Migration from Old to New Schema Language

This section gives an overview how special constructs are converted from the old to the new schema definition language of Tamino by the means of examples of typical cases of application. It finally contains a short conceptual comparison between TSD 3 and TSD 2.

The conversion of old Tamino schema definitions to new ones must take place in a manner that for each schema definition in the old language a new XML document containing an xs:schema element is generated. So the general structure of all new Tamino schema definitions looks like this:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
         xmlns:tsd = "http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition">
</xs:schema>
```

Loose or strict conversion

There are two possible conversion modes representing different strategies when transfering an old Tamino *schema* definition into a new, *XML Schema* compliant one using the Tamino Schema Editor, version 3: *loose* or *strict* conversion mode:

- **’loose’ conversion mode**

  This mode generates a schema definition which in all case will validate.

- **’strict’ conversion mode**

  This mode generates a schema definition that tries to be as near as possible to the original schema definition

Conversion of Collection-related Information

Collection-related Information
Old: `<ino:collection ino:name="MyCollection" ino:key="ID001">`

New:
 	
`<xs:annotation>`
`  <xs:appinfo>`
`    <tsd:schemaInfo tsd:schemaName = "MySchema">`
`      <tsd:collection tsd:name = "MyCollection"/>`
`    </tsd:schemaInfo>`
`  </xs:appinfo>`
`</xs:annotation>`

Conversion of Doctype-related Information

Doctype-related Information

Old

```
<ino:doctype ino:name="MyDoctype" ino:key="MyDoctype2" ino:options="READ INSERT UPDATE DELETE">
```

New

```
<xs:schema ---
    <xs:annotation>
 ---
<tsd:collection tsd:name = "MyCollection">
  <tsd:doctype tsd:name = "MyDoctype1">
   <tsd:logiDoctype>
    <tsd:content>closed</tsd:content>
   </tsd:logiDoctype>
   <tsd:mapDoctype>
    <tsd:mapDoctypeLite/>
   </tsd:mapDoctype>
  </tsd:doctype>
  <tsd:doctype tsd:name = "MyDoctype2"/>
 <tsd:logiDoctype>
  <tsd:content>closed</tsd:content>
 </tsd:logiDoctype>
 ---
  </tsd:doctype>
 ---
    </xs:annotation>

    <xs:element name = "MyDoctype1">
    </xs:element>
---
</xs:schema>
```

Old

```
<ino:doctype ino:name="MyDoctype" ino:key="MyDoctype2" ino:options="READ UPDATE ">
```

New

```
 <tsd:doctype tsd:name = "MyDoctype">
 <tsd:logiDoctype>
   <tsd:content>closed</tsd:content>
   <tsd:accessOptions>
    <tsd:read/>
    <tsd:update/>
   </tsd:accessOptions>
  </tsd:logiDoctype>
 </tsd:doctype>
```

Conversion of Parent-Child_Relationships

Parent-Child_Relationships

Parent-Child_Relationships

Old

```
<ino:node ino:name="Node1" ino:key="NewNode8" ââ,¬¦ />
<ino:node ino:name="Node2" ino:key="NewNode9" ââ,¬¦  ino:parent="NewNode8" ââ,¬¦ />
```

New

```
<xs:element name = "Node1">
 <xs:complexType>
  <xs:sequence>
   <xs:element name = "Node2" ---/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

New

```
<xs:element name = "Node1">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref = "Node2"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name = "Node2" --- />
```

Conversion of Object Type ANY

The transformation of nodes with object type "`ANY`" depends on whether *loose* or *strict* mode has been chosen. A problem arises if an `ANY` element with children occurs. This construct is allowed in TSD 2, the schema definition language of Tamino 2.3, but not in TSD 3, the schema definition language of this Tamino version. In case of strict conversion mode a mixed element is created.

Object Type ANY

Old

New (*loose* mode)

```
<xs:complexType mixed="true">
 <xs:choice maxOccurs="unbounded">
  <xs:choice>
 ââ,¬¦ <subelement1>
     <subelement2>
       .
       .
       .
 </xs:choice>
<xs:any maxOccurs="unbounded" processContent="skip">
 </xs:choice>
```

New (strict mode)

```
<xs:complexType mixed="true">
 <xs:choice>
  <subelement1>
     <subelement2>
       .
       .
       .
</xs:choice>
```

Conversion of Object Type CDATA

Here an example of an attribute definition (i.e. a node with object type "`CDATA`" in *TSD 2*) and its translation into *TSD 3* is given. This example demonstrates the use of an `xs:attribute` element in *TSD 3*.

Object Type CDATA

Old

```
<ino:node ino:name="att" --- ino:obj-type="CDATA" ino:parent="Usage3" --- />
```

New

```
<xs:element name = "---">
 <xs:complexType>
<xs:attribute name = "att" --- />
 </xs:complexType>
</xs:element>
```

Conversion of Object Type PCDATA

Here an example of an element definition (i.e. a node with object type "`PCDATA`" in *TSD 2*) and its translation into *TSD 3* is shown. This example demonstrates the use an `xs:element` element in *TSD 3*.

Object Type PCDATA

Old

```
<ino:node ino:name="---" --- ino:obj-type="PCDATA" ino:parent="Usage3" --- />
```

New

```
<xs:element name = "---" --- />
```

Conversion of Object Type CHOICE

A node with the object type "`CHOICE`" is designated to carry some alternative elements and therefore it must at least have 2 children in *TSD 3*.

Object Type CHOICE

Old

```
<ino:node ino:name="---" --- ino:obj-type="CHOICE" --- />
```

New

```
<xs:element name = "---">
 <xs:complexType>
  <xs:choice minOccurs = "0" maxOccurs = "unbounded ">
   <xs:element name = "---"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

Conversion of Object Type EMPTY

In the new Tamino schema definition language, a node of object type "`EMPTY`" is transformed to an empty complex type definition.

Object Type EMPTY

Old

```
<ino:node ino:name="---" --- ino:obj-type="EMPTY" --- />
```

New

```
<xs:complexType/>
```

Conversion of Object Type SEQ

Similarly to the translation of TSD 2 nodes with object type "`ANY`", the transformation of nodes with object type "`SEQ`" depends on the choice of *loose* or *strict* mode. A problem arises if an "`ANY`".

In case of strict conversion mode a complex type definition element in mixed mode is created that establishes a choice.In case of *loose* conversion mode, however, an `xs:sequence` element is created.

Object Type SEQ

Old

```
<ino:node ino:name="---" ---  ino:obj-type="SEQ" --- />
```

New (*loose* mode)

```
 <xs:element name = "---">
  <xs:complexType>
   <xs:sequence>
    <xs:element name = "---"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
```

New (strict mode)

```
<xs:complexType mixed="true">
 <xs:choice>
  <subelement1>
     <subelement2>
       .
       .
       .
</xs:choice>
```

Conversion of Multiplicity in element definitions

The transformation of multiplicity is ruled by this table:

TSD 2 Multiplicity	TSD 3 minOccurs and maxOccurs Attributes
Blank (1-1)	no entry
? (0-1)	minOccurs = "0" maxOccurs = "1"
* (0-n)	minOccurs = "0" maxOccurs = "unbounded"
+ (1-n)	minOccurs = "1" maxOccurs = "unbounded"

The following example illustrates how to transform the 4 possible values of the multplicity attribute of TSD 2 (`ino:multiplicity`) to the correct settings of the minOccurs and the maxOccurs attribute on TSD3 element definitions.

Multiplicity (in element definitions)

Old

```
<ino:node ino:name="el*" --- ino:multiplicity="*" ino:search-type="text"ino:obj-type="PCDATA" --- "/>
<ino:node ino:name="el+" --- ino:multiplicity="+" ino:search-type="text"ino:obj-type="PCDATA" --- "/>
<ino:node ino:name="el?" --- ino:multiplicity="?" ino:search-type="text"ino:obj-type="PCDATA" --- "/>
<ino:node ino:name="el" --- ino:multiplicity=" " ino:search-type="text"ino:obj-type="PCDATA"  --- "/>
```

New

```
<xs:element name = "el*" minOccurs = "0" maxOccurs = "unbounded"/>
<xs:element name = "el+" maxOccurs = "unbounded"/>
<xs:element name = "el?" minOccurs = "0"/>
<xs:element name = "el"/>
```

Conversion of Multiplicity in attribute definitions

This example illustrates how to transform the 2 possible values of the multiplicity attribute of TSD 2 (`ino:multiplicity`) to the correct settings of the minOccurs and the maxOccurs attribute on TSD3 attribute definitions.

Multiplicity (in attribute definitions)

Old
 	
`<ino:node ino:name="at" --- ino:multiplicity="?" --- ino:obj-type="CDATA"  />`
`<ino:node ino:name="at" --- ino:multiplicity=" "  --- ino:obj-type="CDATA" />`

New
 	
`<xs:attribute name = "at?" use = "optional" />`
`<xs:attribute name = "at" use = "required"/>`

Conversion of Data Types

Data types are mapped according to the following table of associated types :

Data type defined in Tamino 3.1 Schema Language	**Data type defined in Tamino 1.x and 2.x Schema Language**	**Meaning**
string	CHAR	String of single-byte characters
string	VARCHAR	String of single-byte characters, with variable length
string	WCHAR	String of wide characters
string	VARWCHAR	String of wide characters, with variable length
decimal	DECIMAL	Packed decimal
decimal	NUMERIC	Unpacked decimal
short	SMALLINT	Small integer (15-bit precision)
integer	INTEGER	Integer (31-bit precision)
float	REAL	Floating point with single precision
double	FLOAT, DOUBLE	Floating point with double precision
date	DATE	Date representation
time	TIME	Time representation
timeinst	TIMESTAMP	Date and time

For instance, a *TSD 2* definition of a DECIMAL data type could be translated as follows:

Data Type

Old

```
<ino:node ino:name="---"
   .
   .
   .
  ino:data-type="DECIMAL" ---  ino:precision="15" ino:scale="5"/>
```

New

```
<xs:element name = "---">
  <xs:simpleType>
   <xs:restriction base="decimal">
    <xs:precision value= "15"/>
    <xs:scale value = "5"/>
   </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Conversion of Comments

A comment in a TSD 2 schema definition is converted into an xs:annotation element in TSD 3 according to the next example.

Comments

Old

```
<ino:comment="Comment to MyNode">
```

New

```
<xs:annotation>
 <xs:documentation>Comment to MyNode
 </xs:documentation>
</xs:annotation>
```

Conversion of Constant

Constant

Old

```
ino:constant="theConstantValue" ino:map-type="Constant" ino:obj-type="CDATA"
```

New

```
<attribute name="anAttributeName" use=fixed value="theConstantValue"/>
```

Conversion of Constant

Old	New
	`<attribute name="anAttributeName" use=fixed value="theConstantValue"/>`

Conversion of Default Values

Default values that were specified with the `ino:default-value` attribute in TSD 2 are expressed with the default attribute within TSD 3 element definitions.

Default Values

Old

```
<ino:node ino:name="---" ino:obj-type="PCDATA" ino:default-value="abcdef"/>
```

New

```
<xs:element name = "---" default="abcdef " --- />
```

Conversion of Map-Type No/Native/Info-Field

The former maptypes NO, Native, Infofield are not treated as different storage options in Tamino 3.1, now they all result in the default XML storage now, if no `tsd:map` Element is specified

Map-Type No/Native/Info-Field

Old

```
<ino:node ino:map-type="Object" ino:query="@CustomerNo=%CNo" ino:object-ref="Customers/Customer"/>
```

New

```
<xs:element name = "âà,¬¦">
 <xs:annotation>
  <xs:appinfo>
    <tsd:physNative>
   <tsd:mapObjectRef>
    <tsd:collectionRef>Customers</tsd:collectionRef>
      <tsd:accessPredicate tsd:operator = "=">
     <tsd:nodeRef>/Customer/@CustomerNo</tsd:nodeRef>
    </tsd:accessPredicate>
   </tsd:mapObjectRef>
  </tsd:physNative>
 </tsd:elementInfo>
  </xs:appinfo>
 </xs:annotation>
</xs:element>
```

Conversion of Map-Type Object, Query and Object References

Map-Type Object, Query and Object References

Old

```
ino:map-type="Object"
.

.
.
ino:object-ref="Customers/Customer"/>
```

New

```
<xs:element name = "ââ,¬¦">
 <xs:annotation>
  <xs:appinfo>
   <tsd:elementInfo>
    <tsd:physNative>
   <tsd:mapObjectRef>
    <tsd:collectionRef>Customers</tsd:collectionRef>
      <tsd:accessPredicate tsd:operator = "=">
     <tsd:nodeRef>/Customer/@CustomerNo</tsd:nodeRef>
    </tsd:accessPredicate>
   </tsd:mapObjectRef>
  </tsd:physNative>
 </tsd:elementInfo>
  </xs:appinfo>
 </xs:annotation>
</xs:element>
```

Conversion of Search-Type (in the context of elements)

Search type settings of TSD 2 can be transferred to TSD 3 `tsd:text` and `tsd:standard` in a straightforward manner.

Note:
For the former search type 'no', simply no `tsd:index` element is to be specified.

Search-Type (in the context of elements)

Old

```
ino:search-type="standard"
```

New

```
<xs:element name = "a">
 <xs:annotation> <xs:appinfo> <tsd:elementInfo>
 <tsd:physNative>
   <tsd:index>
    <tsd:standard/>
    <tsd:text/>
   </tsd:index>
  </tsd:physNative>
 </tsd:elementInfo> </xs:appinfo> </xs:annotation>
</xs:element>
```

Conversion of Search-Type (in the context of attributes)

Search-Type (in the context of attributes)

Old

```
ino:search-type="standard"
```

New

```
<xs:attribute name = "at" -- >
 <xs:annotation> <xs:appinfo> <tsd:attributeInfo>
  <tsd:physNative>
   <tsd:index>
    <tsd:standard/>
    <tsd:text/>
   </tsd:index>
  </tsd:physNative>
 </tsd:attributeInfo> </xs:appinfo> </xs:annotation>
</xs:attribute>
```

No entry is generated for search type `ino:search-type="no"`.

Conversion of IGNOREUPD Option

The "`IGNOREUPD`" option of TSD 2 can straightforward be converted into a `tsd:ignoreUpdate` element from the physical part of TSD 3.

IGNOREUPD Option

Old

```
<ino:node ino:name="---"  --- ino:options="IGNOREUPD"  --- />
```

New

```
 <xs:element name = "---">
  <xs:annotation>
   <xs:appinfo>
    <tsd:elementInfo>
     <tsd:physXNode>
      ---
      <tsd:ignoreUpdate/>
     </tsd:physXNode>
    </tsd:elementInfo>
   </xs:appinfo>
  </xs:annotation>
 </xs:element>
```

Conversion of INTERNAL Option

Option INTERNAL is now defined in the logical schema as a particle.

Conversion of Adabas File Mapping

The mapping of an XML subtree to an *Adabas* file can be transferred from *TSD 2* to *TSD 3* in the following way:

Adabas File Mapping

Old

```
<ino:node ino:name="AdaFileNode" ino:key="NewNode4" ino:obj-type="EMPTY" ino:parent="MyDoctype2" ino:options="IGNOREUPD" ino:search-type="no" ino:map-type="AdaFile" ino:adabas-fnr="123" ino:adabas-encoding="ISO" ino:adabas-pwd="PWD" ino:adabas-dbid ="1"/>
```

New

```
<xs:element name = "AdaFileNode">
<xs:annotation> <xs:appinfo>
  <tsd:elementInfo>
   <tsd:physXNode>
    <tsd:mapSubTreeAdabas dbid = "1" fnr = "123"  password = "PWD" encoding = "ISO"/>
    <tsd:ignoreUpdate/>
   </tsd:physXNode>
  </tsd:elementInfo>
 </xs:appinfo>  </xs:annotation>
```

Conversion of Adabas Field Mapping

The mapping of a leaf element of the XML tree to an Adabas field can be transferred from TSD 2 to TSD 3 corresponding to this example.

Adabas Field Mapping

Old

```
<ino:node ino:name="AdaField" ino:key="NewNode7" ino:obj-type="PCDATA" ino:parent="NewNode5" ino:options="IGNOREUPD" ino:search-type="no" ino:map-type="AdaField" ino:shortname="Sh" ino:format="A" ino:length="123" ino:adabas-encoding="ISO"/>
```

New

```
<xs:element name = "AdaField">
 <xs:annotation> <xs:appinfo> <tsd:elementInfo>
  <tsd:physXNode>
<tsd:mapNodeAdabasField shortname = "Sh" format = "A" length = "123" encoding = "ISO">

                  </tsd:mapNodeAdabasField>
  </tsd:physXNode>
</tsd:elementInfo> </xs:appinfo> </xs:annotation>
```

Conversion of Adabas Multiple Field Mapping

In case of an Adabas Multiple Field Mapping (this corresponds to `ino:map-type="AdaMu"` in TSD 2) an element must be inserted (Compared to **Adabas Field Mapping**). This leads to the following conversion example:

Adabas Multiple Field Mapping

Old

```
<ino:node ino:name="AdaField" ino:key="NewNode7" ino:obj-type="PCDATA" ino:parent="NewNode5" ino:options="IGNOREUPD" ino:search-type="no" ino:map-type="AdaField" ino:shortname="Sh" ino:format="A" ino:length="123" ino:adabas-encoding="ISO"/>
```

New

```
<xs:element name = "AdaField">
 <xs:annotation>
  <xs:appinfo>
  <tsd:elementInfo>
   <tsd:physXNode>
     <tsd:mapNodeAdabasField shortname = "Sh" format = "A" length = "123" encoding = "ISO">
       <tsd:multiple>
       </tsd:multiple>
       </tsd:mapNodeAdabasField>
   </tsd:physXNode>
  </tsd:elementInfo>
 </xs:appinfo>
</xs:annotation>
```

Conversion of Adabas Periodic Group Mapping

The next example explains the transformation of an XML subtree mapped to an *Adabas* periodic group to the new Tamino schema definition language:

Adabas Periodic Group Mapping

Old

```
<ino:node ino:name="AdaPeNode" ino:key="NewNode6" ino:obj-type="PCDATA" ino:parent="NewNode5" ino:options="IGNOREUPD" ino:search-type="no" ino:map-type="AdaPe" ino:shortname="Sh"/>
```

New

```
<xs:element name = "AdaPe">
 <xs:annotation>
  <xs:appinfo>
   <tsd:elementInfo>
<tsd:physXNode>
                 <tsd:mapSubTreeAdabasPE shortname = "Sh">
                 </tsd:mapSubTreeAdabasPE>
                 <tsd:ignoreUpdate>
                 </tsd:ignoreUpdate>
               </tsd:physXNode>

   </tsd:elementInfo>
  </xs:appinfo>
 </xs:annotation>
```

Conversion of SQL Table Mapping

This example shows how to transform a schema definition of an XML subtree mapped to an SQL table that is connected to Tamino via ODBC with the following parameters:

- User ID: `UID`

- Password: `PWD`
- Schema: SRC
- Table: `USER01`
- ODBC Data source: `SRCD01`
- Primary key columns : `domain`, `login`

SQL Table Mapping

Old

```
<ino:node ino:name="MYTABLE" ino:key="MYTABLE1" ino:obj-type="SEQ" ino:parent="id0000000002" ino:search-type="no" ino:map-type="SqlTable" ino:datasource="SRCD01" ino:sqlschema="SRC" ino:sqltable="USER01" ino:sqlprimarykeys=""domain" "login"" ino:sqluserid="UID" ino:sqlpassword="PWD"/>
```

New

```
<xs:element name = "MYTABLE">
 <xs:annotation> <xs:appinfo> <tsd:elementInfo>
  <tsd:physXNode>
<tsd:mapSubTreeSQL schema = "SRC" table = "USER01" userid = "UID" password = "PWD" datasource = "SRCD01">
                    <tsd:primaryKeyColumn>domain</tsd:primaryKeyColumn>
                    <tsd:primaryKeyColumn>login</tsd:primaryKeyColumn>
   </tsd:mapSubTreeSQL>
  </tsd:physXNode>
 </tsd:elementInfo> </xs:appinfo> </xs:annotation>
```

Conversion of SQL Table Mapping with Object Referencing

An object reference that was expressed by an `ino:query` attribute in TSD 2 can now be reproduced by a `tsd:accessPredicate` element as shown by this example:

SQL Table Mapping with Object Referencing

Old

```
<ino:node --- ino:query="login = %ID04% and myName =%ID05%"/>
```

New

```
<tsd:accessPredicate>
login = <tsd:nodeParameter>/node/aLoginRef</tsd:nodeParameter>
and
myName = <tsd:nodeParameter>/node/aNameRef</tsd:nodeParameter>
</tsd:accessPredicate>
```

Conversion of SQL Column Mapping

The schema definition of a leaf node in the XML tree that is mapped to a single SQL column can be converted according to the following example:

SQL Column Mapping

Old

```
<ino:node ino:name="KEYCOL" ino:key="MYTABLE2" ino:obj-type="PCDATA" ino:parent="MYTABLE1" ino:search-type="no" ino:data-type="VARCHAR" ino:map-type="SqlColumn" ino:sqlcolumn="KEYCOL" ino:precision="50"/>
```

New

```
<xs:element name = "KEYCOL">
 <xs:annotation>
  <xs:appinfo>
  <tsd:elementInfo>
    <tsd:physXNode>
<tsd:mapNodeSQL column = "KEYCOL"></tsd:mapNodeSQL>
    </tsd:physXNode>
   </tsd:elementInfo>
  </xs:appinfo>
 </xs:annotation>
```

Conversion of Server Extension Mapping

Here an example of the transformation of a schema definition for an element mapped to a Tamino *Server Extension*

Server Extension Mapping

Old

```
<ino:node ino:name="SXSNode" ino:key="NewNode" ino:obj-type="EMPTY" ino:parent="MyDoctype2" ino:search-type="no" ino:map-type="SxsFunc" ino:sxsexternal-parseafter="FncIn" ino:sxsexternal-compbefore="FncOut" ino:sxsexternal-ondelete="FncDelete"/>
```

New

```
<xs:element name = "SXSNode">
 <xs:annotation>
  <xs:appinfo>
   <tsd:elementInfo>
    <tsd:physXNode>
     <tsd:mapSubTreeXTension>
      <tsd:onDelete/>
      <tsd:onCompose/>
      <tsd:onParse/>
     </tsd:mapSubTreeXTension>
    </tsd:physXNode>
   </tsd:elementInfo>
  </xs:appinfo>
 </xs:annotation>
```

Comparison between TSD 3 and the Former Tamino Schema Language

In the former versions of Tamino, the schema language is implemented as an XML Document Type Definition (DTD). This DTD consists of three elements, "Collection", "Doctype" and "Node", which stand in the following relation to each other:

```
<!ELEMENT
Collection (Doctype*)> <!ELEMENT Doctype (Node+)> <!ELEMENT Node
EMPTY>
```

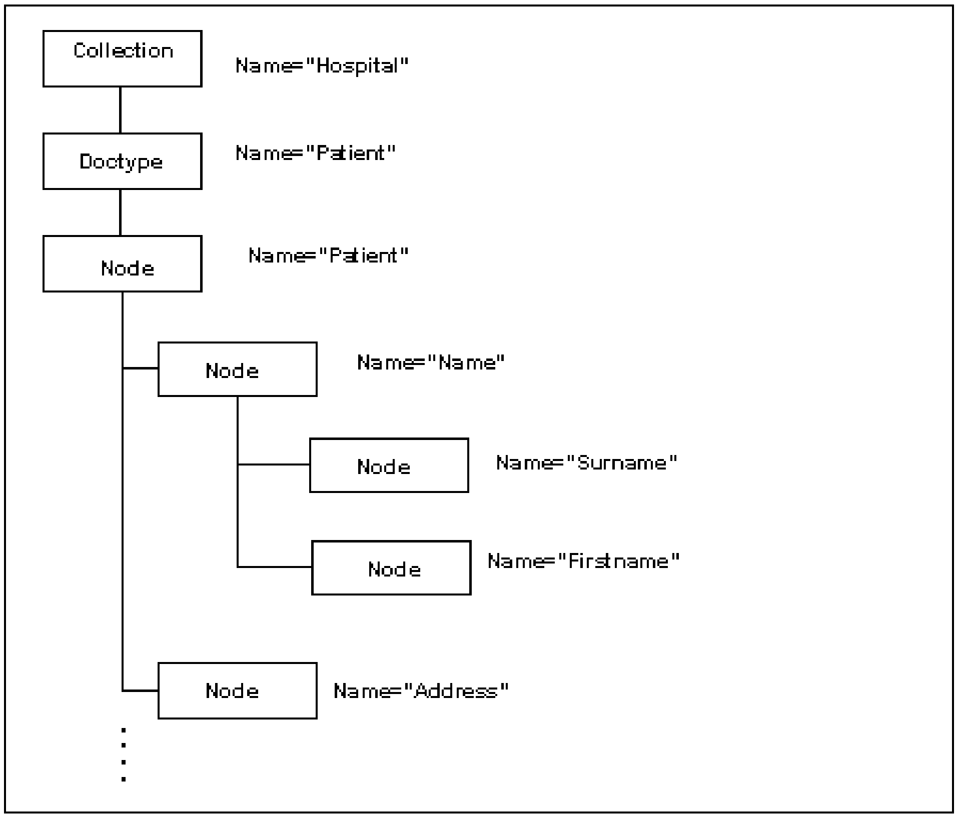
You define a schema by specifying values for the attributes on these elements. The attributes are not shown here: for an overview of the Tamino schema attributes, see Quick Reference to the Mapping Language.

- A Collection relates to a database. The Collection element therefore has no parent element, though the Data Map can contain multiple Collections. A Collection can contain zero or more Doctype elements. Collections are used to combine Doctypes for the purpose of providing parallel access to multiple Doctypes. Examples of Collections are: a hospital's administration database; an inventory database of a supplier of spare parts; an on-line library.

- A Doctype is a data definition, comparable to Tables or Views used in relational database systems. Doctypes relate to "real-world" data objects such as a patient's health record in a hospital database, a particular spare part in a spare parts database, or a book in an online library. One Tamino schema describes one Doctype, so that the two terms are sometimes used synonymously (for example, an "instance of the schema" could also be said to be "an instance of the Doctype"). A Doctype must contain one or more Node elements.

- The Node element is empty. Nodes express all the information items contained in the Doctype, for example, a patient's name, the article number of a spare part, the title of a book. Nodes can be arranged in a tree structure by using attributes to express parent-child relationships. Some Nodes are thus intermediate Nodes, describing a path to the actual information item. For example, since a patient's name can be further structured into a first name and surname, the Node representing the patient's name is defined as an intermediate Node, whereas the Nodes representing first name and surname contain character data and are therefore referred to as "terminal Nodes" or "leaves".

Using the example of a hospital database, the following figure illustrates the schema of a patient's record (Doctype "patient"):



The figure shows that Doctype structures can be seen as trees. Intermediate Nodes and their subordinate Nodes are "branches" or "subtrees". Terminal Nodes (or "leaves") are those that contain "real" data. In the above example, the Node with Name="Name" is an intermediate Node, the Nodes "Surname" and "Firstname" are terminal Nodes. A Collection, as a collection of trees, is said to be a "grove".

The figure also gives you some idea of how, by specifying appropriate attributes, subtrees or Nodes can be assigned to be stored in different data locations.

The Name attribute is available for naming the Collection, the Doctype and the Nodes. Note that the name of the Doctype must be same as the name of the root (top-level) Node. Other essential attributes are ID-declared attributes used to uniquely identify information items, and IDREF-declared attributes to representing the relationship between information items. Other attributes relate to parameters relevant to a specific database (for example, SQL or Adabas-specific attributes) and whether or how data is to be indexed.