names for the same tag)
- *Order=all* cannot be used (DTD's expect a fixed order of elements)

3.3.2 Primitive Datatypes

The following table gives the correspondence between the primitive data types as defined in UML and the same as generated in swiftML. swiftML primitive data types are encoded as defined by W3C, defined at http://www.w3.org/TR/xmlschema-2/#dt-encoding.

Table 4 - Correspondence between UML and XML primitive data types:

UML Name	XML Name	Description
`String`	`string`	Set of finite sequences of UTF-8 characters
`Boolean`	`boolean`	Has the value space of boolean constants “True” or “False”
`Integer`	`integer`	Corresponds to 32 bits integer type
`BigDecimal`	`decimal`	Arbitrary precision decimal numbers
`Float`	`float`	Corresponds to IEEE single-precision 32 bits floating point type
`Double`	`double`	Corresponds to IEEE double-precision 64 bits floating point type
`Long`	`long`	Corresponds to 64 bits integer type
`Date`	`date`	Corresponds to a date. See ISO 8601.

Rest TBD when the XML Schema specification will be approved by W3C.

3.4 DTD vs. Schema

When S.W.I.F.T. will adopt swiftML schema’s, it is prerogative that the same swiftML instance can be validated both through DTD’s and Schema’s. Hence all constraints in either the DTD specification or the XML Schema specification that would create such incompatibility should not be supported.
Where required, meta-information that is supported in XML Schema should be added to the DTD in a manner and naming that is easily convertible to Schema’s.

3.5 Granularity of DTDs or Schemas

3.5.1 Assumptions

- We don’t foresee modular DTDs to be included in user-defined DTDs. Other organizations don’t need granular DTDs for re-use and integration in custom DTDs
- We don’t need granular DTDs for message validation (against simplicity and performance)

3.5.2 Requirements

- DTD/Schema must be independent of the network. Hence there is, at least, a DTD/Schema for the Network header and a DTD/Schema for the Business payload.
- Users need to re-use swiftML elements
- Reflect the concept of a single data dictionary

3.5.3 Solution

There is one DTD per message.

3.6 swiftML optimisation

For technical reasons, it might be necessary to optimise the swiftML structures. Examples are:

- Reducing the swiftML nesting levels
  For example instead of

```
<Buyer>
   <BIC>ABNANL2A</BIC>
</Buyer>
```

we use

```
<Buyer>ABNANL2A</Buyer>
```

- Shorter swiftML tags than their UML equivalent.

These types of optimisation will be done in the model.

4. Naming Conventions and Taxonomy

See Naming Conventions appendix

5. Character set

S.W.I.F.T. Standards recommends using UTF-8 as the (default) character encoding mechanism, for the following reasons:

- It has the most efficient method of character representation:
  - It is the shortest method to represent the characters which are currently the most commonly used in a financial environment (ASCII and EBCDIC characters)
  - It can still represent almost any known character
- It is interoperable with many other encoding schemes through (automatable) conversion algorithms.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

For a more detailed technical background, see Character Sets appendix.

6. swiftML Namespace

A UML namespace does NOT necessarily correspond to an swiftML namespace.
The swiftML repository will be maintained in **one** XML namespace called '**sw**'.
Example:

```
<sw:PmtsAcct>123-456789-0</sw:PmtsAcct>
```

For a more detailed technical background, see XML Namespace Appendix.

7. Versioning

There are two types of versions. This chapter only discusses the public version, i.e. the version known to the outside world.
All elements belonging to the swiftML repository have a version. This version is specified in the DTD through the swiftML attribute **version**.
Example:
All DTD's will also have a version number, as per the XML specification v1.0 from W3C.

A. Appendix A: Example

A.1 Snippet of a hypothetical Notice of Execution

A.1.1.1 Instance

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE NoticeOfExecution SYSTEM “NoticeOfExecution.dtd”>

<NoticeOfExecution>
   <MsgFunction>NewMessage    </MsgFunction>
   <SendersMsgRef>
      <RefNbr>gstpa12345</RefNbr>
   </SendersMsgRef>
   <Buyer>
      <PartyId>ABNANL2A</PartyId>
      <TradingCapacityIndicator>PRINCIPAL</TradingCapacityIndicator>
   </Buyer>
   <Seller>
      <PartyId>DEUTDEFF</PartyId>
   </Seller>
   <ExecutedTradeDetails>
      <FinInstrQty elementID='ABCD2008' type='DCBA0013'>
         <FaceAmt>102</FaceAmt>
      </FinInstrQty>
      <TradeDate>20000427</TradeDate>
      <RequestedSettlementDate>20000428</RequestedSettlementDate >
      <DealPrice elementID='ABCD2000' type='57A817C1'>
         <ActualPrice>
            <CcyCode>EUR</CcyCode>
            <Value>20000</Value>
         </ActualPrice>
      </DealPrice>
   </ExecutedTradeDetails>
</NoticeOfExecution>
```

A.1.2 DTD

```
<!ELEMENT NoticeOfExecution
(MsgFunction,SendersMsgRef,TrackingRef?,CommonRef?,RelatedRef?,BasketRef?,Origin
atorOfMsg?,Buyer,Seller,ExecutedTradeDetails,FI_Id?)>
<!ATTLIST NoticeOfExecution elementID CDATA #FIXED "ABCD1000"
            version CDATA #FIXED "1.0"
>
<!ELEMENT MsgFunction (#PCDATA)>
<!ATTLIST MsgFunction elementID CDATA #FIXED "ABCD1001"
               type CDATA #FIXED "string"
               version CDATA #FIXED "1.0"
>
<!ELEMENT SendersMsgRef (RefNbr)>
<!ATTLIST SendersMsgRef elementID CDATA #FIXED "ABCD1055"
               roleID CDATA #FIXED "ABCD1002"
               type CDATA #FIXED "DCBA1001"
               version CDATA #FIXED "1.0"
>
```

```
<!ELEMENT TrackingRef (RefNbr)>
<!ATTLIST TrackingRef elementID CDATA #FIXED "ABCD1056"
               roleID CDATA #FIXED "ABCD1003"
               type CDATA #FIXED "DCBA1001"
               version CDATA #FIXED "1.0"
>
<!ELEMENT CommonRef (RefNbr)>
<!ATTLIST CommonRef elementID CDATA #FIXED "ABCD1057"
               roleID CDATA #FIXED "ABCD1004"
               type CDATA #FIXED "DCBA1001"
               version CDATA #FIXED "1.0"
>
<!ELEMENT BasketRef (RefNbr)>
<!ATTLIST BasketRef elementID CDATA #FIXED "ABCD1058"
               roleID CDATA #FIXED "ABCD1005"
               type CDATA #FIXED "DCBA1001"
               version CDATA #FIXED "1.0"
>
<!ELEMENT RefNbr (#PCDATA)>
<!ATTLIST RefNbr elementID CDATA #FIXED "ABCD1007"
               type CDATA #FIXED "string"
               version CDATA #FIXED "1.0"
>
<!ELEMENT OriginatorOfMsg (PartyId)>
<!ATTLIST OriginatorOfMsg elementID CDATA #FIXED "ABCD1059"
               roleID CDATA #FIXED "ABCD1006"
               type CDATA #FIXED "DCBA0009"
               version CDATA #FIXED "1.0"
>
<!ELEMENT PartyId (#PCDATA)>
<!ATTLIST PartyId elementID CDATA #FIXED "ABCD1008"
               type CDATA #FIXED "string"
               version CDATA #FIXED "1.0"
>
<!ELEMENT Buyer (PartyId,TradingCapacityIndicator?)>
<!ATTLIST Buyer elementID CDATA #FIXED "ABCD1060"
               roleID CDATA #FIXED "ABCD1010"
               type CDATA #FIXED "DCBA0010"
               version CDATA #FIXED "1.0"
>
<!ELEMENT Seller (PartyId,TradingCapacityIndicator?)>
<!ATTLIST Seller elementID CDATA #FIXED "ABCD1061"
               roleID CDATA #FIXED "ABCD1011"
               type CDATA #FIXED "DCBA0010"
               version CDATA #FIXED "1.0"
>
<!ELEMENT TradingCapacityIndicator (#PCDATA)>
<!ATTLIST TradingCapacityIndicator elementID CDATA #FIXED "ABCD1012"
               type CDATA #FIXED ”string”
               version CDATA #FIXED "1.0"
>
<!ELEMENT ExecutedTradeDetails (FinInstrQty,TradeDate,
RequestedSettlementDate,DealPrice)>
<!ATTLIST ExecutedTradeDetails elementID CDATA #FIXED "ABCD1013"
               type CDATA #FIXED "DCBA0012"
               version CDATA #FIXED "1.0"
>
<!ELEMENT FinInstrQty (Unit|FaceAmt|CurrentFace)>
```

```
<!ATTLIST FinInstrQty elementID (ABCD2008|ABCD2009|ABCD2010) #REQUIRED
roleID CDATA #FIXED "ABCD1015"
               type (DCBA0013|DCBA0012|DCBA0011) #REQUIRED
               version CDATA #FIXED "1.0"
>
<!ELEMENT Unit (#PCDATA)>
<!ATTLIST Unit elementID CDATA #FIXED "ABCD1016"
               type CDATA #FIXED “integer”
               version CDATA #FIXED "1.0"
>
<!ELEMENT FaceAmt (#PCDATA)>
<!ATTLIST FaceAmt elementID CDATA #FIXED "ABCD1017"
               type CDATA #FIXED “float”
               version CDATA #FIXED "1.0"
>
<!ELEMENT CurrentFace (#PCDATA)>
<!ATTLIST CurrentFace elementID CDATA #FIXED "ABCD1018"
               type CDATA #FIXED “float”
               version CDATA #FIXED "1.0"
>
<!ELEMENT TradeDate (#PCDATA)>
<!ATTLIST TradeDate elementID CDATA #FIXED "ABCD2001"
               roleID CDATA #FIXED "ABCD1020"
               type CDATA #FIXED "date"
               version CDATA #FIXED "1.0"
>
<!ELEMENT RequestedSettlementDate (#PCDATA)>
<!ATTLIST RequestedSettlementDate elementID CDATA #FIXED "ABCD1063"
               roleID CDATA #FIXED "ABCD1021"
               type CDATA #FIXED "date"
               version CDATA #FIXED "1.0"
>
<!ELEMENT DealPrice (PctPrice|ActualPrice)>
<!ATTLIST DealPrice elementID (ABCD2000|EEFA30F9) #REQUIRED
               roleID CDATA #FIXED "ABCD1030"
               type (57A817C1|EEFA30F9) #REQUIRED
               version CDATA #FIXED "1.0"
>
<!ELEMENT PctPrice (#PCDATA)>
<!ATTLIST PctPrice elementID CDATA #FIXED "ABCD1031"
               type CDATA #FIXED “float”
               version CDATA #FIXED "1.0"
>
<!ELEMENT ActualPrice (CcyCode,Value)>
<!ATTLIST ActualPrice elementID CDATA #FIXED "ABCD1032"
               type CDATA #FIXED "DCBA0015"
               version CDATA #FIXED "1.0"
>
<!ELEMENT CcyCode (#PCDATA)>
<!ATTLIST CcyCode elementID CDATA #FIXED "ABCD1033"
               type CDATA #FIXED ”string”
               version CDATA #FIXED "1.0"
>
<!ELEMENT Value (#PCDATA)>
<!ATTLIST Value elementID CDATA #FIXED "ABCD1034"
               type CDATA #FIXED “float”
               version CDATA #FIXED "1.0"
>
```

B. Appendix B: Naming conventions and taxonomy scheme

B.1 Introduction

The purpose of this document is to explain the methodology to be used when generating meaningful swiftML tags. It provides the general principles of classification of swiftML tags for NextGen standards.
It is very important to have a structure in place to 'control' the way data elements are tagged.

It will allow us to keep swiftML as condensed as possible
It will limit a proliferation of different usages by different developers
It provides a way to easily trace and manage a swiftML (and UML) repository which is based on meaningful tags.

B.2 Constraints / Assumptions

- To have as much as possible a one to one mapping between UML names and swiftML tags.
- To use normalised names: to abide to the tagging constraints imposed by the W3C XML specification v1.0., C++ and Java
- To have one namespace that will contain all business elements covered by S.W.I.F.T.
- To have no elements that can be expressed in swiftML and cannot be expressed in UML.
- To have business information which is expressed as swiftML elements/values and meta data information which is expressed as swiftML attributes.
- To have DTD's and Schema's that support inheritance. Consequently, attributes and aggregates can be reused or overridden.

B.3 Naming rules

As already stated, the purpose is to have a straightforward mapping from UML names to swiftML names. Hence, all below rules apply for UML names as well as swiftML names.

B.3.1 General rules

Use the English vocabulary.
Abide to the (character) restrictions described in swiftML for naming elements:

- All names must start with an alphabetic character, '_', or ':'
- All characters following the first characters may be alphabetic characters, numeric characters, ".", '-', '_' or ':'.

Apply camel case convention:

- Names for elements and attributes may be made up of multiple words each consisting of alphanumeric characters.
- Each word starts with a capital letter.
- All white spaces between words are removed.

B.3.2 Well-known acronyms

- Use commonly used and well-known acronyms as-is, i.e. capitalised
  - ISIN
  - IBAN

- BIC
- ISO
- FI (Financial Institution)

- Separate the abbreviation from the rest of the tag using the underscore '_"
  Example: <ISO_NetProceed>

B.3.3 Collections

In swiftML, instances of collections are separated by using the plural of the word of given to the collection. In case a plural is required, an '_' is added (E.g. Accounts becomes Acct_)

B.3.4 Reusable words

Frequently used names will be abbreviated using following table:
(Non-exhaustive list)

Account	Acct
Amount	Amt
Average	Avg
Condition	Cond
Currency	Ccy
Description	Desc
Financial	Fin
Frequency	Freq
Identifier / Identification	Id
Information	Info
Instrument	Instr
Maximum	Max
Minimum	Min
Narrative	Narr
Number	Nbr
Percentage	Pct
Physical	Phys
Qualifier	Qual
Quantity	Qty
Reference	Ref
Request	Req
Transaction	Tran
…	

B.3.5 Order of the words in a name

B.3.5.1 When a tag consists of multiple words, following hierarchy must be followed, with increasing priority.

Level 1: when one of the below words is used, they must be the last word of the tag

- Acct
- Amt
- Ccy
- Date
- Party
- Qty

- Status

Level 2: when one of the below words is used, they must be the last word of the tag
- ID
- Code
- Narr
- Ref

Examples:
<ValueDate>
<ValueDateCode>

B.3.5.2 Avoid 'Of', 'In', 'The',…

AVOID	**BETTER**
QtyOfFinInstr	FinInstrQty
…	

B.4 Additional requirements

A next step is to force designers to apply above rules software-wise when creating new data-elements. A new requirement could be (cf. Word) to perform a 'grammar' check to abide to the above rules. This requirement is needed
- to ensure a consistent generation of data dictionary elements
- to have a better performing query engine

C. Appendix C Character Sets

XML specification v1.0 supports Unicode.
Unicode includes a number of different encoding schemes, which are named according to the number of bytes they need (E.g. UCS-2 which is identical to Unicode, UCS-4, etc)
ISO10646 is more sophisticated. Using mapping schemes called UCS Transformation Formats (UTF), ISO 10646 allows for a variable number of bytes to be used.
The XML processor has to recognise following character sets in the encoding declaration:

- UTF-8 and UTF-16
- ISO10646-UCS-2 and -4
- ISO8859-1 to –9
- ISO2022-JP
- Shift-JIS
- EUC-JP

The default value supported by XML parsers is UTF-8.

C.1 Requirements

- To be as interoperable as possible with character sets used in programming languages, platforms,….
- We should endeavour to keep the swiftML representation as short as possible to save space and bandwidth.
- To support as many different (types of) characters as possible (e.g. Arabic, Chinese, Cyrillic, symbols)

This means a trade-off has to be found between these last two requirements. In other words, we must choose the alternative that defines in the most efficient and uniform way the currently defined characters.
It should be noted that it is also possible in XML to declare entities that contain different character sets or encoding schemes than the rest of the document. This would mean however, that the application behind the XML parser has to be capable of supporting those different encoding schemes or character sets as well. So for simplicity reasons one supported character set/encoding scheme is preferred.

C.2 Main alternative character sets / encoding schemes

Only the most appropriate alternatives will be discussed.

C.2.1 ISO8859

Each ISO 8859 variant is tailored for a specific (Western) language.
Only one byte per character is used.
Does not support exotic character sets.

C.2.2 UCS-2 and UCS-4

Any character is represented using two respectively four bytes.
They both support (almost) any character known (e.g. Arabic, Chinese, etc.)

C.2.3 UTF-8

UTF-8 has a variable length character representation, depending on the character set that needs to be represented. UTF-8 is an encoding mechanism that can represent (due to its variable length) almost any known character. The idea here is to represent the most commonly used characters as short as possible:

- One byte each for the ASCII charset, also known as the ‘basic Latin’ block in Unicode. This is better than UTF-16, UCS-2, UCS-4, Unicode, and equivalent to ASCII.

- Two bytes each for charsets such as (but not limited to) Latin-1, Latin Extended-A and B, the IPA extensions, spacing modifier letters, combining diacritical marks, Greek, Cyrillic, Armenian, Hebrew and Arabic. These characters are represented using the same length as Unicode, UTF-16 and UCS-2
- Three bytes each to cover the vast majority of the rest of the Unicode character set. This is worse than UTF-16.
- Four bytes to potentially cover more characters than all the previous ranges put together, but it is currently nearly unused. This is no worse than UTF-16.

UTF-8 has no byte order ambiguity compared to multi-byte encoding schemes and for many purposes, can be treated simply as a string of bytes (e.g. for string searches)
Conversion from UCS-2, UCS-4 and UTF-16 to UTF-8 can be automated without conversion tables (i.e. only conversion rules).
Mostly used by C-programmers.

C.2.4 UTF-16

Mostly used by Java programmers.
Any character is represented using two bytes.
Supports (almost) any known character set.

D. Appendix D XML Namespace

D.1 Introduction

The last step in NG business modelling is the conversion from the business messages at the logical layer (represented in UML) into a physical representation, in casu XML messages. One of the chapters in the design rules for mapping from UML to XML is the naming of tags both in UML and XML. Within this context, the usage of XML namespaces has to be clearly defined.
A rule of thumb is to re-use as much as possible the names used in the logical model to tag names in XML. The way namespaces are used in XML influences the way tags are named in XML. It is however very important to understand what XML namespaces are and what they were designed for. Only then we will be able to determine the appropriate usage.

D.2 XML namespace characteristics

It provides a syntax to associate qualified names with URIs, thereby allowing certain names to be universalised.
Namespaces are partitions; they are NOT hierarchical. Consequently, the problem of reusability of elements that is neatly solved in UML with packages cannot be solved in XML using namespaces.
Using namespaces increase the processor / parser overhead (although this impact seems minimal compared to e.g. complexity of the tree).
Namespaces cannot be nested in XML. In UML however, this is possible. Hence, a method has to be found to convert a nested namespace in UML into a single namespace in XML.
Using namespaces may lengthen in some cases the messages unnecessarily, since when using multiple namespaces within a document, all namespaces will have to be defined (even if you would have a default namespace).
There is currently a lack of conforming applications.

D.3 When should XML namespaces be used

Namespaces can be used to differentiate data dictionaries. (For this purpose, a data dictionary is a representation of a business realm owned by one registration authority). This differentiation is necessary to avoid name clashes. Name clashes can (or should) only occur when a document is using elements assigned by different authorities which are not taking into account each others dictionary when assigning names.
The purpose of a namespace is to ensure the uniqueness of an element outside the business realm of that namespace. In other words, to ensure cross-boundary business interoperability between the known business realm and the unknown business realm.
The same reasoning applies to reusable elements since they imply a common business realm. Hence different namespaces should only be used in case of clearly distinct business realms (i.e. partitioned business realms). One owner of two dictionaries with overlapping business should define the names of both dictionaries as if there was only one dictionary and thus one namespace.

D.4 When should XML namespaces not be used

Namespaces should not be used to solve name clashes within the same business realm. In this case, different names should be assigned (since they have a different business meaning).
Namespaces should not be used to resolve Data Dictionary management issues (see also further).

D.5 Case study: one business realm, many namespaces

Suppose we adopt as rule to map each UMLpackage (whatever its granularity) into its own XML namespace. Hence one can divide elements into two main categories and therefore two types of namespaces:

- reusable elements belonging to a global namespace (i.e. elements that have the same behaviour throughout the different businesses, common elements)
- specific elements (i.e. elements that cannot be reused outside of that namespace)

Experience has taught us that it is extremely difficult to define the boundaries (i.e. its granularity) of a specific package and as such a specific namespace. If the boundary cannot be clearly identified (which will be the case with most boundaries since the business is a 'living thing'), the multiple namespaces will have to be managed:

- we can move elements from the global to the specific namespace or vice-versa, or
- we can make elements pertain to multiple namespaces, or
- we can have multiple namespaces in one document.

This makes the usefulness of a global namespace questionable:

D.5.1 It may add confusion to the user

Reusing elements from other namespaces may be confusing.
Example:
Suppose a payments account has been defined in the GSTPA namespace <GSTPA:PaymentsAccount>. If this element would be used in a payments environment, the user might wonder why this specifically refers to GSTPA).

D.5.2 Moves but does not remove DD management

Part of the management of data elements is now moved (not solved) to the level of namespaces. Managing multiple namespaces won't be an easy task.
It is easy to put an element in a specific namespace when one is sure that this element's behaviour is typical to that namespace, and knowing up front that it will behave differently in another namespace. However, what is the rule if you don't know (yet) whether this behaviour is unique? Does one put that element in a specific namespace, or in the global namespace (= common).
Continuing on previous example, one might decide to move "PaymentsAccount" to the common namespace, or to add the same element in the "Payments" namespace.
So how does one know up front that an element will always behave in the same way? This ignorance may result in having to manipulate where the element belongs, i.e. managing the namespaces. So instead of removing the aspect of DD management, it is simply moved.
All of the above can be reworded into one question: What is the level of granularity of the business to use different namespaces?

D.5.3 Requires criteria on the granularity of namespaces

What are the conditions to have a separate namespace?
A different market?
A different market infrastructure?
A different market practice?
Technical vs. business elements?
Throughput issues (verbose and non-verbose elements)?
Etc…

D.6 Case Study: One S.W.I.F.T. namespace

Having one namespace means any element defined in the business data dictionary will have a unique meaning. This means business experts will have to agree upon the exact usage and definition cross-market of each element. In other words, if a business element has a different usage in a different market, a different name has to be given to that element. Therefore, since any business element is unambiguously defined, it will be valid within any business that is covered by the data dictionary, and will thus not require a different namespace in XML to make it cross-market unique.
Having only one namespace could flatten and hide the concept of reusable elements and packages at syntax level. However, since all XML messages and elements are based on models, this becomes a non-

issue. Moreover, through the use of XML Schemas and RDF, it will be possible to define type hierarchies that enable the creation of new elements that have a traceable relationship with an existing and well-known element.

D.7 Open issues

- None

D.8 Recommendations

The standards department will adopt the use of one business data dictionary and the use of one (default) namespace.

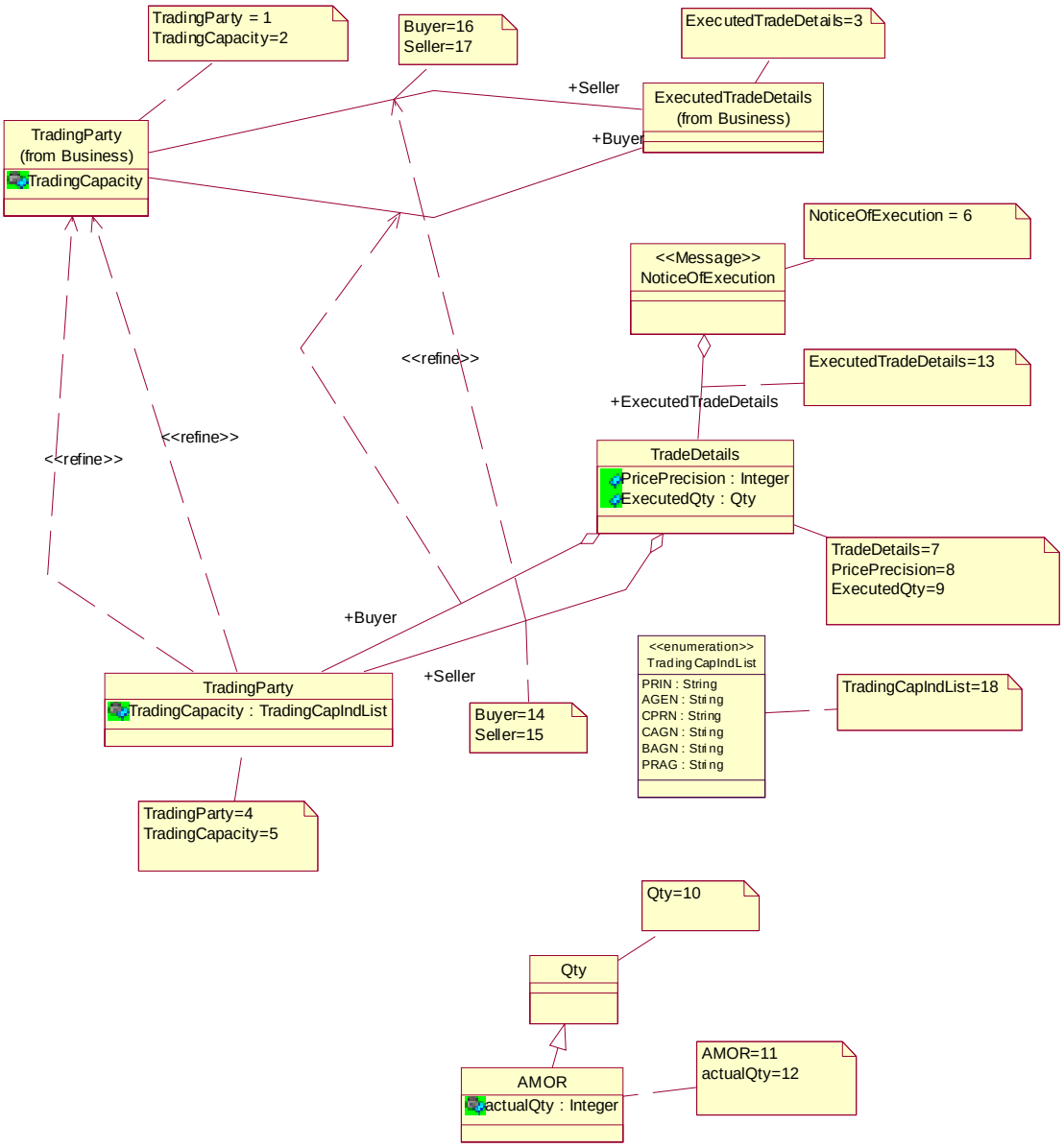
E. General Rules on SwiftID and Traceability Links

E.1 Traceability follow-up obeys to the following rules:

1. only “refine” traceability links are followed
2. Recursion: i.e. when an element reached via a traceability link also has an outgoing traceability link, this link is followed further.
3. Putting many outgoing “refine” traceability links on a single element is not allowed.
4. Cycle protection: i.e. the Standards Workstation stops following traceability link as soon as it detects that an outgoing link goes to an element it has already explored.
5. Layers independence: the DTD-implementation doesn’t take into account the existence of methodology layers. It’s only based on the existence of traceability links.
6. Independence regarding other relations: e.g. if the class, endpoint of a traceability link, is specialised this specialisation is ignored. In other words: the implementation only follows traceability links.
7. The generation of the DTD is always started at the technical layer, which implies that – at least for the message itself – a technical class must be created. Each technical element MUST have a traceability link to the logical layer.

E.2 Example

E.3 Class Diagram


TradingParty = 1
TradingCapacity=2
Buyer=16
Seller=17
ExecutedTradeDetails=3
+Seller
ExecutedTradeDetails
(from Business)
+Buyer
TradingParty
(from Business)
TradingCapacity
NoticeOfExecution = 6
<<Message>>
NoticeOfExecution
<<refine>>
ExecutedTradeDetails=13
+ExecutedTradeDetails
<<refine>>
<<refine>>
TradeDetails
PricePrecision : Integer
ExecutedQty : Qty
TradeDetails=7
PricePrecision=8
ExecutedQty=9
+Buyer
+Seller
<<enumeration>>
TradingCapIndList
PRIN : String
AGEN : String
CPRN : String
CAGN : String
BAGN : String
PRAG : String
TradingCapIndList=18
TradingParty
TradingCapacity : TradingCapIndList
Buyer=14
Seller=15
TradingParty=4
TradingCapacity=5
Qty=10
Qty
AMOR=11
actualQty=12
AMOR
actualQty : Integer

E.4 XML

```
<?xml version="1.0"?>
<NoticeOfExecution elementID="6">
   <ExecutedTradeDetails roleID="13" elementID="7" type=”7”>
     <PricePrecision roleID="8" elementID=”8” type="integer">1</PricePrecision>
     <ExecutedQty elementID="911" roleID="9 type=”11">
          <actualQty roleID="12" elementID="12” type=”integer">21</actualQty>
     </ExecutedQty>
     <Buyer roleID="16" elementID="1" type=”4”>
          <TradingCapacity roleID="2" elementID="2” type=”18">PRIN</TradingCapacity>
     </Buyer>
     <Seller roleID="17" elementID="1" type=”4”>
          <TradingCapacity roleID="2" elementID="2” type=”18">PRAG</TradingCapacity>
     </Seller>
   </ExecutedTradeDetails>
 </NoticeOfExecution>
```