software AG

Tamino

XQuery User Guide

Version 9.7

April 2015

WEBMETHODS

This document applies to Tamino Version 9.7.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: INS-XQUERY-97-20160318

Table of Contents

Preface v
I First Steps 1
1 Sample bib 3
2 Sample reviews 5
3 Query Examples 7
Using Constructors 8
Basic FLWOR Expressions 9
Using Filters 10
Sorting 12
Joining 14
Text Retrieval 15
Updating Documents 16
II The Concepts of XQuery 21
4 The Nuts and Bolts of XQuery 23
Expressions and Sequences 24
Retrieving Data 25
Constructors 26
Path Expressions 27
Data Types 29
Functions 30
5 FLWOR Expressions 33
6 Performing Update Operations 37
Inserting Nodes 38
Deleting Nodes 39
Renaming Nodes 40
Replacing Nodes 40
Using FLWU Expressions 41
Schema Conformance 42
Conflicts 43
Security 47
7 Calling XQuery through a Web Service 49
Setting Preferences 50
Using the Wizard 53
Using the Generated Web Service 59
8 CRUD Usage of a Tamino Doctype as a Web Service 61
Setting Preferences 94
Using the Wizard 65
Using the Generated Web Service 72
9 Advanced Usage 75
Namespaces 76
User-Defined Functions 77
Defining and Using Modules 78
Serializing Query Results 80

Collations 82
10 Text Retrieval 85
Simple Text Search 86
Context Operations 87
Highlighting Retrieval Results 88
Phonetic Searches 91
Stemming 92
Rules for Searches Using Phonetic Values and Stemming 94
Thesaurus 96
Pattern Matching 98
III Related Information 105
11 Related Information 107
Internal Resources 108
W3C Resources 108
Index 111

Preface

This document describes how to use Tamino XQuery, Software AG's implementation of the W3C XQuery language.

This documentation is directed to anyone who wants to use Tamino XQuery as a means to submit queries to a Tamino database. This includes users who wish to communicate directly with Tamino via the Tamino Interactive Interface as well as application programmers who want to deploy XQuery in their database client applications.

It introduces you to Tamino XQuery by presenting typical query use cases. It discusses the core concepts and illustrates them with several examples. You will also find pointers to information that is related to XQuery, but not part of the language.

The documentation covers the following topics:

First Steps	■ **Query Examples**
The Concepts of XQuery	■ **The Nuts and Bolts of XQuery** ■ **FLWOR Expressions** ■ **Performing Update Operations** ■ **Calling XQuery through a Web Service** ■ **CRUD Usage of a Tamino Doctype as a Web Service** ■ **Text Retrieval**
Advanced Usage	■ **Namespaces** ■ **User-Defined Functions** ■ **Defining and Using Modules** ■ **Serializing Query Results** ■ **Collations**
Related Information	■ **Internal Resources** ■ **W3C Resources**

I First Steps

XQuery is the standard query language in Tamino for performing queries on XML objects. It allows you not only to retrieve database contents, but also to *compose* your query result using constructors. With Tamino XQuery you can:

- Query XML objects from Tamino databases;
- Use simple datatypes as defined in XML Schema;
- Perform "join" operations across doctypes and collections;
- Construct new elements in your query results.

In this chapter you will see some typical queries which illustrate the capabilities of Tamino XQuery. We use two XML documents of two different doctypes, namely `reviews` and `bib`, describing reviews of computer science books and bibliographical entries. They have been published as the W3C Working Group Note **XML Query Use Cases** and are listed in the appendix for your convenience.

For each use case you will find the query, the output of Tamino, and an explanation. If you want to check these examples yourself, create a new database in the Tamino Manager, and in the Tamino Interactive Interface you can define the schema and load the data into Tamino.

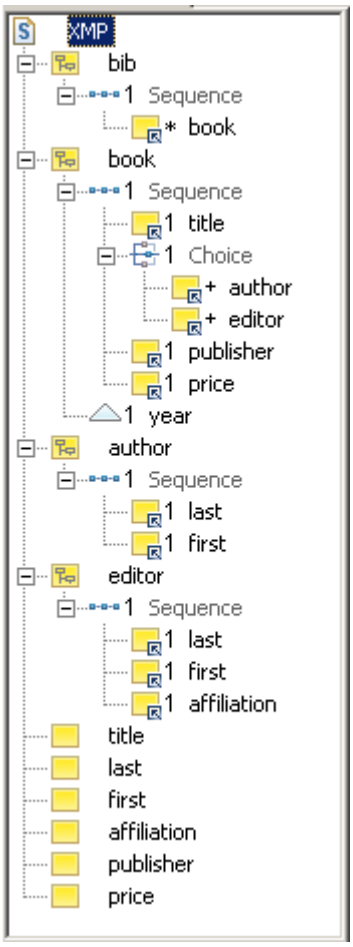
1 Sample bib

This doctype describes a simple bibliography.

This is the sample data:

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

Below you see the Schema Editor representation of this doctype on the left and the document type definition on the right.




```
<!ELEMENT bib          (book* )>
<!ELEMENT book         (title,  (author+ | editor+ ), ↩
publisher, price )>
<!ATTLIST book
   year                CDATA  #REQUIRED >
<!ELEMENT author       (last, first )>
<!ELEMENT editor       (last, first, affiliation )>
<!ELEMENT title        (#PCDATA )>
<!ELEMENT last         (#PCDATA )>
<!ELEMENT first        (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher    (#PCDATA )>
<!ELEMENT price        (#PCDATA )>
```

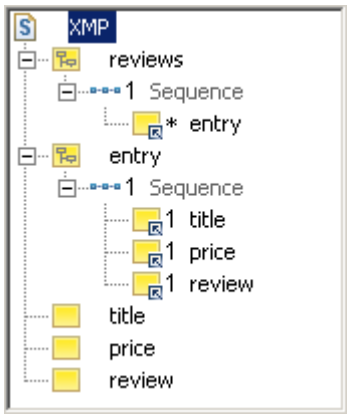
2 Sample reviews

This doctype describes reviews.

This is the sample data:

```
<reviews>
  <entry>
    <title>Data on the Web</title>
    <price>34.95</price>
    <review>A very good discussion of semi-structured database systems and ↩
XML.</review>
  </entry>
  <entry>
    <title>Advanced Programming in the Unix environment</title>
    <price>65.95</price>
    <review>A clear and detailed discussion of UNIX programming.</review>
  </entry>
  <entry>
    <title>TCP/IP Illustrated</title>
    <price>65.95</price>
    <review>One of the best books on TCP/IP.</review>
  </entry>
</reviews>
```

Below you see the Schema Editor representation of this doctype on the left and the document type definition on the right.




```
<!ELEMENT reviews   (entry*)>
<!ELEMENT entry     (title, price, review)>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT price     (#PCDATA)>
<!ELEMENT review    (#PCDATA)>
```

3 Query Examples

- Using Constructors 8
- Basic FLWOR Expressions 9
- Using Filters 10
- Sorting 12
- Joining 14
- Text Retrieval 15
- Updating Documents 16

In the following sections, XQuery keywords are rendered bold . If you have already set up a database, you can directly copy the text and paste it into the **XQuery** text field of the Tamino Interactive Interface. In the **Collection** text field, enter "XMP" for the doctypes `bib` and `reviews`, and "Hospital" for the doctype `patient`.

Using Constructors

A central concept of XQuery is compositionality: You can use *constructors* to compose new XML elements.

Query

```
<fact>This section contains {3 + 6} examples.</fact>
```

Result

This first query returns the following result from Tamino as presented by Microsoft's Internet Explorer:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
    xmlns:xql="http://metalab.unc.edu/xql/">
  - <xq:query xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
      <![CDATA[ <fact>This section contains {3 + 6} examples.</fact>  ]]>
    </xq:query>
  - <ino:message ino:returnvalue="0">
      <ino:messageline>XQuery Request processing</ino:messageline>
    </ino:message>
  - <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
      <fact>This section contains 9 examples.</fact>
    </xq:result>
  - <ino:message ino:returnvalue="0">
      <ino:messageline>XQuery Request processed</ino:messageline>
    </ino:message>
  </ino:response>
```

Tamino returns a well-formed XML document with the document element `ino:response`. It contains information about the query and its processing: the query expression itself (`xq:query`), messages about any actions before and after processing (`ino:message`) and in between the actual query result embedded into the `xq:result` element that reads:

```
<fact>This section contains 9 examples.</fact>
```

You can also reduce the output of Tamino to the bare query result. See the section **Suppressing the Response Wrapper** for details.

Explanation

Here, an *element constructor* creates the element `fact`. It contains the expression `3 + 6` that is enclosed by braces. The arithmetic expression is evaluated and the constructed element is embedded into the output document element returned by Tamino.

There are other constructors that you can use to create other types of nodes such as attribute nodes.

See AdditiveExpr and ElementConstructor in the *Tamino XQuery Reference Guide* for details.

Basic FLWOR Expressions

In this query we use the technique of constructing elements in a *FLWOR expression*:

Query

```
for    $b in input()/bib/book
return $b/title
```

Result

It returns a list of all book titles:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix environment</title>
<title>Data on the Web</title>
<title>The Economics of Technology and Content for Digital TV</title>
```

Explanation

The basic form of a FLWOR expression (pronounced: "flower") is used here. You could very roughly compare a FLWOR expression with the SQL expression `SELECT - FROM - WHERE`. The letters in FLWOR stand for the XQuery keywords `for`, `let`, `where`, `order by` and `return`, two of which are used here. Let us examine them more closely:

```
for $b in input()/bib/book
```

The `for` clause binds all values that are evaluated from the expression following the keyword `in` as an ordered sequence of items to the variable `$b`. The expression `input()/bib/book` evaluates to all instances of `bib/book` elements in the default collection (`input()`). So the variable `$b` loops over a sequence with four complete `book` elements.

```
return $b/title
```

The `return` clause uses the expression following the keyword `return` to construct the result of the FLWOR expression. Here, for each `book` element its child element `title` is returned.

See FLWORExpr in the language reference for details.

Note: According to the XQuery specification, all keywords in XQuery must be written in lower case. It is an error to use upper case or mixed case.

Using Filters

You can use filters to restrict the result sequence of a query. You can specify a filter by using a `where` clause in a FLWOR expression:

Query

```
for    $b in input()/bib/book
where  $b/@year > 1994
return
  <book>
    { $b/@year }
    { $b/title }
  </book>
```

Result

This query returns all `book/title` elements of the current collection together with the year of publication provided that the year of publication is 1995 or later:

```
<book year="2000">
  <title>Data on the Web</title>
</book>
<book year="1999">
  <title>The Economics of Technology and Content for Digital TV</title>
</book>
```

Explanation

Again, a FLWOR expression is used in the query, but this time there is an additional `where` clause:

```
where $b/@year > 1994
```

It restricts the bindings to the variable `$b` to those that meet the condition in the expression following the keyword `where`: Only those `book` elements are retained that have an attribute `year` whose numerical value is greater than 1994. So it has the same effect as the `WHERE` clause in SQL.

```
return
  <book>
    { $b/@year }
    { $b/title }
  </book>
```

An element constructor is used that creates a new element `book` which is then filled by two enclosed expressions: the first one evaluates to an attribute that is attached to the element book, the second expression is used as before.

You could also introduce an additional variable that is bound to the attribute `year` by using the `let` clause. The query then reads:

```
for    $b in input()/bib/book
let    $y := $b/@year
where  $y > 1994
return
  <book>
    { $y }
    { $b/title }
  </book>
```

The `let` clause adds an additional binding so that you can refer to `$y` instead of referring to `$b/@year`.

See FLWORExpr in the language reference for details.

Note: Although a sequence of `book` elements is not a well-formed XML element by itself, the resulting node sequence is serialized by Tamino into an `xq:result` node, which is in itself a new well-formed XML document.

Sorting

The facility of sorting is available with the expression `sort by`. You can use it for sorting query results as in the following example:

Query

```
for    $b in (input()/bib/book) sort by (title)
let    $y := $b/@year
where  $y > 1991
return
  <book>
    <year> { string($y) } </year>
    { $b/title }
  </book>
```

Result

This query returns all `book` elements sorted by their title:

```
<book>
  <year>1992</year>
  <title>Advanced Programming in the Unix environment</title>
</book>
<book>
  <year>2000</year>
  <title>Data on the Web</title>
</book>
<book>
  <year>1994</year>
  <title>TCP/IP Illustrated</title>
</book>
<book>
  <year>1999</year>
  <title>The Economics of Technology and Content for Digital TV</title>
</book>
```

Explanation

Building upon the FLWOR expression from the last example, we modified the `return` clause:

```
return
  <book>
    <year> { string($y) } </year>
    { $b/title }
  </book>
```

The year of publication is now the contents of the new element `year`. As the expression `$b/@year` represents an attribute node, we need to turn its value into a string by applying the function `string()`.

```
sort by (title)
```

All `book` elements are sorted by their child element `title` in ascending order. The FLWOR expression evaluates to a sequence of items and determines the context node for `sort by` (XQuery calls this evaluation context *inner focus*). These input items are then reordered according to the sort criterion and returned as a sequence of output items. As `book` is the context node for each input item, the result is a sequence of `book` elements sorted alphabetically by `title` in the default order, which is ascending.

An alternative version of this query is:

```
for    $b in input()/bib/book
let    $y := $b/@year
where  $y > 1991
return
  <book>
    <year> { string($y) } </year>
    { $b/title }
  </book>
sort by (title)
```

Putting the sort at the end of the `for` clause has the advantage that the data type of `title` is retained and the query can be optimized, while newly constructed nodes have no type information.

See SortExpr and fn:string in the language reference for details.

Joining

You can perform join operations on documents of different doctypes and collections:

Query

```
for    $b in input()/bib/book,
       $a in input()/reviews/entry
where  $b/title = $a/title
return
  <book>
    { $b/author }
    { $b/title }
    { $a/review }
  </book>
```

Result

This join query returns all books for which a review exists, with all authors, title and the review text.

```
<book>
  <author><last>Stevens</last><first>W.</first></author>
  <title>TCP/IP Illustrated</title>
  <review>One of the best books on TCP/IP.</review>
</book>
<book>
  <author><last>Stevens</last><first>W.</first></author>
  <title>Advanced Programming in the Unix environment</title>
  <review>A clear and detailed discussion of UNIX programming.</review>
</book>
<book>
 <author><last>Abiteboul</last><first>Serge</first></author>
 <author><last>Buneman</last><first>Peter</first></author>
 <author><last>Suciu</last><first>Dan</first></author>
 <title>Data on the Web</title>
 <review>A very good discussion of semi-structured database systems and XML.</review>
</book>
```

Explanation

A join is constructed in a similar way as in SQL: you identify the items that must match, determine the join criterion and define the output:

```
for    $b in input()/bib/book,
       $a in input()/reviews/entry
```

Two variables `$a` and `$b` are bound: `$b` is bound to all instances of `bib/book`, while `$a` is bound to all instances of `reviews/entry`. Both doctypes, bib and reviews, are available in the same collection (XMP).

```
where  $b/title = $a/title
```

The FLWOR expression is processed by repeated construction: tuples consisting of an item bound to `$a` and an item bound to `$b`. Only those tuples are retained that satisfy the condition that `$b/title` of doctype `bib` is equal to `$a/title` of doctype `reviews`. Equality is based on the value of the nodes. This is what you could call an *equijoin* in XML Query.

```
return
  <book>
    { $b/author }
    { $b/title }
    { $a/review }
  </book>
```

As before, we use constructors with embedded expressions to define the output. Note that `{ $b/author }` applies to all instances of `author` so that the third book appears with all three authors. Also, as the `author` element contains children elements, these are included as well.

Text Retrieval

You can perform text search operations by using one of the functions `tf:containsText`, `tf:containsAdjacentText` or `tf:containsNearText`. Other retrieval operations include "highlighting" of text, navigating in user-defined thesauri and searching based on phonetic similarities, word stemming or semantic relationships.

Query

```
for $a in input()/bib/book
where tf:containsText ($a/title, "UNIX")
return $a
```

Result

This query returns all `book` elements that contain the word "UNIX" regardless of the case:

```
<book year="1992">
  <title>Advanced Programming in the Unix environment</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

Explanation

All `book` elements are checked whether they contain the word "UNIX" in their `title` element independent from case. Those that contain this word will be returned as result.

As the function `tf:containsText()` is from a different namespace than the standard namespace, you need to declare this namespace first. In this namespace you will find all functions that are specific to Tamino. See `tf:containsText` in the language reference for details. The section *Unicode and Text Retrieval* contains more information about the fundamentals of word-wise search in Tamino.

Updating Documents

You can perform update operations on documents to insert, replace, rename or delete nodes or node sequences (*node-level update*). It is easy to identify any update operation, since the keyword `update` always appears at the start of the expression right after the prolog.

Note: Any update operation requires that you have permission to perform this operation. In short, Tamino checks to see if the *resulting* document is such that you may write it back into the database.

Deleting Nodes

The first simple query deletes all books from the current `bib` collection that have been edited by Darcy Gerbarg:

Query

```
update delete input()/bib/book[editor/last="Gerbarg"]
```

Result

As the result is a modification of the current collection, you receive a confirmation that the operation has been performed successfully:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
    xmlns:xql="http://metalab.unc.edu/xql/">
  - <xq:query xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
      <![CDATA[ update delete input()/bib/book[editor/last="Gerbarg"]  ]]>
    </xq:query>
  - <ino:message ino:returnvalue="0">
      <ino:messageline>XQuery Update Request processing</ino:messageline>
    </ino:message>
  - <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
      <ino:object ino:collection="XMP" ino:doctype="bib" ino:id="1" />
    </xq:result>
  - <ino:message ino:returnvalue="0">
      <ino:messageline>XQuery Update Request processed</ino:messageline>
    </ino:message>
</ino:response>
```

Explanation

From the document returned by Tamino you can see the original query expression in the marked CDATA section, and the `xq:result` element that provides information about where the update operation took place: The first document instance (`ino:id="1"`) of the document type `bib` (`ino:doctype="bib"`) in the collection XMP (`ino:collection="XMP"`).

You can use queries like `count(input()/bib/book)` to check the number of books before and after the delete operation.

See UpdateExpr and DeleteClause in the language reference for details.

Inserting Nodes

This query reinserts the book with the title "The Economics of Technology and Content for Digital TV" into the `bib` element of the current collection.

Query

```
update insert
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <author>
      <last>Gerbarg</last>
      <first>Darcy</first>
    </author>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
into input()/bib
```

Explanation

The current `bib` element is updated by inserting the `book` element as child element. The new `book` element is now the last child element of `bib`.

See UpdateExpr and InsertClause in the language reference for details.

Renaming Nodes

Having a closer look at the book just inserted, we see that we added Darcy Gerbarg as an author. However, she really is the editor and not the author of the book so we need to rename the element and insert the necessary `affiliation` element:

Query

```
update for $a in input()/bib/book
where $a/title = "The Economics of Technology and Content for Digital TV"
do (
    insert <affiliation>CITI</affiliation> following $a/author/first
    rename $a/author as editor
   )
```

Explanation

For the operation to succeed, an `editor` element must be allowed at the same hierarchical position as the `author` element. This means, they must be siblings as defined in the schema, which is the case.

See UpdateExpr and RenameClause in the language reference for details.

II The Concepts of XQuery

Tamino XQuery is an implementation of the W3C standard XML Query (XQuery). It defines a genuine XML query language that is firmly based on existing XML standards. XQuery is a functional, strongly typed language that satisfies the requirements of a database query language in the context of native XML databases much better than a path language such as XPath 1.0. In this chapter we introduce the core concepts of XQuery as they are realized in Tamino XQuery.

You should have a general understanding of how to locate and address nodes and fragments in XML documents, which means XML objects in collections of Tamino databases. There is a short refresher on XPath 1.0 in the documentation of Tamino X-Query, the previous XPath-based query language, see the pointers to related information at the end of this documentation.

In general, Tamino adheres as closely as possible to published standards. The current W3C recommendation, dated 23 January 2007, specifies version 1.0 of the language. In Tamino, the functionality which is most important and which can also be regarded as settled in terms of specification is available, together with functionality that has been present for the last years in the previous X-Query implementation. The current state of the XQuery language gives you many compelling arguments to use this implementation.

Note: In this documentation, if the term XQuery is used without further qualification, it denotes Tamino XQuery. The W3C XQuery recommendation is referred to as "W3C XQuery".

The Nuts and Bolts of XQuery

FLWOR Expressions

Performing Update Operations

Calling XQuery through a Web Service

CRUD Usage of a Tamino Doctype as a Web Service

Text Retrieval

4 The Nuts and Bolts of XQuery

- Expressions and Sequences 24
- Retrieving Data 25
- Constructors 26
- Path Expressions 27
- Data Types 29
- Functions 30

In this chapter you will learn about the nuts and bolts of Tamino XQuery. It will pave the way for a solid understanding of the whole language.

Expressions and Sequences

In XQuery, you use *expressions*. Expressions can be of different kinds, some of which can be nested in a general way. Each XQuery operator and function expects its operands to be of a certain type. This makes XQuery a functional, strongly-typed language.

Every expression evaluates to a *sequence*, which is an ordered collection of items. An *item* is either an atomic value or a node. An *atomic value* does not contain any other value and is either a primitive data type or a derived data type as defined in XML Schema. A *node* is one of the seven kinds element, attribute, namespace, text, comment, processing instruction or document node. It has an identity, because its creation is independent of its value.

A sequence can be empty, consist of only a single item (*singleton* sequence) or more items. Sequences have the following properties:

- Sequences are ordered.

  ```
  (input()/bib/book/author/first, input()/bib/book/author/last)
  ```

  Even if `last` elements appear before `first` elements in the document, in this sequence the order is as follows: first `first` elements, then `last` elements. The comma serves as concatenation operator on sequences.

  Note: In XPath 1.0, sets and node sets were always kept in forward or reverse document order, depending on the axis.

- Sequences are always flat.

  ```
  (1, 2, ("a", "b", "c"), 3, 4)
  ((1, (2)), (("a", "b", "c")), (3, 4))
  ```

  Although you can use nested sequence constructors, the result is always a “flattened” sequence. Any nested sequence items will be arranged in the same order, as if there were no nestings at all. So, both example sequences are equivalent to:

```
(1, 2, "a", "b", "c", 3, 4)
```

- Sequences may contain duplicates.

```
(input()/bib/book/author/first, input()/bib/book/author/last, ↩
input()/bib/book/author/first)
(1, 2, 3, 4, 3, 2, 1)
```

Now that there is an order on a sequence, sequence items may occur more than once in a sequence. These duplicates can have the same value or the same node identity.

Note: In XPath 1.0, a node could only appear once in a node set.

Remember that every expression in XQuery evaluates to a sequence. Even if we have an XQuery expression such as

```
let    $x := 5
return $x * 30
```

that defines a local variable `$x` and returns its value multiplied by 30, the XQuery expression, strictly speaking, returns a sequence with the single integer value 150.

In contrast to the `let` variable the type of the sequence for other expressions is constrained to be a special sequence. For example, a `for` variable is always an item (identical to a singleton sequence):

```
for    $bib in input()/bib
return $bib
```

Note: In XQuery, all keywords are written in lower case. It results in a parsing error if you use mixed or upper case.

Retrieving Data

In Tamino XQuery, there are two functions that provide access to data stored in a Tamino database. The function `input()` takes no parameters and is an implementation-defined method to assign nodes from a source to the *input sequence* which is evaluated in a query expression. In Tamino, it is always the current collection of a Tamino database that `input()` provides access to. The input sequence then consists of all document nodes of the current collection. Similarly, you can use the function `collection()` to access nodes from a collection that may be different from the default collection. The collection is specified as parameter.

`input()`	`collection("XMP")`
`input()/bib/book/title`	`collection("XMP")/bib/book/title`

The first `input()` expression returns the document instances of all doctypes in the current collection. The second `input()` expression returns a sequence of all `title` elements that are child nodes of `book` elements that are child nodes of the `bib` document element. The `collection()` expressions on the right side correspond to the `input()` expressions on the left side, provided that the current collection for the `input()` expressions is `XMP`.

In XPath 1.0, any expression locates nodes *in a single document*. However, in XQuery as well as in the previous X-Query language, expressions are evaluated with regard to a *collection of documents*. More precisely, the input for an expression is a sequence of document nodes in a collection.

Constructors

In XQuery, you can conveniently compose your query result using constructors for new elements and attributes. With constructors, you can construct new element and attribute nodes within a query expression:

```
let $a := input()/bib/book/author
return
<index type="author">
  { $a/last }
  { $a/first }
</index>
```

This XQuery expression compiles a name index from all authors of the `book` doctype in the current collection. It constructs an element `index` with an attribute `type` indicating the type of index. The `index` contains two expressions enclosed in braces. They evaluate to element nodes `last` and `first` from all `author` elements.

It is sufficient to literally write the start and end tags of an element to construct it. Whenever you need to evaluate some expression, you have to enclose it in braces.

Path Expressions

XQuery uses path expressions to locate nodes in a document tree in much the same way as XPath 1.0 defined it originally:

```
let $b := input()/bib/book/author
return $b/last

input()/patient//type
```

The first expression returns the `last` child element nodes of all `author` elements. The second expressions returns all `type` elements that are descendant nodes of the `patient` element. Here, `//` is the abbreviated syntax for `/descendant-or-self::node()/`.

The structure of a path expression has only slightly changed with regard to XPath 1.0: A path expression consists of a sequence of steps which can be distinguished into general steps and location steps. A general step is an expression that evaluates to a node sequence, e.g. the `input()` function that delivers the document nodes of the current collection. It can only be the first step in a path expression. A location step consists of three parts:

- An axis, which specifies the relationship between the set of selected nodes and the context node,
- A node test, which specifies type and/or name of the set of selected nodes, and
- Zero or more predicates, which further restrict the set of selected nodes.

Axes

XQuery supports a number of axes. An axis originates in the context node and determines the initial node sequence that is further refined by node tests and predicates. In XQuery and XPath 2.0, you can specify a path in either unabbreviated or abbreviated syntax. The following table lists each axis along with its direction (normal document order or reverse document order) and a short description. In the unabbreviated syntax, a double colon '::' follows the name of the axis.

Axis	Direction	Meaning
`ancestor::`	reverse	all ancestor nodes (parent, grandparent, great-grandparent, etc.)
`attribute::`	implementation-defined	attached attribute nodes
`child::`	normal	immediate child nodes (default axis)
`descendant::`	normal	all descendant child nodes
`descendant-or-self::`	normal	current node and all its descendant child nodes
`parent::`	reverse	parent node (or attaching node for attribute and namespace nodes)
`self::`	normal	the current node

Tamino also supports the abbreviated notation of path expressions with axes. The following table shows how they correspond to the unabbreviated axes (as defined in the W3C XQuery specification):

Abbreviation	Description
no axis	nodes along the `child::` axis satisfying node tests and optional predicates
`@`	nodes along the `attribute::` axis satisfying node tests and optional predicates
`.`	`self::node()`, which is the current node of any type
`..`	`parent::node()`, which is the empty sequence if the current node is the document node; the attaching node if the current node is an attached node (of type attribute or namespace); otherwise the parent node
`//`	`/descendant-or-self::node()/`, which is the absolute path at the start of an expression, or the relative path elsewhere

The following query expressions are thus equivalent:

1.
```
for $a in input()/bib/book
return $a/title
```
```
for $a in input()/bib/book
return $a/child::title
```

2.
```
for $a in input()/bib/book
return $a/@*
```
```
for $a in input()/bib/book
return $a/attribute::*
```

Node Tests

The node test determines the type and optionally the name of the nodes along the axis direction. For each axis, there is a principal node type: for the attribute axis, it is attribute; for other axes, it is element. You can select a node by applying one of the following node tests. The node is selected if the test evaluates to "true".

NodeTest	Description
`processing-instruction()`	a processing instruction node (regardless of name)
`processing-instruction('Literal')`	a processing instruction node with name `Literal`; if name is omitted, then the test is "true" for any processing instruction node
`comment()`	a comment node
`text()`	a text node
`node()`	a node of any type (regardless of name)
`'Name'`	a node of the principal node type with the specified name
`'prefix:name'`	according to the axis used: an element or attribute node in the specified namespace with the specified local name
`'prefix:*'`	according to the axis used: all element or attribute nodes in the specified namespace

NodeTest	Description
'*:name'	according to the axis used: all element or attribute nodes in the specified namespace with the specified local name (regardless of namespace)
'*'	according to the axis used: all element or attribute nodes

Predicates

The last, optional part of a step is one or more predicates to filter the sequence of selected nodes according to the predicate expression. This expression is always enclosed in square brackets `[` and `]`. A selected node is retained if the predicate truth value of the predicate expression evaluates to "true".

The predicate truth value is derived by applying the following rules, in order:

1. If the value of the predicate expression is an atomic value of a numeric type, the predicate truth value is true if the value of the predicate expression is equal to the context position, and is false otherwise.
2. Otherwise, the predicate truth value is the effective boolean value of the predicate expression.

The effective boolean value of an expression is false if its operand is any of the following:

- An empty sequence
- The boolean value false
- A zero-length value of type `xs:string` or `xdt:untypedAtomic`
- numeric value that is equal to zero

Otherwise, `fn:boolean` returns "true".

The filtered node sequence is ordered according to the direction of the selected axis.

Data Types

The XQuery type system is much richer than that of XPath 1.0. It uses the built-in data types as defined in XML Schema 1.0. The set of built-in data types consist of primitive types and derived types. They fall roughly into these categories:

- Boolean values (true and false)
- Numbers: decimals, floating-point numbers with single and double precision
- Character Strings
- Data types for dates, times, and durations (two of which are not yet defined in XML Schema)

- XML-specific data types such as QName and NOTATION

In addition, there are *derived types* that are derived from the primitive types. In the XML schema documentation you will find a diagram that summarizes the primitive and derived types, which are all supported by Tamino XQuery.

Expressions and functions expect operands and parameters to be of a certain type. If the required type cannot be provided, type conversion is attempted. The following general methods can be applied:

Atomization

Atomization takes place when an atomic value or a sequence of atomic values are expected. When atomizing a given value, the following cases can be distinguished: If the value is an atomic value or the empty sequence, then that value is returned. If the value is a single node, then the typed value of that node is returned. Otherwise an error is raised.

Atomization is used when processing arithmetic expressions, comparison expressions, function calls and sort expressions.

Type Promotion

During processing of arithmetic expressions and value comparisons, an atomic value can be *promoted* from one type to another. As a general rule the value of a derived type can be promoted to its base type. The value of the base type is the same as that of the original type. For example, a value of type `xs:long` can be promoted to its base type `xs:decimal` retaining its original value. Two further promotions between base types are possible: a value of type `xs:decimal` can be promoted to `xs:float`, the value being as close as possible to the original value. And a value of type `xs:float` can be promoted to `xs:double` also retaining its original value.

Functions

A number of functions that operate on different types of data and perform various tasks are defined. Most of them are defined in the W3C specification ***XQuery 1.0 and XPath 2.0 Functions and Operators***.

```
let $a := input()/bib/book
return
<p>Currently, there are { count($a) } books stored.</p>
```

In addition, Tamino XQuery provides further functions that perform full-text operations or deal with special aspects of documents stored in Tamino. These functions use the namespace *http://namespaces.softwareag.com/tamino/TaminoFunction*, usually prefixed by `tf`. They do not belong to the standard namespace *http://www/w3.org/2002/08/xquery-functions*, which is prefixed by `fn`.

Since `tf` is a predefined namespace prefix, you do not have to qualify them with their namespace nor declare the namespace.

```
for    $t in input()/bib/book
where  tf:containsText($t/title, "UNIX")
return $t
```

```
for    $a in input()/bib/book
where  $a/title = "TCP/IP Illustrated"
return tf:getCollection($a)
```

The first query uses a text retrieval function to look for all books that contain the word "UNIX" in their title. The second query uses a comparison expression to look for all books whose title is equal to the string "TCP/IP Illustrated".

5 FLWOR Expressions

FLWOR expressions (speak: flower) are at the heart of XQuery. In some way, they correspond to the SQL statement `SELECT FROM WHERE`.

A FLWOR expression contains clauses that are introduced by the keywords `for`, `let`, `where`, `order by` and `return`. It begins with at least one of the clauses `for` and `let`, may be followed by a `where` and `order by` clauses and ends with the `return` clause. The clauses `for` and `let` generate tuples of variable bindings according to the evaluations of the expressions that follow. The `where` clause restricts and filters the sequence of tuples, the `order by` clause sorts the results, and the mandatory `return` clause constructs the result of the FLWOR expression with the help of the expression that follows.

The Clauses `for` and `let`

Both `for` and `let` clauses use variables that are bound in some way to the evaluations of the expression to follow. The scope of the variable is always its enclosing FLWOR expression. If the same variable has already been bound, the variable name refers to the newly bound variable. If the variable goes out of scope, the variable name refers to the previous binding.

Both clauses bind variables, but they do so quite differently. Consider the following query expression:

```
let $a := input()/bib/book
return
<p>Currently, there are { count($a) } books stored.</p>
```

This yields the following result:

```
<p>Currently, there are 4 books stored.</p>
```

In a `let` clause each variable is bound directly to the result of an expression. A single tuple is then generated that contains all variable bindings. The `return` clause is evaluated once for this single tuple. As `$a` is bound to all instances of `book` elements, `count($a)` evaluates to the total number of books present in the collection. Since the `return` clause is invoked only once, we get the expected result. Now consider the same expression using the `for` clause:

```
for $a in input()/bib/book
return
<p>Currently, there are { count($a) } books stored.</p>
```

The result is:

```
<p>Currently, there are 1 books stored.</p>
<p>Currently, there are 1 books stored.</p>
<p>Currently, there are 1 books stored.</p>
<p>Currently, there are 1 books stored.</p>
```

Instead of creating a single tuple, the `for` clause creates tuples of variable bindings from the Cartesian product of the sequence of items to which the expressions evaluate. In this example this means that the variable is bound to the evaluation of the expression `input()/bib/book` resulting in a sequence of four document instances. So `$a` is bound four times. For each of these bindings a tuple is generated and the `return` clause is called for each tuple. The `count` function only sees one document instance at a time.

To further illustrate the processing of `for` clauses consider this example that is published in a similar form in the W3C XQuery specification:

```
for $i in (1, 2),
    $j in (3, 4)
return
 <tuple>
   <i>{ $i }</i>
   <j>{ $j }</j>
 </tuple>
```

In the example before there is only one expression in the `for` clause so that the Cartesian product is equivalent to the sequence of items that the expression evaluates to. In this query you can see that the tuples of variable bindings are created from the Cartesian product of the sequences `(1, 2)` and `(3, 4)`:

```
<tuple><i>1</i><j>3</j></tuple>
<tuple><i>1</i><j>4</j></tuple>
<tuple><i>2</i><j>3</j></tuple>
<tuple><i>2</i><j>4</j></tuple>
```

If there are `for` and `let` clauses in the FLWOR expression then the variable bindings created by the `let` clause are added to the tuples created by the `for` clause.

The `where` Clause

By using the `where` clause you can define conditions that previously generated tuples must satisfy. If the condition is met, the tuple is retained, if not, the tuple is discarded. It depends on the **effective boolean value** of the expression in the `where` clause, whether tuples will be retained or not. For the remaining tuples all variable bindings are still valid so that they can be used in the `return` clause.

A `where` clause can simply act as a filter and then represents a more verbose form of a path expression using a predicate. The query expressions

```
input()/bib/book[@year < 2000]/title
```

and

```
for    $a in input()/bib/book
where  $a/@year < 2000
return $a/title
```

are equivalent: Both return the titles of books that have been published before 2000. They differ in that the path expression additionally performs sorting in document order and elimination of duplicates according to node identity. Therefore it is not necessary to use the FLWOR expression here.

But you can use `where` clauses for more than defining predicates in path expressions. For example, they are necessary in defining a join criterion:

```
for    $b in input()/bib/book,
       $a in input()/reviews/entry
where  $b/title = $a/title
return
  <book>
    { $b/author }
    { $b/title }
    { $a/review }
  </book>
```

See also the section **Joining** for a short explanation of this example.

The `order by` and `return` Clauses

The mandatory `return` clause determines the result of the whole FLWOR expression. It is invoked for every tuple that is retained after evaluating the `where` clause. In the expression contained in the `where` clause you can use any variable bindings of the current FLWOR expression. As you can see from the join example above, this is also the place to use element constructors for tailoring your result.

If there is no `order by` clause, the order of the tuple stream is determined by the `for` and `let` clauses. Otherwise the tuples in the tuple stream are reordered in a new, value-based order. In either case, the resulting order determines the order in which the `return` clause is evaluated, once for each tuple, using the variable bindings in the respective tuples.

If the values to be compared are strings, you can specify the collation to be used (if no collation is specified, the default collation is used.)

6 Performing Update Operations

- Inserting Nodes 38
- Deleting Nodes 39
- Renaming Nodes 40
- Replacing Nodes 40
- Using FLWU Expressions 41
- Schema Conformance 42
- Conflicts 43
- Security 47

In Tamino XQuery you can perform the following elementary update operations on the node level: inserting, deleting, replacing and renaming. All update operations follow the syntactic pattern that they begin with the keyword `update`. You can either specify directly one of the update operations using `insert`, `delete`, `replace`, `rename` or you can construct more complex expressions by using a special form of the FLWOR expression that is only used for update operations. In general, all update operations have to result in well-formed documents.

Inserting Nodes

For inserting nodes you have to specify two expressions. The first expression represents the node(s) to be inserted and the second expression determines the *update node*, namely the position in all matching documents at which the insert operation should take place. As a result, the documents will contain the additional element or attribute nodes inserted at the update node. This is done differently for element and attribute nodes.

Inserting Element Nodes

Consider the following query expression that extends our current bibliography:

```
update insert
<book year="2001">
  <title>XML Schema Part 0: Primer</title>
  <editor>
    <last>Fallside</last>
    <first>David C.</first>
    <affiliation>IBM</affiliation>
  </editor>
  <publisher>World Wide Web Consortium</publisher>
  <price>0.00</price>
</book>
into input()/bib
```

This query inserts the `book` element as last child element of each `bib` element. Using the keyword `into` always tries to insert an element as the last child element of the update node. If the update node has not yet any child elements, the elements to insert will then be its first child elements.

Apart from `into` there are two other keywords that you can use when inserting element nodes. Using `preceding` the element nodes will be inserted as preceding siblings to the update node. Using `following` the element nodes will be inserted as siblings following the update node.

Inserting Attribute Nodes

Attributes are always associated with an element and there is no order on an element's attributes defined. This is why you can only use `into` to insert attribute nodes into a specified element:

```
update insert attribute edition {"1"}
into input()/bib/book[title = "TCP/IP Illustrated"]
```

This query uses a computed attribute constructor to insert an attribute node `edition` into each `book` element whose title is "TCP/IP Illustrated", marking it as the first edition (see ElementConstructor for information about computed constructors). If there is not yet such an attribute in that `book` element, then it is inserted as a new attribute provided that the resulting document is still valid according to the schema (see the section **Schema Conformance** for details). If there is already an attribute `edition`, then the operation is rejected and its content will not be overwritten. In this case, you can replace the contents of the node by using the `replace` expression.

Deleting Nodes

You can remove nodes by using the `update delete` expression. This is not limited to element nodes:

```
update delete input()/reviews/entry/review/text()

update delete input()/bib/book/author[2]
```

The first query effectively blanks all review texts: from all `review` elements the content, which is retrieved by the node test `text()`, is deleted. However, the `review` elements themselves are retained. The second query expression deletes all `author` element nodes that are the second `author` child nodes of `book` elements.

The above queries have no impact on the validity of documents according to the schema. However, if you want to delete necessary element nodes, this will usually fail due to the required validity, see the section **Schema Conformance**. But you can delete nodes such as comment nodes that are siblings of the root element.

Furthermore, you can delete whole documents as in the following query:

```
update for $a in input()/reviews
do ( delete root($a) )
```

This query deletes all `reviews` documents, since the root function returns the document nodes of all `reviews` documents in the current collection.

Renaming Nodes

For element and attribute nodes as well as for processing instruction nodes you can change the name of the node by using the `rename` operation.

```
update rename input()/bib/book/@year as jahr
```

This query translates the `year` attribute name of `book` elements into German. All `rename` operations on elements and attributes have direct impact on the schema validity of documents. See the section **Schema Conformance** for details.

Replacing Nodes

The syntax for replacing nodes is similar to the `insert` expression: The first expression determines the update node while the second expression specifies the replacing node(s). You can replace an element node as in the following query:

```
update replace input()/bib/book/title[. = "TCP/IP Illustrated" ]
          with <title>TCP/IP Illustrated I. The Protocols</title>
```

This replaces the `title` element with the content "TCP/IP Illustrated" with a `title` element that has the contents "TCP/IP Illustrated I. The Protocols". You can also replace the text node of the element directly:

```
update replace input()/bib/book/title[. = "TCP/IP Illustrated" ]/text()
          with text{"TCP/IP Illustrated I. The Protocols"}
```

As the replacement can be any valid XQuery expression, you can also perform replacements with nodes containing other nodes such as:

```
update replace input()/bib/book[title = "TCP/IP Illustrated" ] with
  <book year="1995">
    <title>TCP/IP Illustrated II. The Implementation</title>
    <author><last>Stevens</last><first>W.</first></author>
    <author><last>Wright</last><first>G.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>67.99</price>
  </book>
```

This query completely replaces the `book` with the title "TCP/IP Illustrated". You can also replace attributes:

```
update replace input()/bib/book[title = "TCP/IP Illustrated"]/@year
         with attribute year {"2003"}
```

Anticipating a new edition of this book, the `year` attribute is replaced with an attribute of the same name and the contents "2003", using a computed attribute constructor.

Using FLWU Expressions

There is a variant of the FLWOR expression that allows you to use the flexibility of FLWOR expressions with update operations. Principally you could rewrite all of the update queries shown above as FLWU expressions. The query

```
update delete input()/bib/book/author[2]
```

is equivalent to the following query using a FLWU expression:

```
update for $a in input()/bib/book
do delete $a/author[2]
```

There are two differences to a regular FLWOR expression: The keyword `update` always appears in front of the `for` clause. The `return` clause is replaced with a `do` clause that is followed by one or more update expressions. Using a FLWU expression always binds at least one variable, which allows query expressions that are not possible without. Consider the following query using the patient database:

```
update for $a in input()//doctor
       let $b := $a/@pager
       where starts-with($b, "3")
    do replace $b
       with attribute pager { string-join(("11", $b), "-") }
```

The numbers of all doctor's pagers that start with "3" are prepended with "11-". The `for` clause creates tuples with `doctor` elements somewhere and binds them to variable `$a`. For each of the tuples a variable binding for `$b` is added containing the value of the attribute `pager`. The tuples are then restricted to those that meet the condition that the value of `$b` (the pager number) starts with the value "3". For these tuples the contents of the `pager` attribute is replaced by using a computed attribute constructor that constructs the attribute `pager` with the concatenation of the strings "11-" and the previous pager number. With the sample documents *atkins.xml* and *bloggs.xml* stored, one pager number is affected ("342") and will thus be changed to "11-342" in two tuples found.

Schema Conformance

In contrast to other query operations it is the distinguishing property of update operations that they *modify* XML documents. This can lead to problems. Consider the following query:

```
update delete input()/bib//affiliation
```

Submitting this query proves not to be successful:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
    xmlns:xql="http://metalab.unc.edu/xql/">
  - <xq:query xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
      <![CDATA[ update delete input()/bib//affiliation ]]>
    </xq:query>
  - <ino:message ino:returnvalue="0">
      <ino:messageline>XQuery Update Request processing</ino:messageline>
    </ino:message>
  - <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
      <ino:object ino:collection="XMP" ino:doctype="bib" ino:id="2" />
    </xq:result>
  - <ino:message ino:returnvalue="7730">
      <ino:messagetext ino:code="INOXDE7730">(cvc-model-group.1):invalid end of sequence</ino:messagetext>
      <ino:messageline>Line 0, Column 0: [element name: editor]</ino:messageline>
    </ino:message>
  </ino:response>
```

To understand why this operation has failed, the first step is to look up the explanation to message INOXDE7730 which says: “A sequence does not contain all required elements. In particular, the sequence ends without containing all required elements.” The information which elements are required at which place is in the **schema definition** and that states that inside an `editor` element an `affiliation` element is required as last child element and may not be omitted. The query would thus result in a document instance that is not valid according to the schema and this is why it is rejected by Tamino.

In general, documents must be valid according to the schema definition. This also includes documents on which an update operation has been performed. Let us reconsider a query from the section **First Steps**:

```
update for $a in input()/bib/book
where $a/title = "The Economics of Technology and Content for Digital TV"
do (
    insert <affiliation>CITI</affiliation> following $a/author/first
    rename $a/author as editor
   )
```

In this FLWU expression there are two update operations: an insert operation and a rename operation. Taken for themselves, both update operations would fail because the document gets invalid: an `author` element may not have an `affiliation` element as a child, and an `author` element cannot

be simply renamed to `editor` because that element requires an `affiliation` element as last child element. However, in this query both update operations are enclosed in the `do` clause and the operation succeeds. Why?

If you have more than one update operation in a single query expression, the order in which the update operations are performed is not relevant. For all tuples that remain after applying the `where` clause, *both* update operations are performed resulting in temporary documents. If all these temporary documents conform to the schema definition, the operation as a whole succeeds. If at least one of the documents is no longer valid, the operation fails.

There are situations in which you want to perform update operations in such a way that they violate the schema. As a resort, you can modify the validation mode in the schema description. The Tamino Schema Editor normally enforces validation so that you will find the following information for the document element:

```
<tsd:logical>
  <tsd:content>closed</tsd:content>
</tsd:logical>
```

Closed content means that validation is strict. To make it lax, you need to change the schema description and update it into the database. See the section Open Content vs. Closed Content Validation for details.

Conflicts

FLWU expressions allow several elementary update operations. If you specify more than one update operation, the order in which these operations are performed does not play a role. However, there are some situations that can lead to the following conflicts:

1. The result of the elementary update operations is ambiguous. This can have one of the following reasons:

   a) The result of the elementary operations depends on the order in which they are performed (they are not commutative).

   b) Inserting more than one attribute with the same name into an element node or inserting an attribute that already exists.

   c) Inserting into the position preceding or following the update node if more than one `insert` operation acts on the same node (not commutative).

2. An elementary update operation is performed that has no effect on the result because of other elementary update operations. This can happen if an elementary update operation affects a node that may no longer exist by the time the operation is performed.

Note: In any of these conflict cases, the update operation is rejected by Tamino.

The following tables summarize the possible conflicts. The first table shows the conflicts for operations on one element node, the other two are subsets of the first table, since not all operations can be performed on any kind of node. As an example, if you use a `replace` operation and an `insert preceding` operation on the same element node, conflict 2 arises. If you try to delete and replace the same comment node in a single update operation, conflicts 1a and 2 arise.

Operations on one element node	`delete`	`replace`	`rename`	`insert attribute`	`insert into`	`insert preceding`	`insert following`
`delete`	2	1a and 2	2	2	2	2	2
`replace`		1a and 2	2	2	2	2	2
`rename`			1a	—	—	—	—
`insert attribute`				1b	—	—	—
`insert into`					1a	—	—
`insert preceding`						1c	—
`insert following`							1c

Operations on one attribute or PI node	`delete`	`replace`	`rename`
`delete`	2	1a and 2	2
`replace`		1a and 2	2
`rename`			1a

Operations on one node of some other kind	`delete`	`replace`
`delete`	2	1a and 2
`replace`		1a and 2

Examples

1a Two `insert into` operations on the same node:

```
update for $a in input()/patient
where $a/name/surname = "Atkins"
do ( insert <middlename>J.</middlename> into $a/name
     insert <title>Prof.</title> into $a/name
   )
```

Although the schema allows the elements `middlename` and `title` to be inserted as child elements of `name`, the result of the operation depends on the order of the elementary update operations. In this case you can use two successive elementary update operations as a workaround:

```
update for $a in input()/patient
where $a/name/surname = "Atkins"
do insert <middlename>J.</middlename> into $a/name
```

This inserts the `middlename` as new child element of `name`, in the order that is prescribed by the schema.

```
update for $a in input()/patient
where $a/name/surname = "Atkins"
do insert <title>Prof.</title> into $a/name
```

And this second FLWU expression then inserts the `title` element.

Note that this could also be performed in a single operation, when the content is declared as a sequence:

```
update for $a in input()/patient
let   $i:= (<middlename>J.</middlename>,<title>Prof.</title>)
where $a/name/surname = "Atkins"
do ( insert $i into $a/name )
```

1b Provided that the `patient` nodes that are retained after processing the `where` clause do not contain a `type` element with the attribute `brand`:

```
update for $a in input()/patient
where $a/name/surname = "Bloggs"
do ( insert attribute brand {"Somnex"} into $a//type
     insert attribute brand {"Cardiovelocimil"} into $a//type )
```

This conflict cannot be resolved: the operation fails with the message INOXQE 6450. This query also fails, if there already existed an element `$a//type` with an attribute `brand`.

1c This query tries to replace the `book` with the title "TCP/IP Illustrated" and to add the other entries for the complete three-volume set:

```
update for $a in input()/bib/book
       let   $title := $a/title
       where $title = "TCP/IP Illustrated"
    do (
         replace $title/text() with string-join(($title/text(), "I. The ↩
Protocols"), " ")
         insert
               <book year="1995">
                 <title>TCP/IP Illustrated II. The Implementation</title>
                 <author><last>Stevens</last><first>W.</first></author>
                 <author><last>Wright</last><first>G.</first></author>
                 <publisher>Addison-Wesley</publisher>
                 <price>67.99</price>
                </book>
           following $a
         insert
               <book year="1996">
                 <title>TCP/IP Illustrated III.</title>
                 <author><last>Stevens</last><first>W.</first></author>
                 <publisher>Addison-Wesley</publisher>
                 <price>49.95</price>
                </book>
           following $a
        )
```

The update nodes consist of `book` elements that have a child element `title` with the content "TCP/IP Illustrated". Then three update operations are performed:

1. `replace`
   For all `book`s found, the contents of the `title` element is replaced with the previous contents concatenated with the string "I. The Protocols" using a blank as separator.
2. **Two `insert` Operations**
   For all `book`s found, the two specified XML fragments are inserted as sibling elements following the `book` element.

In this query two `insert following` operations act on the same update node. The result is ambiguous regarding the order of the inserted elements (not commutative). Volume 2 would be inserted as next sibling to volume 1, and then volume 3 could either be inserted as next sibling to volume 1 (thus shifting volume 2 to the next position), or it could be inserted after the just inserted volume 2.

As a solution, you can define a sequence to be inserted:

```
update for $a in input()/bib/book
       let   $title := $a/title
       let $i := (<book year="1995">
                    <title>TCP/IP Illustrated II. The Implementation</title>
                    <author><last>Stevens</last><first>W.</first></author>
                    <publisher>Addison-Wesley</publisher>
                    <price>67.99</price>
                   </book>,
                   <book year="1996">
```

```
                  <title>TCP/IP Illustrated III.</title>
                  <author><last>Stevens</last><first>W.</first></author>
                  <publisher>Addison-Wesley</publisher>
                  <price>49.95</price>
                 </book>)
      where $title = "TCP/IP Illustrated"
   do (
      replace $title/text()
        with  string-join(($title/text(), "I. The Protocols"), " ")
      insert $i following $a        )
```

2 Two delete operations on the same node:

```
update for $a in input()//doctor
do ( delete $a
     delete $a)
```

Two `rename` operations on the same node:

```
update for $a in input()//doctor
do ( rename $a as surgeon
     rename $a as dentist )
```

These conflicts cannot be resolved: both operations fail with the message INOXQE 6451.

Security

Apart from any conflicts, Tamino checks for each update operation whether the resulting documents may be written in that form by the user. If permission is not granted by Tamino, the operation is rejected.

7 Calling XQuery through a Web Service

- Setting Preferences 50
- Using the Wizard 53
- Using the Generated Web Service 59

Tamino provides a plugin wizard that enables you to expose XQuery functions that are defined in an XQuery module as a web service. The wizard generates a web service archive that can subsequently be processed either manually or by means of the Software AG Web Services Stack (WSS).

The following topics are discussed in this chapter:

Setting Preferences

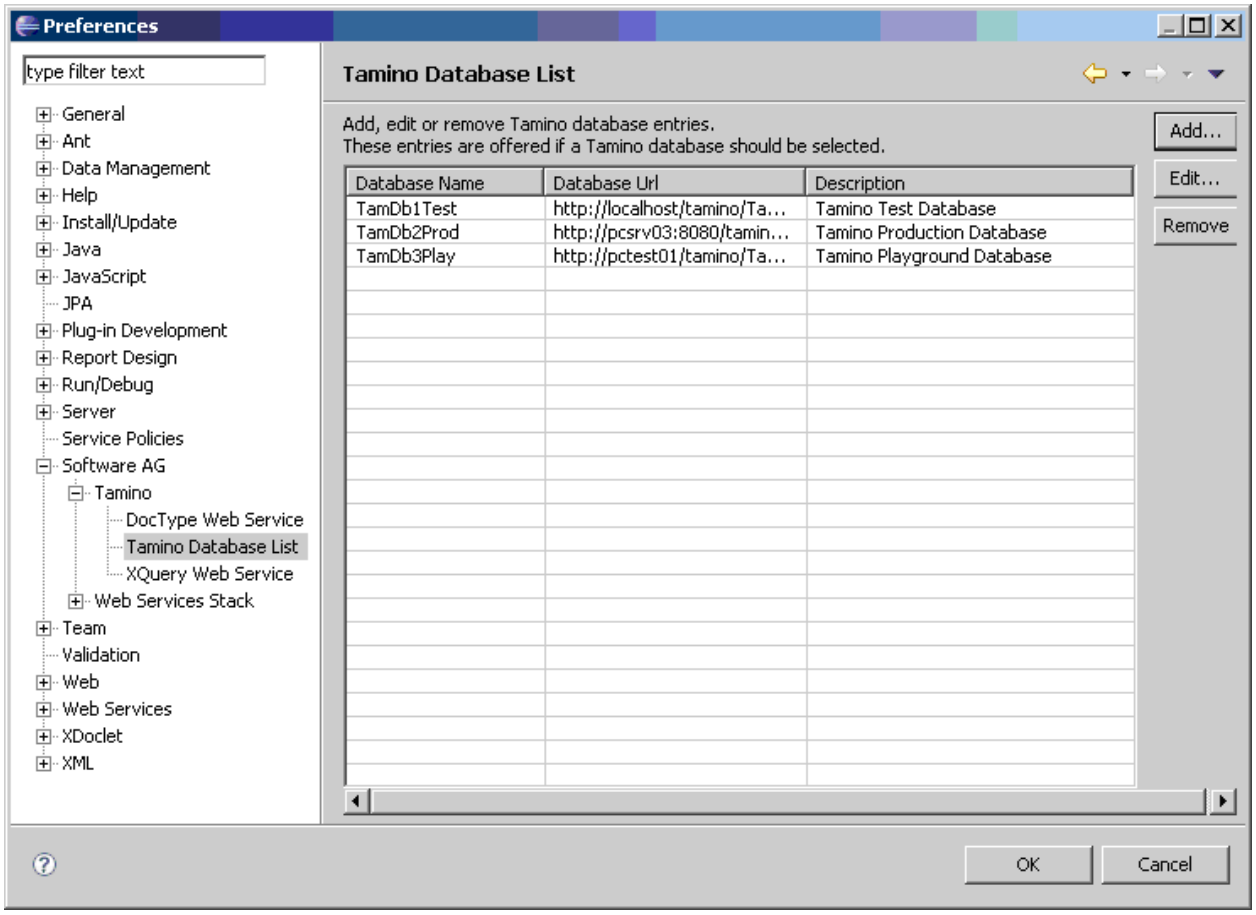
Before using the wizard to generate a web service archive, you can set preferences that specify its behavior. There are two sets of preferences:

- **Database selection**;
- **XQuery Web Service preferences**.

Database Selection

In a later step, the XQuery web service wizard prompts you to select a Tamino database from a selection list. In order to populate the selection list, proceed as follows.

In Eclipse, choose **Window** > **Preferences**. Then, in the navigator tree, select **Software AG** > **Tamino** > **Tamino Database List**. The list of Tamino databases available for selection is displayed, as shown in the following screenshot:

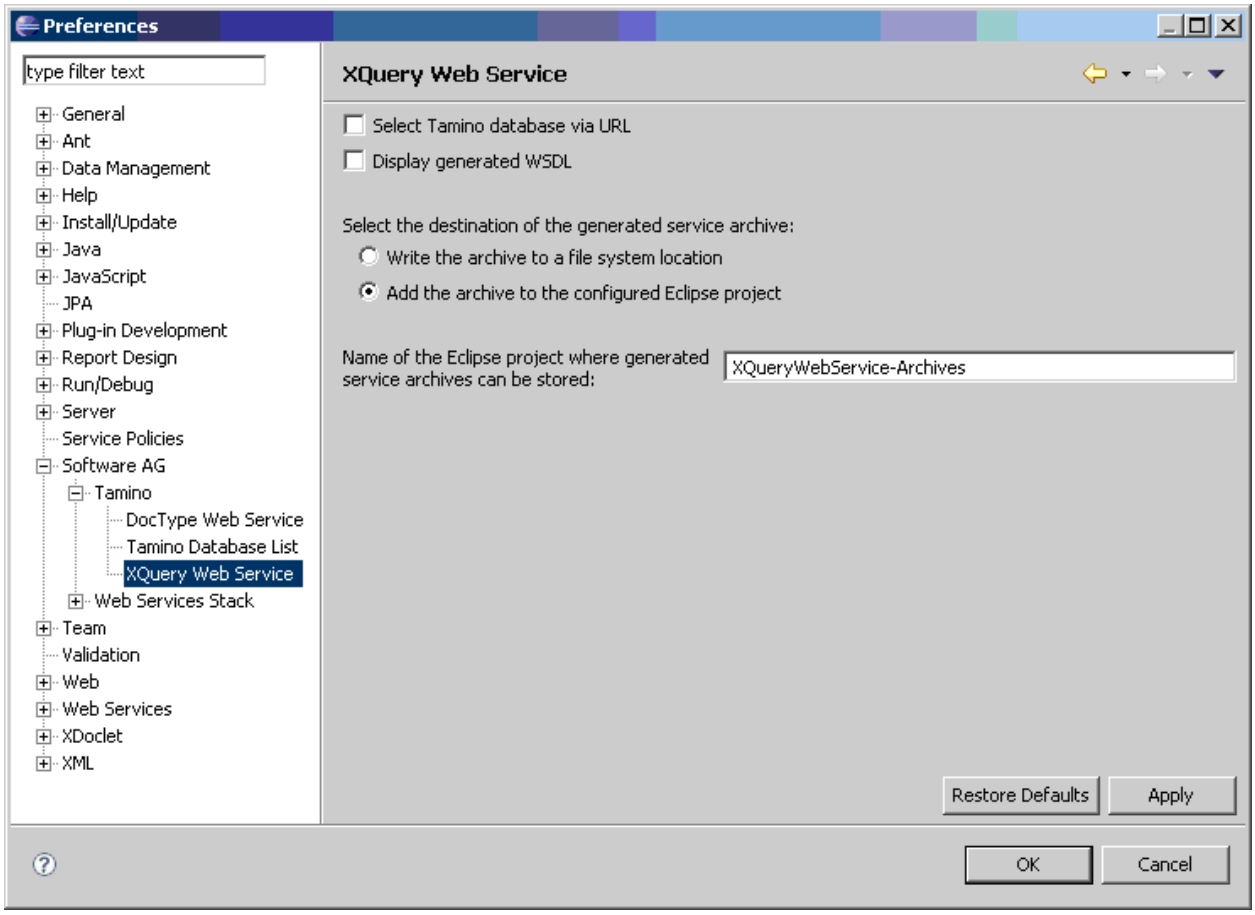


You can now use the **Add...**, **Edit...** and **Remove** functions to modify the selection list as required.

Note: In order to generate a web service successfully, the database selection list must contain at least one entry.

XQuery Web Service Preferences

In Eclipse, choose **Window** > **Preferences**. Then, in the navigator tree, select **Software AG** > **Tamino** > **XQuery Web Service**. The list of preferences for the Tamino XQuery web service wizard is displayed, as shown in the following screenshot:



The following paragraphs describe the available preferences.

Select Tamino database via URL

This switch specifies whether the database selection list that appears in subsequent wizard screens should display the names of the databases or their URLs.

Default: The database selection list displays the databases by name.

Display generated WSDL

This switch specifies whether the WSDL file that is generated by the wizard should be shown to the user; at the same time, the user is given the opportunity of saving the WSDL file in the file system.

Default: Do not display the WSDL file.

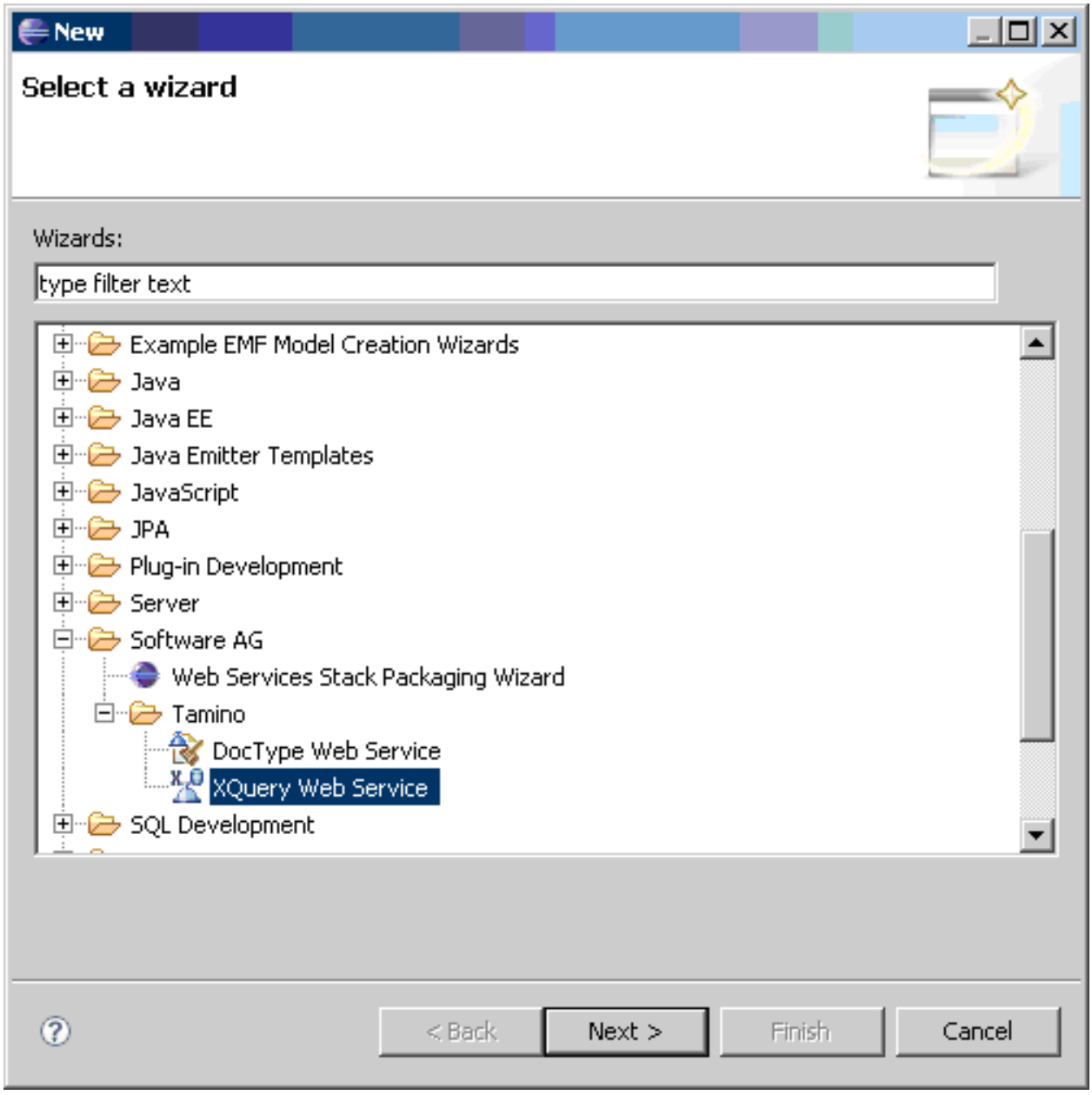
Select the destination of the generated service archive

This specifies whether the generated service archive should be stored to a user-specified file system location, or in an Eclipse project (where it can be further processed by means of the Software AG Web Services Stack). You can also specify the name of the desired Eclipse project here.

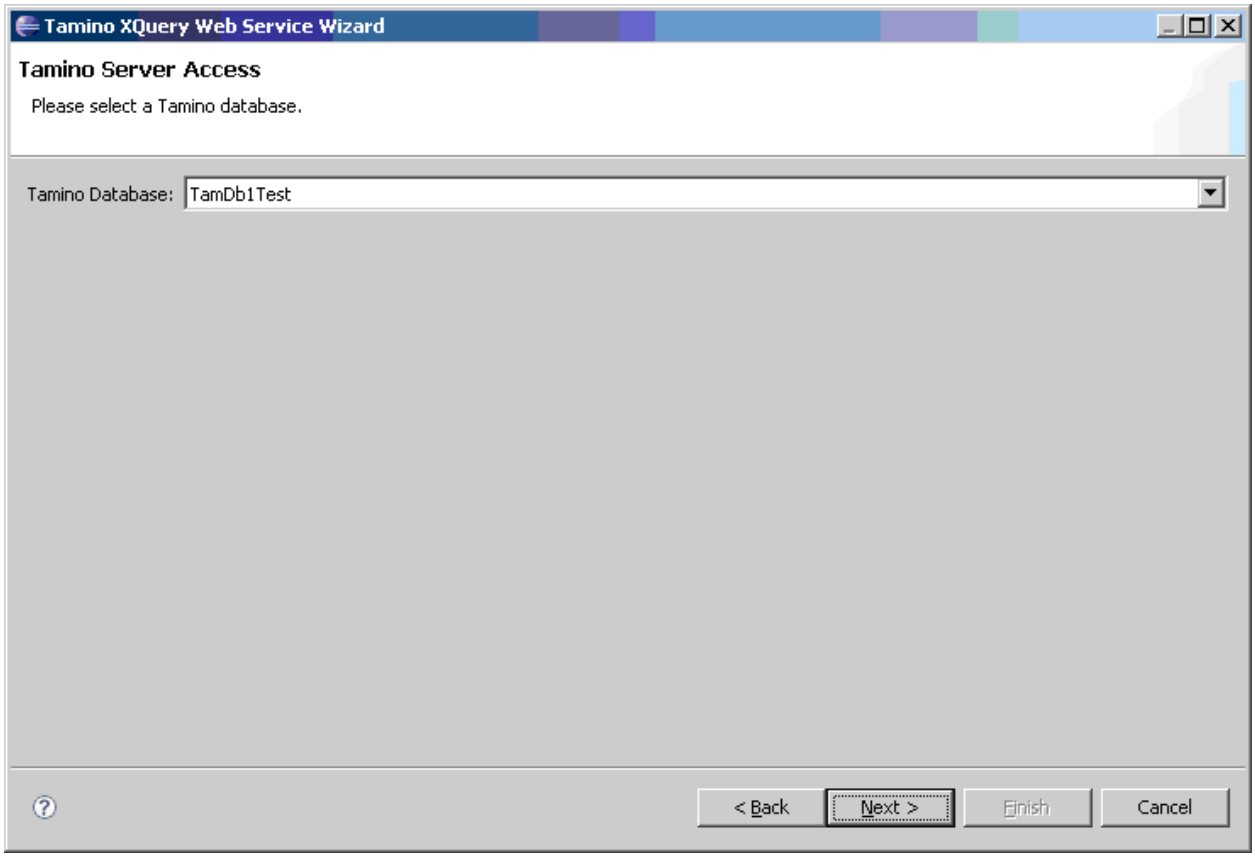
Default: Store the generated service archive in an Eclipse project named *XQueryWebService-Archives*.

Using the Wizard

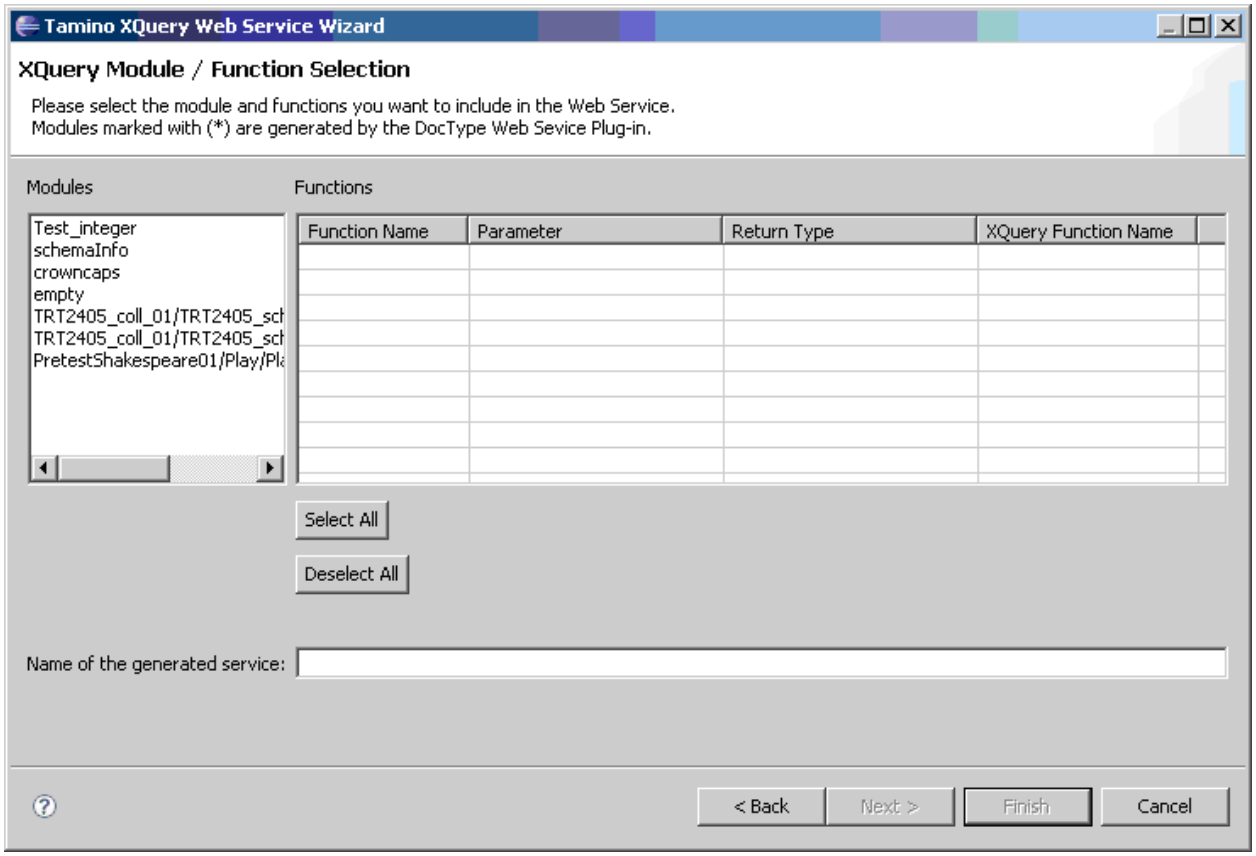
The XQuery Web Service wizard plugin runs in your Tamino Eclipse environment. To start the wizard, choose **File** > **New** > **Other**; then, in the tree view, select **Software AG** > **Tamino** > **XQuery Web Service**, as shown in the following screenshot:



Choose **Next**. The XQuery Web Service wizard starts, and displays a page from which you can select a Tamino database from the pulldown database list, which was constructed using the preferences as **described above**:

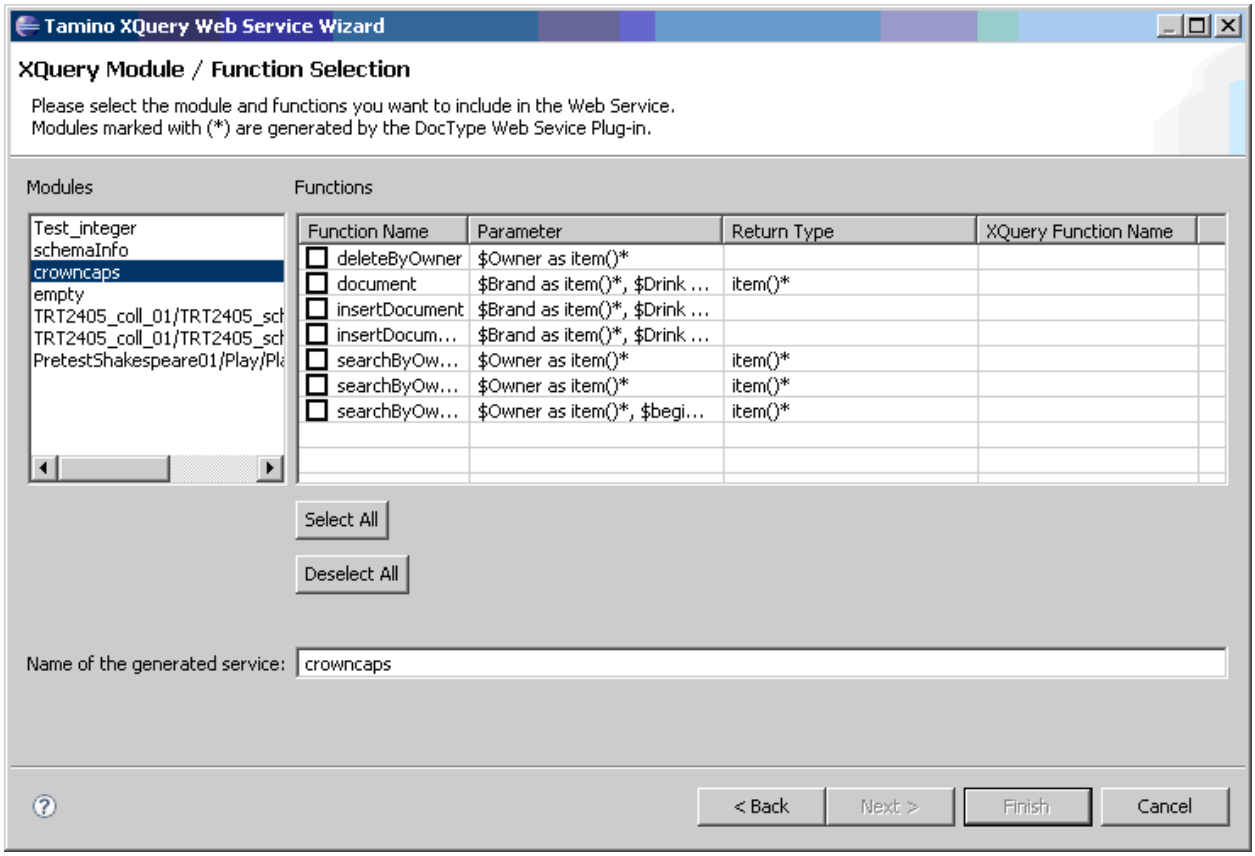


Choose **Next** to display the next page of the wizard:

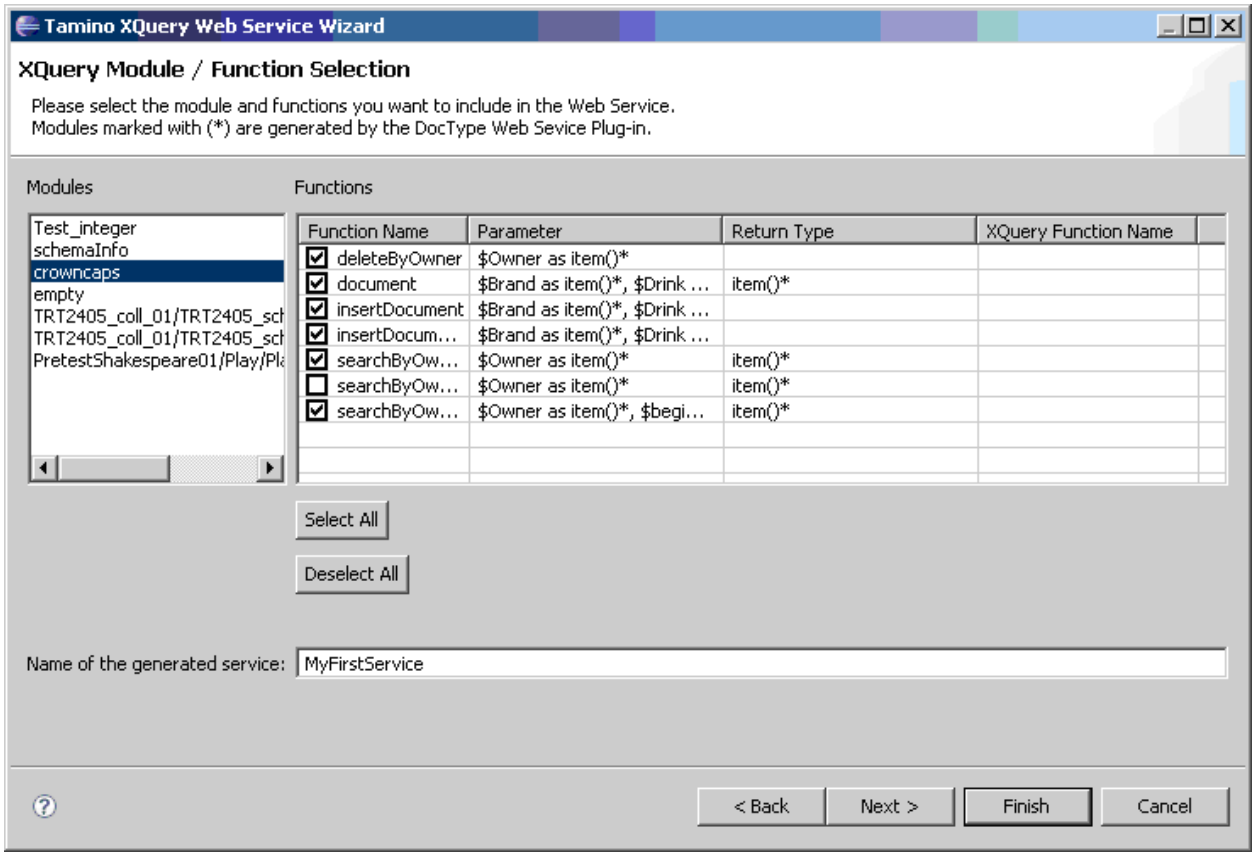


Here you can select an XQuery module from the chosen Tamino database. A default name is suggested for the web service.

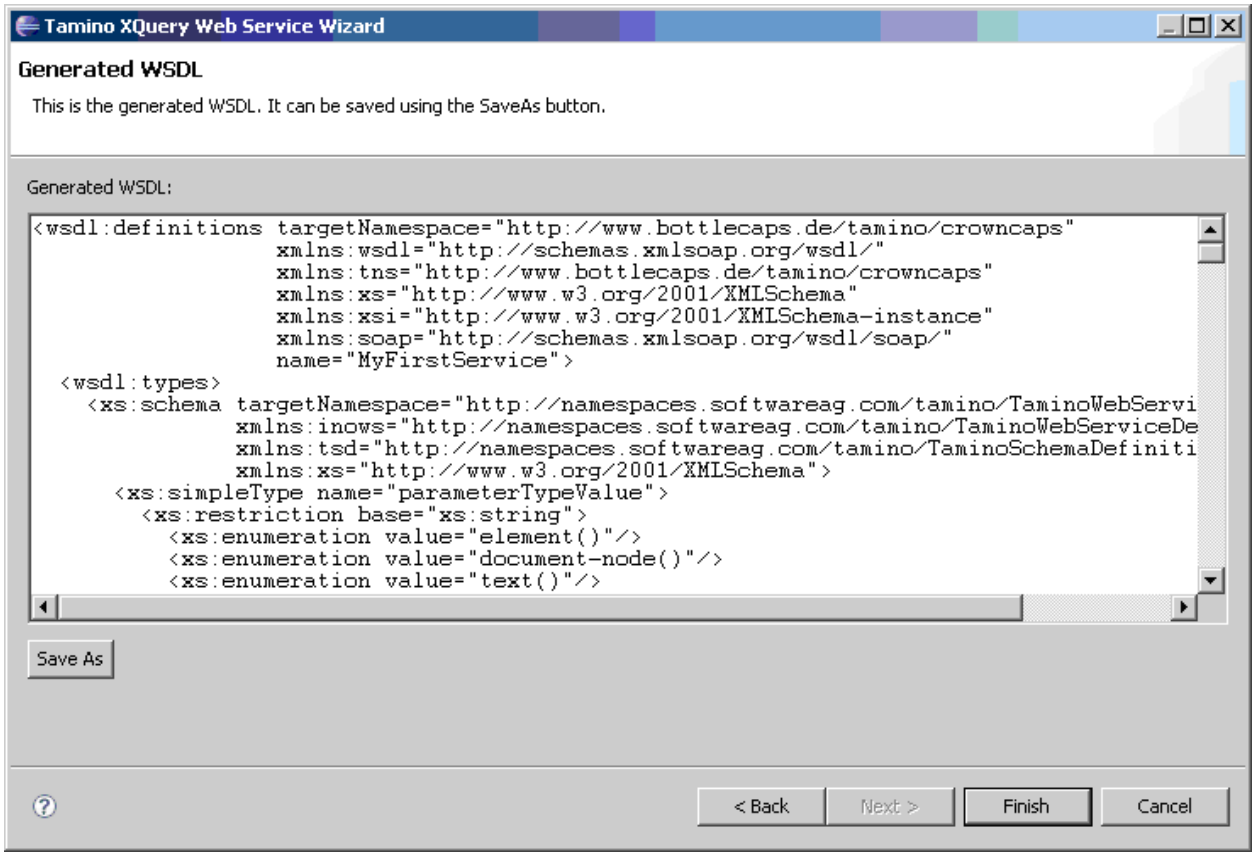
In the following screenshot, you see that a module has been selected; the wizard now shows all the functions that are included in the selected module, and displays a checkbox for each function. Check (tick) each function that should be included in the XQuery web service.



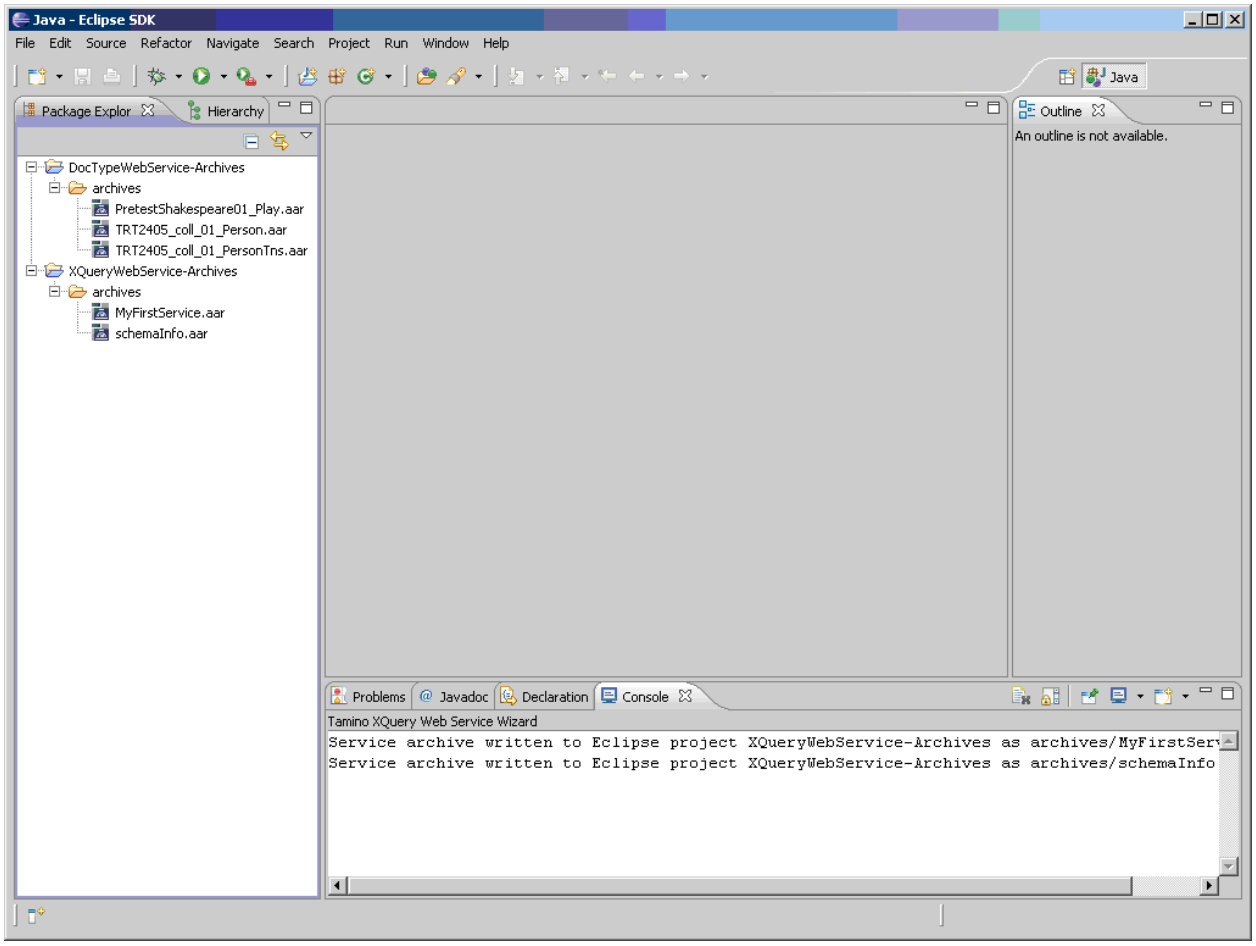
The following screenshot shows that several functions have been selected. Also, the user has overwritten the default name for the web service by the name *MyFirstService*.



The next step depends on the setting on the preference **Display generated WSDL**, which was **specified above**. If this option was selected (i.e. checked/ticked), the next screen displays the generated WSDL and you can use the **Save As** button to save it to the file system, as shown in the example below. If this option was not selected, this step is skipped.

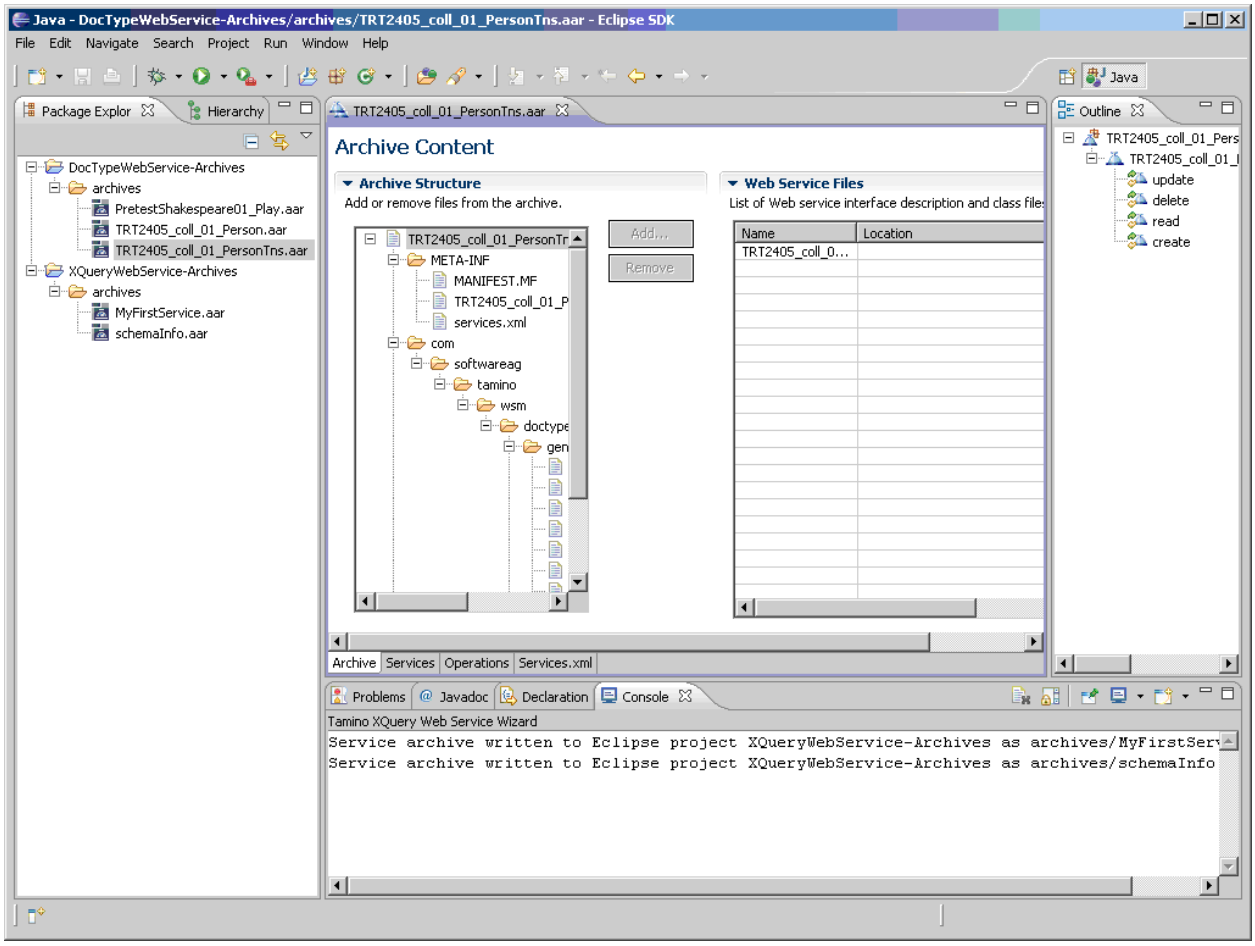


Finally, pressing the **Finish** button generates the web service archive, either at a selected file system location or in the Eclipse project as shown below, according to the preferences **described above**.



Using the Generated Web Service

After the web service archive has been generated and written to the Eclipse archive, you can use the Software AG Web Services Stack (WSS) to process it further. In the Eclipse package explorer, double-click on the archive; this opens the Web Services Stack editor, where you can set some service properties, as shown below:



For further details, please refer to the documentation of the Software AG Web Services Stack.

You can also use the Web Services Stack to deploy the generated web service, and to register it in CentraSite. These functions can be accessed from the archive context menu (right mouse button).

8 CRUD Usage of a Tamino Doctype as a Web Service

- Setting Preferences 94
- Using the Wizard 65
- Using the Generated Web Service 72

Tamino provides a plugin wizard that supports the creation, reading, updating and deleting (often referred to as “CRUD”) of XML documents via a web service. The wizard generates a doctype web service archive that can subsequently be processed either manually or by means of the Software AG Web Services Stack (WSS).

The following topics are discussed in this chapter:

Setting Preferences

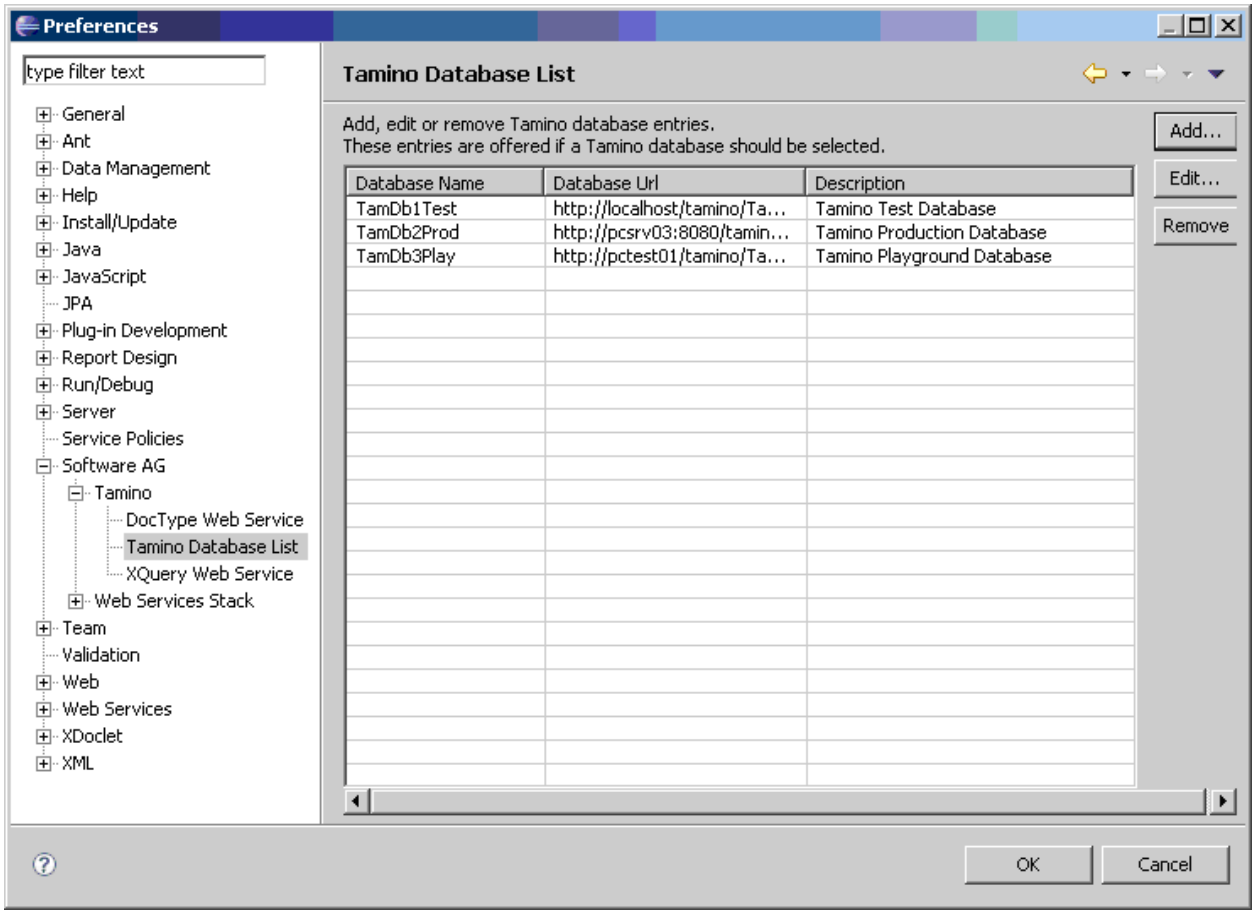
Before using the wizard to generate a web service archive, you can set preferences that specify its behavior. There are two sets of preferences:

- **Database selection**;
- **DocType Web Service preferences**.

Database Selection

In a later step, the doctype web service wizard prompts you to select a Tamino database from a selection list. In order to populate the selection list, proceed as follows.

In Eclipse, choose **Window** > **Preferences**. Then, in the navigator tree, select **Software AG** > **Tamino** > **Tamino Database List**. The list of Tamino databases available for selection is displayed, as shown in the following screenshot:

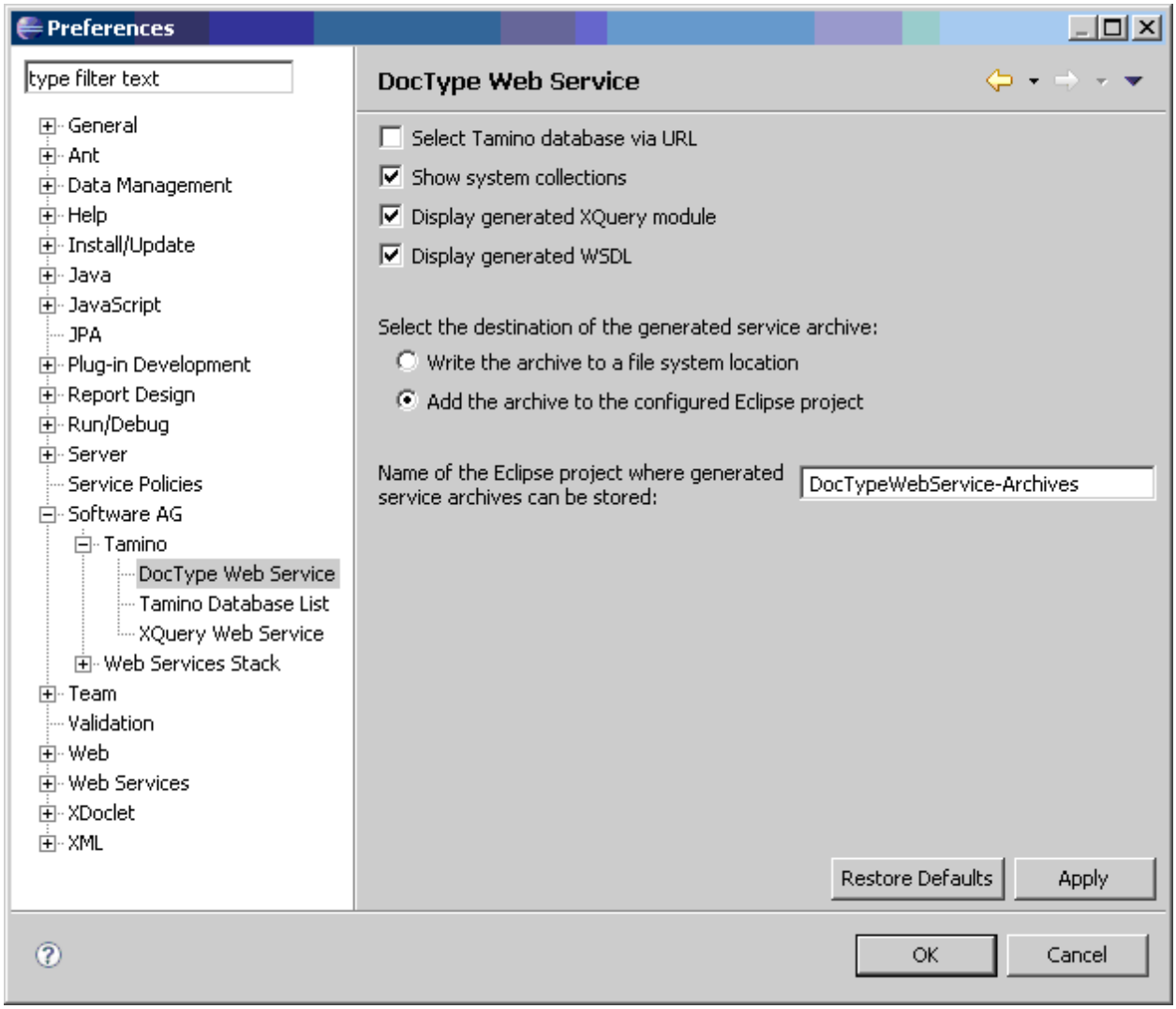


You can now use the **Add...**, **Edit...** and **Remove** functions to modify the selection list as required.

Note: In order to generate a web service successfully, the database selection list must contain at least one entry.

DocType Web Service Preferences

In Eclipse, choose **Window** > **Preferences**. Then, in the navigator tree, select **Software AG** > **Tamino** > **DocType Web Service**. The list of preferences for the Tamino doctype web service wizard is displayed, as shown in the following screenshot:



The following paragraphs describe the available preferences.

Select Tamino database via URL

This switch specifies whether the database selection list that appears in subsequent wizard screens should display the names of the databases or their URLs.

Default: The database selection list displays the databases by name.

Show system collections

This switch specifies whether Tamino system collections are included in the list of collections from which you can select.

Default: Do not include the Tamino system collections.

Display generated XQuery module

This switch specifies whether the XQuery module that is generated by the wizard should be shown to the user; if this switch is selected, the user is also given the opportunity of saving the XQuery module in the file system.

Default: Do not display the XQuery module.

Display generated WSDL

This switch specifies whether the WSDL file that is generated by the wizard should be shown to the user; if this switch is selected, the user is also given the opportunity of saving the WSDL file in the file system.

Default: Do not display the WSDL file.

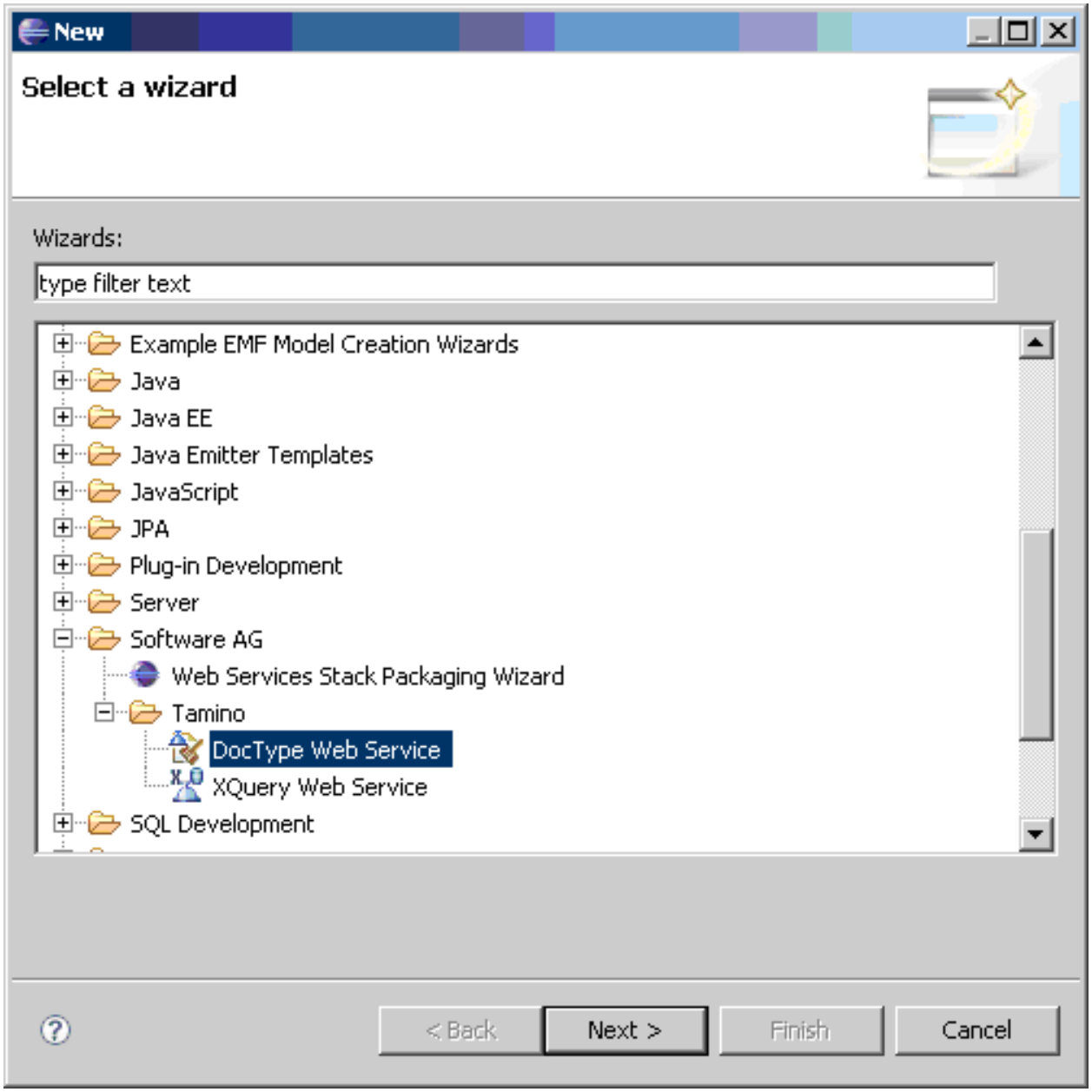
Select the destination of the generated service archive

This specifies whether the generated service archive should be stored to a user-specified file system location, or in an Eclipse project (where it can be further processed by means of the Software AG Web Services Stack). You can also specify the name of the desired Eclipse project here.

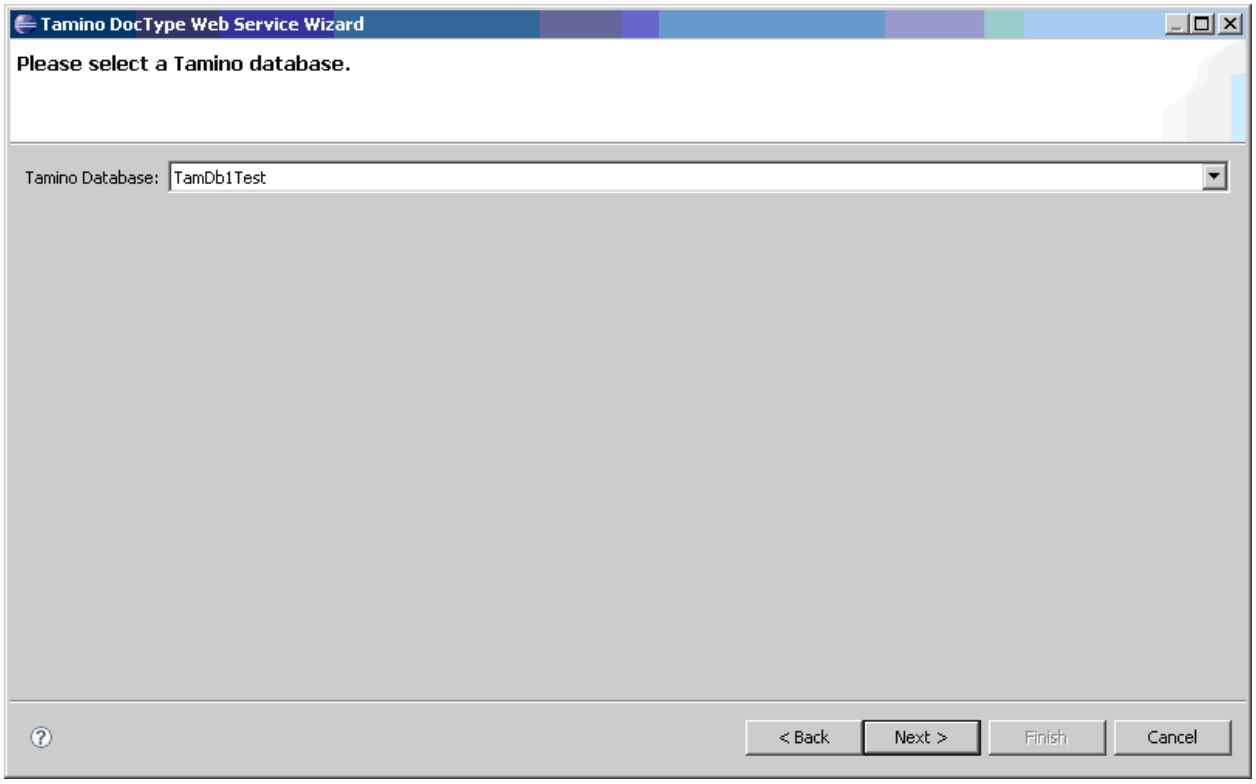
Default: Store the generated service archive in an Eclipse project named *DocTypeWebService-Archives*.

Using the Wizard

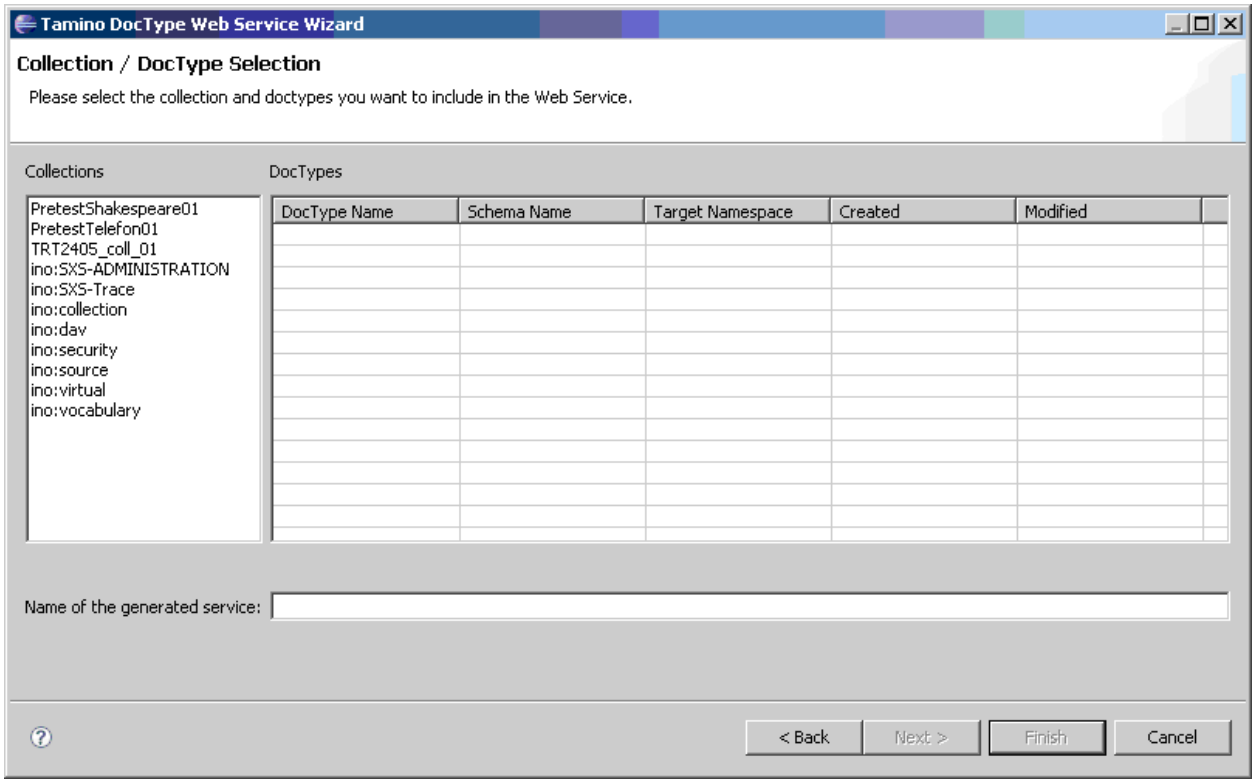
The DocType Web Service wizard plugin runs in your Tamino Eclipse environment. To start the wizard, choose **File** > **New** > **Other**; then, in the tree view, select **Software AG** > **Tamino** > **DocType Web Service**, as shown in the following screenshot:



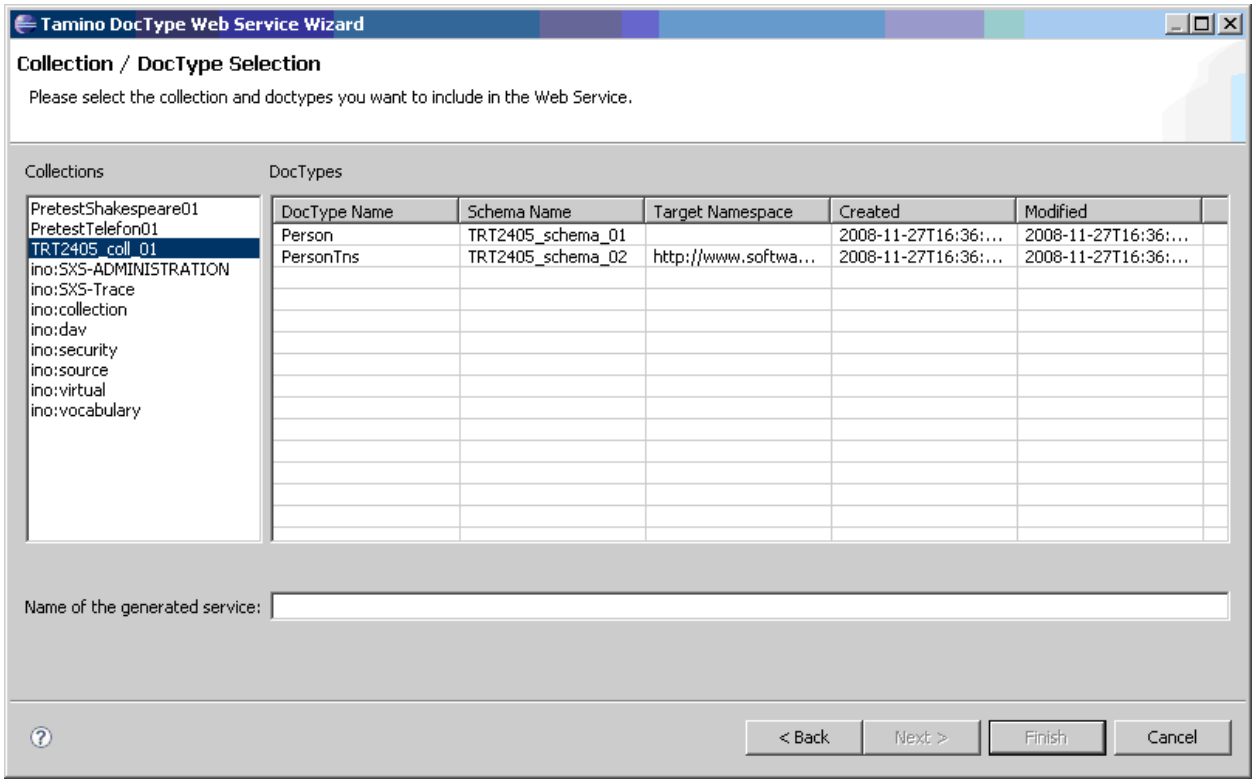
Choose **Next**. The DocType Web Service wizard starts, and displays a page from which you can select a Tamino database from the pulldown database list, which was constructed using the preferences as **described above**:



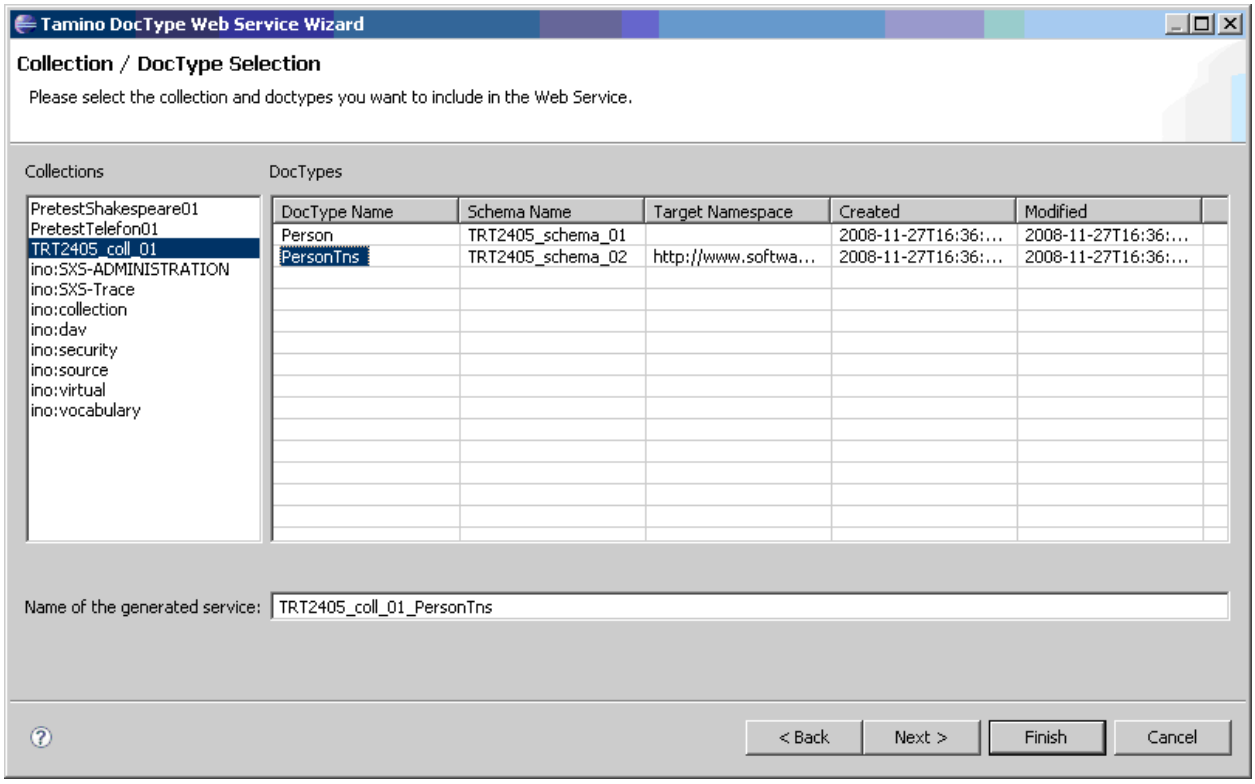
The next screen lists the collections contained in the selected Tamino database that have a schema, as shown in the example below:



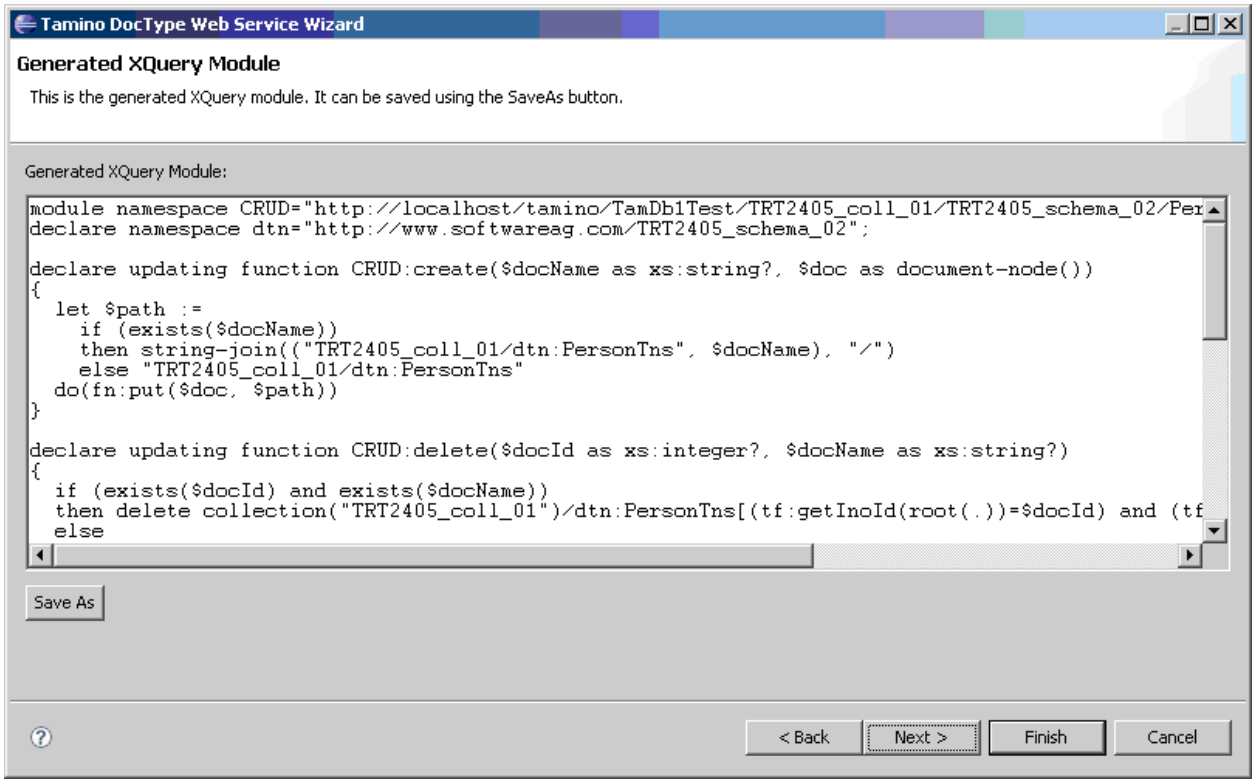
Select the desired collection. The wizard now lists all the doctypes that are contained in the selected collection:

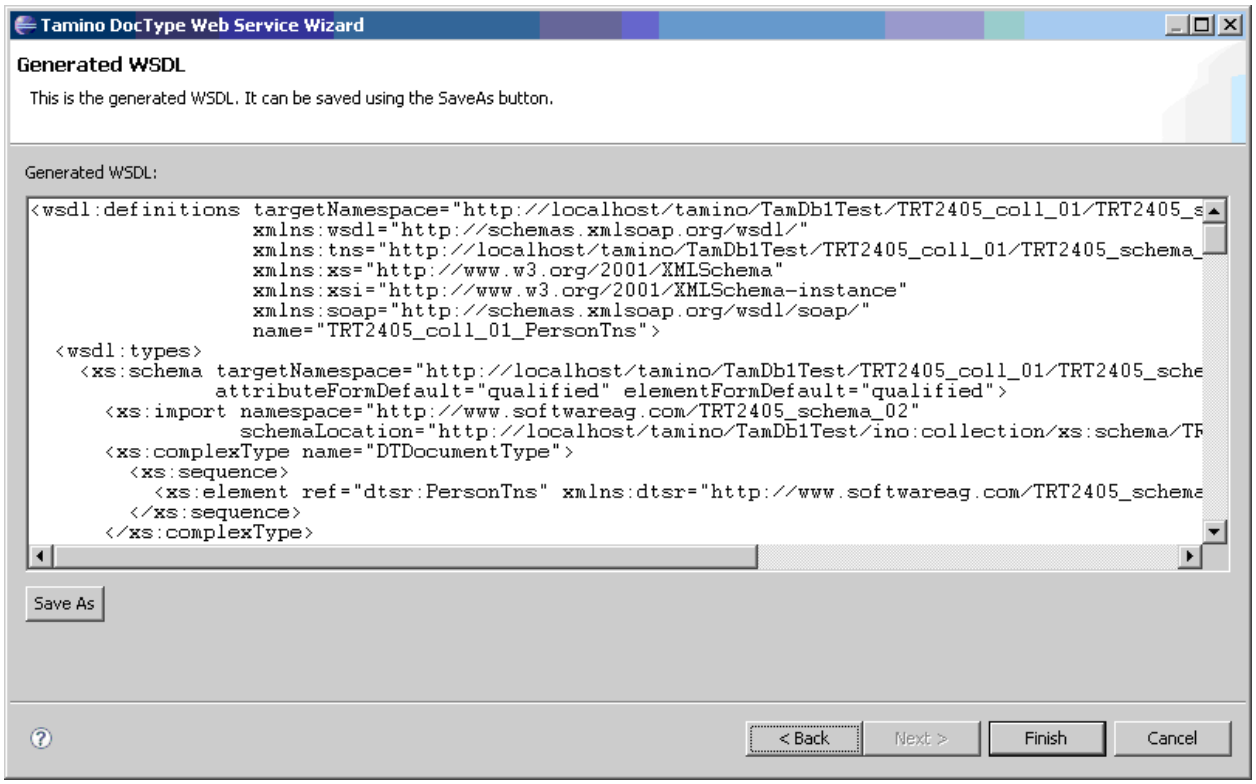


Select the desired doctype. The wizard suggests a default name for the web service that will be generated, but you can overwrite it:

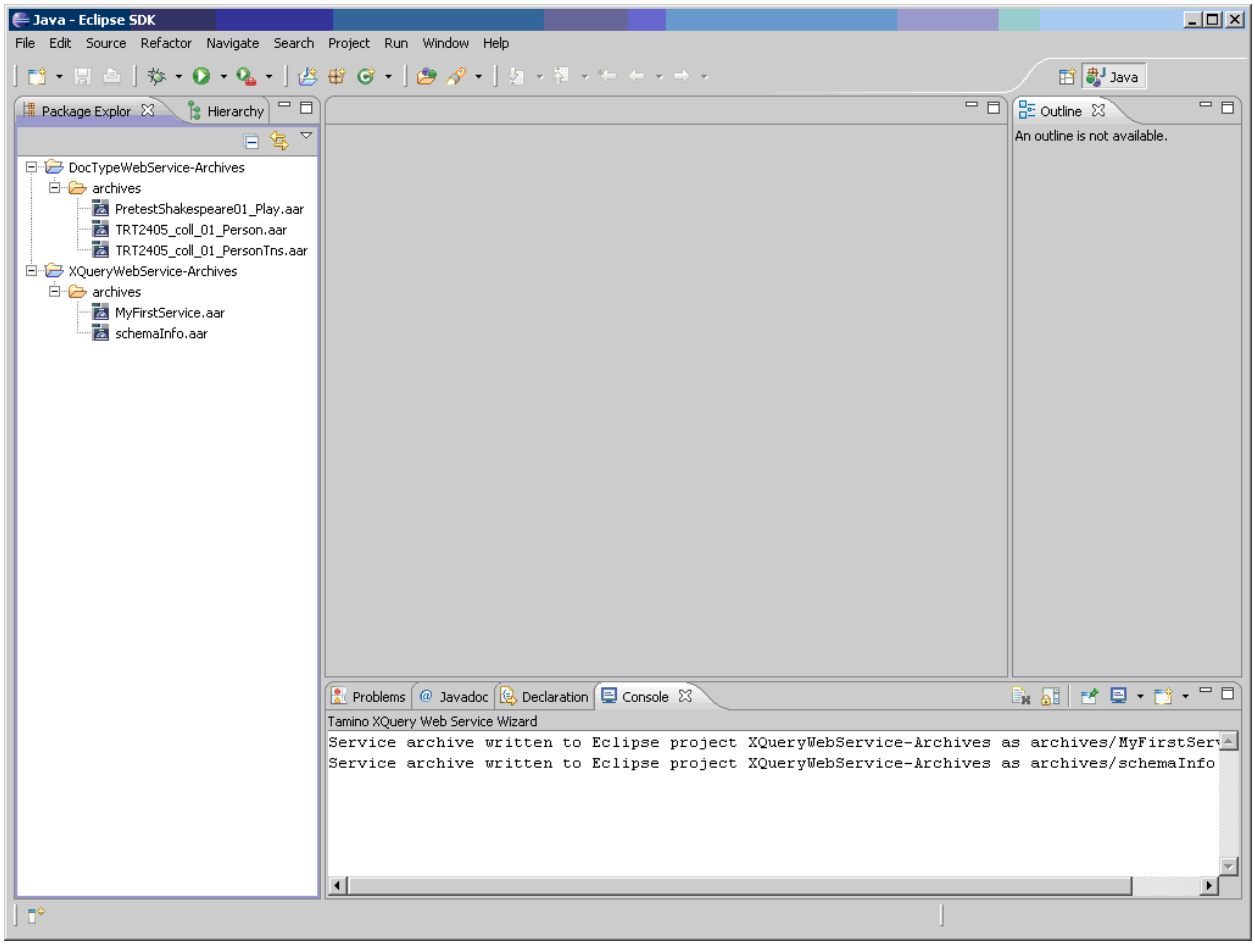


The next steps depend on the settings on the preferences **Display generated XQuery module** and **Display generated WSDL**, which were **specified above**. If one or both of these options were selected (i.e. checked/ticked), the next screens display the generated XQuery module and/or the generated WSDL. You can use the **Save As** button to save the generated XQuery module and/or the generated WSDL to the file system, as shown in the example below.


Tamino DocType Web Service Wizard
Generated XQuery Module
This is the generated XQuery module. It can be saved using the SaveAs button.
Generated XQuery Module:
module namespace CRUD="http://localhost/tamino/TamDb1Test/TRT2405_coll_01/TRT2405_schema_02/Per
declare namespace dtn="http://www.softwareag.com/TRT2405_schema_02";
declare updating function CRUD:create($docName as xs:string?, $doc as document-node())
{
let $path :=
if (exists($docName))
then string-join(("TRT2405_coll_01/dtn:PersonTns", $docName), "/")
else "TRT2405_coll_01/dtn:PersonTns"
do(fn:put($doc, $path))
}
declare updating function CRUD:delete($docId as xs:integer?, $docName as xs:string?)
{
if (exists($docId) and exists($docName))
then delete collection("TRT2405_coll_01")/dtn:PersonTns[(tf:getInoId(root(.))=$docId) and (tf
else
Save As
< Back
Next >
Finish
Cancel

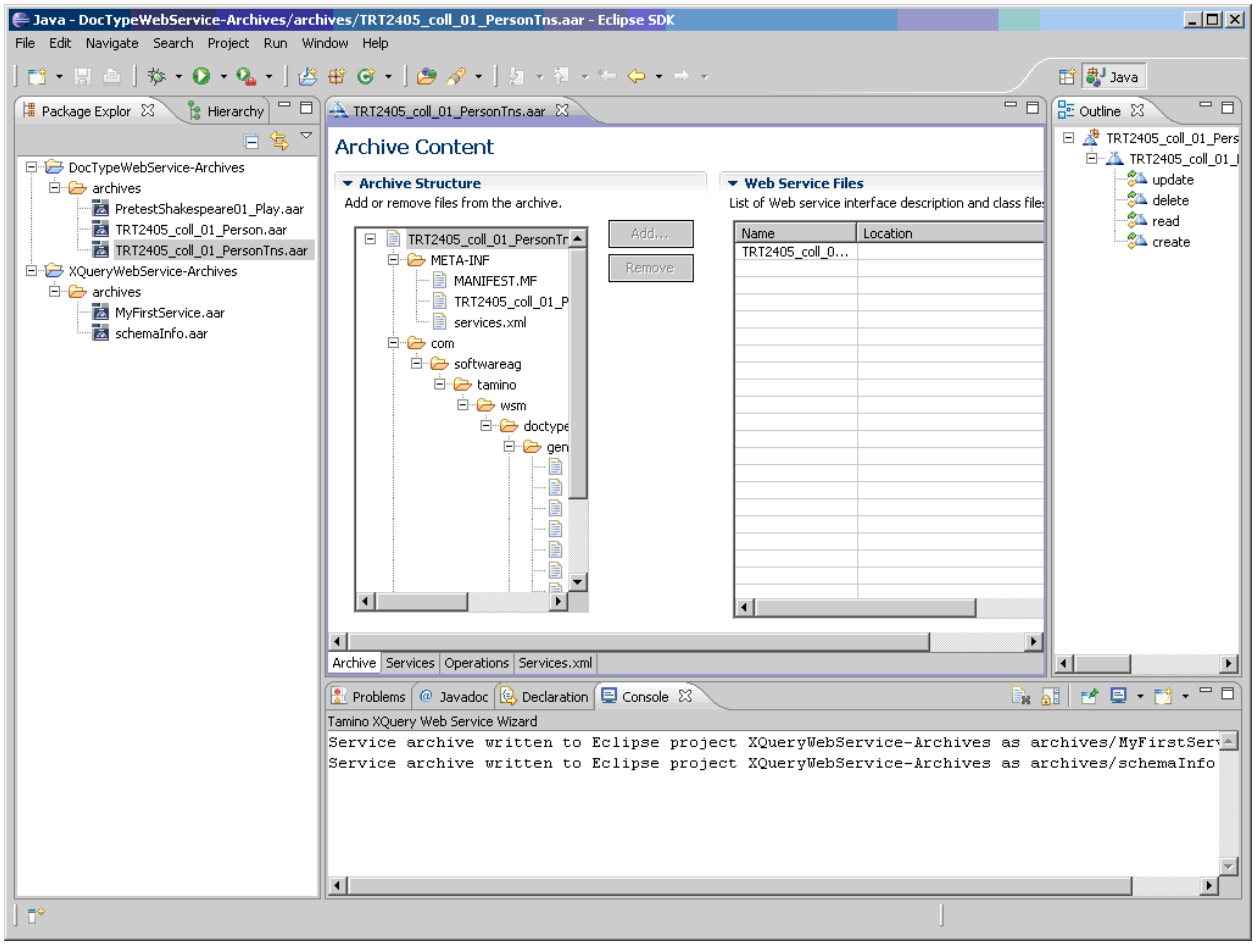

Tamino DocType Web Service Wizard
Generated WSDL
This is the generated WSDL. It can be saved using the SaveAs button.
Generated WSDL:
<wsdl:definitions targetNamespace="http://localhost/tamino/TamDb1Test/TRT2405_coll_01/TRT2405_s
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://localhost/tamino/TamDb1Test/TRT2405_coll_01/TRT2405_schema_
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
name="TRT2405_coll_01_PersonTns">
<wsdl:types>
<xs:schema targetNamespace="http://localhost/tamino/TamDb1Test/TRT2405_coll_01/TRT2405_sche
attributeFormDefault="qualified" elementFormDefault="qualified">
<xs:import namespace="http://www.softwareag.com/TRT2405_schema_02"
schemaLocation="http://localhost/tamino/TamDb1Test/ino:collection/xs:schema/TR
<xs:complexType name="DTDocumentType">
<xs:sequence>
<xs:element ref="dtsr:PersonTns" xmlns:dtsr="http://www.softwareag.com/TRT2405_schema
</xs:sequence>
</xs:complexType>
Save As
< Back
Next >
Finish
Cancel

Finally, pressing the **Finish** button adds the generated XQuery module to the Tamino database, and generates the web service archive, either at a selected file system location or in the Eclipse project as shown below, according to the preferences **described above**.



Using the Generated Web Service

After the web service archive has been generated and written to the Eclipse archive, you can use the Software AG Web Services Stack (WSS) to process it further. In the Eclipse package explorer, double-click on the archive; this opens the Web Services Stack editor, where you can set some service properties, as shown below:



For further details, please refer to the documentation of the Software AG Web Services Stack.

You can also use the Web Services Stack to deploy the generated web service, and to register it in CentraSite. These functions can be accessed from the archive context menu (right mouse button).

9 Advanced Usage

- Namespaces 76
- User-Defined Functions 77
- Defining and Using Modules 78
- Serializing Query Results 80
- Collations 82

This chapter covers some advanced XQuery usage. This functionality is partly tied to using Tamino query pragmas that affect the processing of the query. With Tamino query pragmas you can import modules and add declarations for namespaces and collations as well as functions and external variables. They have the syntactic form `{?piname?}`, which is why they were previously called XQuery processing instructions. This chapter discusses some of these concepts:

Another important area of advanced XQuery usage is related to performance questions and optimizing your Tamino XQuery programs. You can find more information about this topic in the *Performance Guide*.

Namespaces

A namespace declaration defines a namespace prefix and associates it with a valid namespace URI. This namespace declaration is valid throughout the query expression. For example, you can define your own namespace:

```
declare namespace dotcom="http://company.dot.com/namespaces/corporate"
for $a in input()/bib/book
return
<dotcom:bookshelf>
 { $a }
</dotcom:bookshelf>
```

It is an error (INOXQE 6356) to redefine a namespace:

```
declare namespace dotcom="http://company.dot.com/namespaces/corporate"
declare namespace dotcom="http://company.dot.com/new-namespaces/corporate"
for $a in input()/bib/book
return
<dotcom:bookshelf>
 { $a }
</dotcom:bookshelf>
```

But you can use two different namespaces at the same time:

```
declare namespace dotcom="http://company.dot.com/namespaces/corporate"
declare namespace tf="http://namespaces.softwareag.com/tamino/TaminoFunction"
for    $a in input()/bib/book
where  tf:containsText($a/title, "UNIX")
return
<dotcom:bookshelf>
 { $a }
</dotcom:bookshelf>
```

You can also declare default namespaces that can be applied to elements or attributes alike.

```
declare default element namespace "http://company.dot.com/namespaces/corporate"
for $a in input()/bib/book
return
<bookshelf>
 { $a }
</bookshelf>
```

The default element namespace applies to all unqualified element names, in both expressions and constructors, unless a different default namespace is defined in an inner scope, as shown in the following example:

```
declare default element namespace "X"
declare namespace y="Y"
let $x := <x1>{element x2{<x3 xmlns="Y">{element x4{}}</x3>}}</x1>
return $x/x2/y:x3/y:x4
```

In this example, the default element namespace applies to all occurrences of `x1` and `x2`; however, it does not apply to `x3` and `x4`, because their default namespace is defined by the inner specification `xmlns="Y"`.

The default element namespace does not apply to attribute names.

There is no default attribute namespace.

There is a default function namespace. It applies to function names in function definitions and function calls.

Tip: You need not declare a namespace in the prolog if you use one of the predeclared namespaces. You can find the list of predeclared namespaces in the description of `NamespaceDecl` in the *XQuery Reference Guide*

User-Defined Functions

In addition to the large set of XQuery functions as defined by the W3C and those that are specific to Tamino, you can write your own functions. User-defined functions consist of a function header representing its signature and the function body containing an XQuery expression. The header consists of an identifier, a list of parameters and corresponding type information, and the type of the return value. Type information is optional: if you omit it, `item*` will be assumed. Note that the set of allowed types is restricted to all simple types and some common XQuery types (the reference description of FunctionDecl contains a complete list).

The function identifier must be a QName together with a non-empty namespace prefix. To avoid introducing a new namespace for locally used functions, that is functions used in the current module, XQuery provides an implicit namespace to which the prefix `local` is bound. This is an example for a function that uses this local namespace:

```
declare function local:siblings($a as element()) as element()*
{
 for    $x in $a/../*
 where  not($x is $a)
 return $x
}

let    $a := <a> <a1>foo</a1> <a2>bar</a2> </a>
return local:siblings($a/a1)
```

It takes an element node as argument and returns all its sibling element nodes. Following the function declaration it is directly used in a query body resulting in the node `<a2>bar</a2>`.

User-defined functions may be recursive, that is a function can contain a call to itself:

```
declare function local:depth($e as node()) as xs:integer
{
 if (not($e/*))
 then
  1
 else
  max(for $c in $e/* return local:depth($c)) + 1
}
```

This function computes the depth of a node by calling itself for its children nodes.

To avoid running out of memory during the execution of a user-defined function, the XQuery processor restricts the amount of memory that can be used by a single query processing thread during query execution. The amount of memory can be modified by an XQuery pragma. Similarly, you can avoid a stack overflow by restricting the call stack of the function. The following pragma sets the available memory to 100 MB and the maximum call stack depth to 1000 (default value is 256):

```
{?execution memory="100" call-stack-depth="1000"?}
```

Note: See also the Performance Guide for information about optimizing user-defined functions using inlining techniques.

Defining and Using Modules

In XQuery, code can be organized into modules. There are two principal kinds of modules: a *main module* that contains the query body, and *library modules* that you can import into other library modules or into the main module. For example, you can put the functions defined in the last section into a module of its own:

```
module namespace tree="http://www.examples.com/tree"

declare function tree:depth($e as node()) as xs:integer
{
 if (not($e/*))
 then
  1
 else
  max(for $c in $e/* return tree:depth($c)) + 1
}

declare function tree:siblings($a as element()) as element()*
{
 for    $x in $a/../*
 where  not($x is $a)
 return $x
}
```

Before you can use the functions defined in a library module, you must store the module at a special place to make it directly accessible to the XQuery processor. In Tamino, modules are stored in the system collection `ino:source`. From a module definition, some meta data is derived to provide access to module properties that are relevant for efficient lookup of modules and user-defined functions. The meta data also simplifies querying module properties in order to generate interface descriptions such as WSDL definitions.

Modules are stored as non-XML data in the `ino:module` doctype of the `ino:source` collection. This way you can define modules using the `_process` command and remove them with `_delete`. Alternatively, you can use the Tamino Interactive Interface.

Note that a library module must not contain variable declarations. Also, Tamino query pragmas which are directives to the query processor are not allowed.

To use a module, you must import it with an import statement. Consider using the library of tree functions:

```
import module namespace tree="http://www.examples.com/tree"

let    $a := <a> <a1>foo</a1> <a2>bar</a2> </a>
return tree:siblings($a/a1)
```

This import statement imports the module with the target namespace *http://www.examples.com/tree*. Note that you cannot import the same module more than once.

To check for existing modules, you can query the `ino:source` collection in the following way to retrieve the non-XML data of all modules of a given namespace:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
root(
 for $a in collection("ino:source")/ino:module
 where $a/@ino:targetNamespace="http://www.examples.com/tree"
 return $a
)
```

The query functions defined in an XQuery module may also be used for defining a computed index. For details on computed indexes please refer to Performance Guide > Advanced Indexes.

The XQuery Tool delivered with Tamino provides convenient support for maintaining XQuery modules.

Serializing Query Results

When Tamino executes a query, it handles the query in terms of its data model: an XML document is treated as a tree with a document root and different types of nodes. The query result is then *serialized* and wrapped into a Tamino response document. You can determine the serialization by using a Tamino query pragma, or XQuery processing instruction, which always has the form `{?piname?}`. Like a regular XML processing instruction, it can have additional information. This section discusses the available serializations:

- Suppressing the Response Wrapper
- Setting the MIME Media Type
- Defining a Custom Output Handler

Suppressing the Response Wrapper

Normally, when Tamino executes a query, it returns the result using a response wrapper. This is the result of the very first query in this manual:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <ino:response xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
    xmlns:xql="http://metalab.unc.edu/xql/">
  - <xq:query xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
      <![CDATA[ <fact>This section contains {3 + 6} examples.</fact>  ]]>
    </xq:query>
  - <ino:message ino:returnvalue="0">
      <ino:messageline>XQuery Request processing</ino:messageline>
    </ino:message>
  - <xq:result xmlns:xq="http://namespaces.softwareag.com/tamino/XQuery/result">
      <fact>This section contains 9 examples.</fact>
    </xq:result>
  - <ino:message ino:returnvalue="0">
      <ino:messageline>XQuery Request processed</ino:messageline>
    </ino:message>
  </ino:response>
```

You can, however, suppress the response wrapper so that you get the bare result by using the serialization pragma and choosing the method "xml":

```
{?serialization method="xml"?}
<fact> This section contains { 3 + 6 } examples.</fact>
```

The output is then much shorter and is restricted to the query result as such:

```
<fact>This section contains 9 examples.</fact>
```

Setting the MIME Media Type

Using `{?serialization method="xml"?}` always implies that "text/xml" is used as media type for the response document. You can change this by adding `media-type`. The following query returns a literal SVG document which has the media type "image/svg+xml":

```
{?serialization method="xml" media-type="image/svg+xml"?}
<svg width="200" height="200">
  <circle cx="100" cy="50" r="40" stroke="black" stroke-width="3" fill="red"/>
</svg>
```

This delivers the SVG document with the appropriate MIME media type which is then set in the HTTP response header `contentType`. It also means that the normal response wrapper is not used.

Defining a Custom Output Handler

You can define your own method of delivering the response document by using a server extension (SXS) as output handler. This output handler will then act in place of the internal Tamino XQuery processor. In general you use:

```
{?serialization method="<outputHandler>"
  parameter="<outputHandlerParameter>"?}
```

For example, this allows using the XSLT server extension to transform query results on-the-fly:

```
{?serialization method="XSLT.transform"
  parameter="stylesheets" parameter="stylesheets/patientRecords.xsl"?}
<report>
  { for    $a in collection('Hospital')/patient
    return <patient>{ $a/name, $a/sex, $a/born, $a/address }</patient>
  }
</report>
```

The `method` attribute identifies the output handler. The two parameters specify the XSL stylesheet to be used and the query. Tamino then passes the query result, the list of patients, to this output handler which transforms it using the stylesheet *patientRecords.xsl*.

Please note that you can also use `media-type` in the serialization pragma for an output handler.

Collations

For certain operations on strings such as comparison or sort operations collations are used. They inherently determine the result of a comparison to accommodate for special regional purposes or linguistic needs. If a collation is not defined, Tamino uses Unicode codepoints for performing these operations. There are two ways to use collations: In the schema you can add collation information per element by using `tsd:collation`. In addition, you can assign a default collation in the query prolog such as:

```
declare default collation "collation?language=fr"
```

The string literal is the *collation URI* which is interpreted as relative URL with the base URI *http://www.softwareag.com/tamino*. In this example, the collation defined for the French language is used.

You can use the Tamino-specific XQuery function `tf:getCollation` to deploy collations as in the following query:

```
for $i in input()/patient/name
where compare($i/surname, 'Müller', tf:getCollation($i/surname)) = 0
return $i
```

This query returns those `name` elements whose `surname` child element has a contents that is compared equal to the string "Müller" using the collation information defined in the Tamino schema for `surname` elements.

Please see also the section *Collations* in the *Tamino XML Schema User Guide* for details about collation definitions in schemata.

10 Text Retrieval

- Simple Text Search 86
- Context Operations 87
- Highlighting Retrieval Results 88
- Phonetic Searches 91
- Stemming 92
- Rules for Searches Using Phonetic Values and Stemming 94
- Thesaurus 96
- Pattern Matching 98

Tamino text retrieval operations consider text as sequences of words. A word is a sequence of characters that is delimited by characters such as whitespace or punctuation characters. The process of analyzing text and determining words and delimiters is called tokenization. Depending on the language, there are different tokenizers available in Tamino. The default, so-called "white space-separated" tokenizer is suitable for most letter-based languages. However, ideographic languages such as Chinese, Japanese or Korean (often referred together as CJK languages) have a totally different concept of segmenting a sequence of ideographs into “word” tokens. Tamino offers a special tokenizer for Japanese.

Tokens are categorized into character classes such as "character", "delimiter" or "number". You can find detailed information about this topic in the section *Implications Concerning Text Retrieval* in *Unicode and Text Retrieval*.

For the purpose of doing text retrieval in XQuery, it is sufficient to think of full text in terms of words and delimiters. In this context, “words” are referred to as word tokens, regardless of whether a tokenizer analyzed a series of letters or of ideographs.

This chapter covers the following topics:

Simple Text Search

There are two possibilities to search nodes in XML documents for some text: You can either use an exact search or search for word tokens. Let us look for divers among the patients in our hospital database. We know that there is someone who is a “professional diver”:

```
for    $a in input()/patient
where  $a/occupation = "Professional Diver"
return <divers level="professional">{ $a/name }</divers>
```

And Tamino will return Mr. Atkins as the only representative of the hospital's professional divers. To find all divers, professional or not, you are inclined to ask:

```
for    $a in input()/patient
where  $a/occupation = "diver"
return <divers>{ $a/name }</divers>
```

The result is an empty sequence, so we lost even the professional diver. The `where` clause contains an equality expression that is true if the text contents of the node read exactly "diver". Since the `occupation` element in Mr. Atkin's reads "Professional Diver", the result is false and no divers are returned. However, using the function `tf:containsText` you can search for the word "diver" somewhere in the node `occupation` and without looking at the case:

```
for    $a in input()/patient
where  tf:containsText($a/occupation, "diver")
return <divers>{ $a/name }</divers>
```

Now Tamino returns Mr. Atkins in the diver list, since the tokenizer identifies "Diver" as a word, delimited on the left by a space character. After applying lower case as standard tokenizer rule the token "diver" is found and the function `tf:containsText` returns `true`. Please note that this function looks for word tokens, so using `tf:containsText($a/occupation, "dive")` would yield `false`, since "dive" is not a token that can be found in any of the `occupation` nodes. Similarly, a query using `tf:containsText($a/occupation, "professional diver")` returns `true`, since the two word tokens are found in that order in the `occupation` node. However, `tf:containsText($a/occupation, "diver professional")` returns `false`: Although both tokens are found, they are not in the specified order.

Context Operations

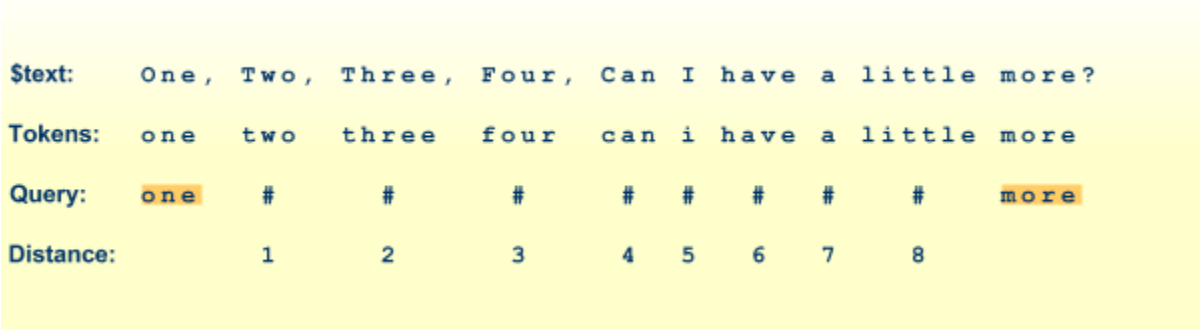
With context operations you can search for expressions that consist of one or more words which do not necessarily follow after one another. For example, you can search for variants of the expression "text retrieval" such as "retrieval of text" or in "text storage and retrieval". In Tamino, there are functions that let you specify the following context operations (here, "#" stands for one token):

- **Maximum word distance**
  "text # # retrieval" matches "text retrieval" and "text storage and retrieval"
- **Word order**
  If word order is not significant, "text retrieval" matches "text retrieval" and "retrieval of text"

The functions `tf:containsAdjacentText` and `tf:containsNearText` both expect a maximum word distance as second argument. Consider the following query:

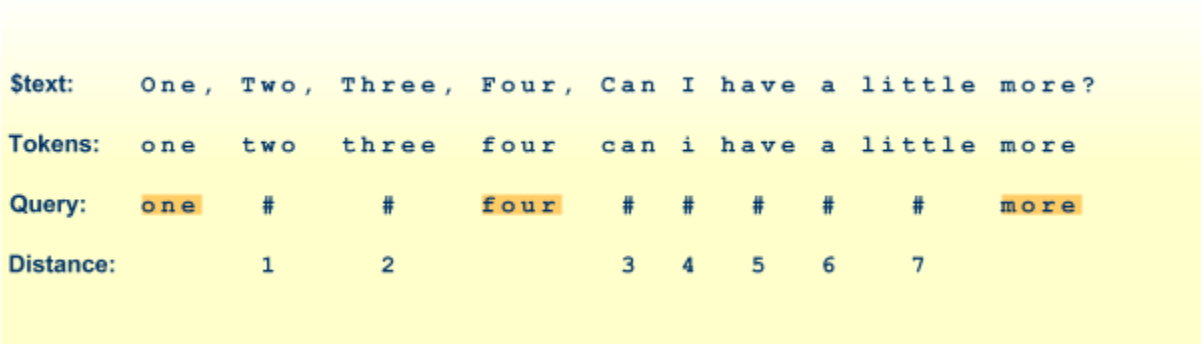
```
let $text := text{"One, Two, Three, Four, Can I have a little more?"}
return tf:containsAdjacentText($text, 9, "one", "more")
```

This graphics shows you the search string, its tokens and how the query matches the search string:



The function `tf:containsAdjacentText` returns `true` if the tokens "one" and "more" are found in that order within a distance of less than nine tokens. Since there are eight unmatched tokens, the function returns `true` for the above query. Let us slightly change the query expression:

```
let    $text := text{"One, Two, Three, Four, Can I have a little more?"}
return tf:containsAdjacentText($text, 9, "one", "four", "more")
```



The function returns `true` if all the tokens "one", "four" and "more" are found in that order within a distance of less than nine tokens, not including any matched tokens in between such as "four". Since there are seven unmatched tokens, the function returns `true` for the above query.

Generally, if you use a distance value of "1", it means that the tokens follow immediately one after another. It follows that `tf:containsAdjacentText($mynode, 1, "search", "text")` is equivalent to `tf:containsText($mynode, "search text")`.

While `tf:containsAdjacentText` respects the word order, the function `tf:containsNearText` does not. The following query using `tf:containsNearText` returns `true`, using `tf:containsAdjacentText` it would return `false`:

```
let    $text := text{"One, Two, Three, Four, Can I have a little more?"}
return tf:containsNearText($text, 2, "four", "one", "two")
```

Highlighting Retrieval Results

When retrieving information from some text corpus, it is desirable to visualize the information found. In Tamino XQuery, you can do so by “highlighting” retrieval results. Consider the following query from the Tamino XQuery reference guide which searches in all `review` nodes for the word "discussion":

```
for    $a in input()/reviews/entry
let    $ref := tf:createTextReference($a/review, "discussion")
where  $ref
return tf:highlight($a, $ref, "REV_DISC")
```

A Tamino client application could highlight the results as follows:

```
<entry>
  <title>Data on the Web</title>
  <price>34.95</price>
  <review>A very good discussion of semi-structured database systems and XML.</review>
</entry>
<entry>
  <title>Advanced Programming in the Unix environment</title>
  <price>65.95</price>
  <review>A clear and detailed discussion of UNIX programming.</review>
</entry>
```

You can see from the query expression that there are two steps involved when highlighting retrieval results:

1. Generate a reference description to the locations that should be highlighted.
2. Apply highlighting to the document according to the locations.

Generating Reference Descriptions

A reference description is necessary for highlighting later on. It consists of at least the following global information:

- collection
- document type (“doctype”)
- document number
- node id

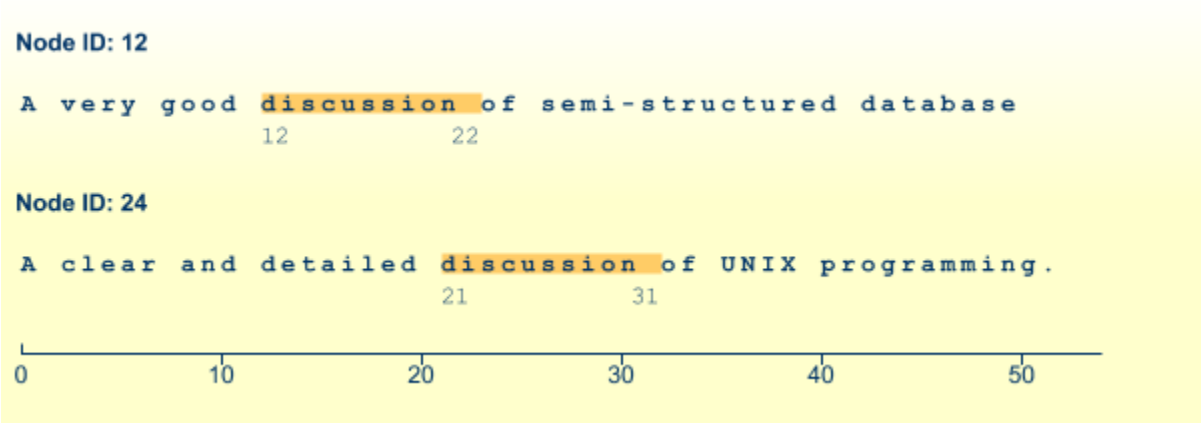
References to text within a node further need to describe the locations of start and end points, the *range*. You can create reference descriptions by using the following functions: `tf:createAdjacentTextReference`, `tf:createNearTextReference`, and `tf:createTextReference`. These functions work exactly like their `tf:contains`*`XXX`*`Text` counterparts, only that they return reference descriptions of text ranges instead of a Boolean value. There is another function `tf:createNodeReference` to create reference descriptions of nodes. Let us have a look at the reference descriptions that will be used in our example query:

```
for $a in input()/reviews/entry
return tf:createTextReference($a/review, "discussion")
```

Tamino will return these two object descriptions in its response:

```
<ino:object ino:collection="XMP" ino:docid="1" ino:doctype="reviews" ino:doctypeid="1" ino:end="22" ino:nodeid="12"
  ino:start="12" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
<ino:object ino:collection="XMP" ino:docid="1" ino:doctype="reviews" ino:doctypeid="1" ino:end="31" ino:nodeid="24"
  ino:start="21" xmlns:ino="http://namespaces.softwareag.com/tamino/response2" />
```

The figure below shows you the text ranges for which reference descriptions have been created:



Highlighting Documents

Any location, for which a reference description exists, can be highlighted by using the function `tf:highlight`. The query performing the highlighting is repeated here for your convenience:

```
for $a in input()/reviews/entry
let $ref := tf:createTextReference($a/review, "discussion")
where $ref
return tf:highlight($a, $ref, "REV_DISC")
```

As arguments, the function `tf:highlight` requires a node, a previously-generated reference description and a marker string. Tamino uses processing instructions (PIs) to indicate the start and end of the range to be highlighted so that a client application receiving the Tamino response document can parse and process them. The marker string is used as the so-called PI target. You can easily identify the highlighted text ranges in the response document for the above query:

```
<entry>
  <title>Data on the Web</title>
  <price>34.95</price>
  <review>A very good <?REV_DISC + 1 ?>discussion<?REV_DISC - 1 ?> of semi-structured ↩
database systems and XML.</review>
</entry>
<entry>
  <title>Advanced Programming in the Unix environment</title>
  <price>65.95</price>
  <review>A clear and detailed <?REV_DISC + 2 ?>discussion<?REV_DISC - 2 ?> of UNIX ↩
programming.</review>
</entry>
```

The start and end of the highlighted text range are indicated by the plus and minus signs in the PI. Furthermore, highlighted ranges are numbered. This is also true when highlighting complete nodes:

```
for    $a in input()/bib
let    $ref:= tf:createNodeReference($a/book[1])
return tf:highlight($a, $ref, "FIRST")
```

The resulting document is:

```
<book year="1994"><?FIRST + 1?>
   <title>TCP/IP Illustrated</title>
   <author><last>Stevens</last><first>W.</first></author>
   <publisher>Addison-Wesley</publisher>
   <price>65.95</price>
 <?FIRST - 1?></book>
```

Phonetic Searches

A “phonetic search” allows you to search for words that are phonetically equivalent. Linguistically speaking, you are looking for *homophones*. For example, in English the letters c and s as in "city" and "song" are both pronounced [s], and the words "eight" and "ate" are both pronounced [eɪt] (in square brackets you see the phonetic transcription using the IPA, the International Phonetic Alphabet).

There are many areas where this facility proves valuable: Imagine a patient database having lots of patients with German names. Now you want to retrieve all patients with the name "Maier". In German, it is an everyday surname, just as "Kim", "Smith", and "Andersson" are for Korean, English and Swedish respectively. You can use a query performing a simple text search such as:

```
for    $a in input()/patient
where  $a/name = "Maier"
return $a/name
```

or you can use `tf:containsText`:

```
for    $a in input()/patient
where  tf:containsText($a/name, "Maier")
return $a/name
```

Unfortunately, there are at least four variants of the name which are all pronounced [maɪə]: "Maier", "Mayer", "Meier", and "Meyer". Instead of constructing long Boolean expressions that try to cover all existing homophones, you can use `tf:phonetic` in the scope of one of the search functions:

```
for    $a in input()/patient
where  tf:containsText($a/name, tf:phonetic("Maier"))
return $a/name
```

This query will return all patients whose names sound like [maɪə].

Tamino uses a set of rules to determine phonetic equivalency. There are rules pre-defined, which are explained in more detail in the reference documentation to tf:phonetic. These rules are suitable for German and English, but you can create your own set of rules. See the section **Rules for Searches Using Phonetic Values and Stemming** below.

Note: You can not use `tf:phonetic` standalone, but only in the context of one of the following functions: `tf:containsText`, `tf:containsAdjacentText`, `tf:containsNearText`, `tf:createAdjacentTextReference`, `tf:createNearTextReference`, `tf:createTextReference`. This means that the result of calling this function in another context is unspecified and might change in a future Tamino version.

Stemming

A corpus with text in an inflecting language such as English or German often contains words in inflected forms: nouns are declined and verbs are conjugated: "The nightingales were singing in the trees." If you want to search for all occurrences of the verb "to sing" or of the nouns "nightingale" and "tree", you need to know how these words are inflected and derived. One method is to reduce any inflected form to its word *stem*. It is the stem to which morphemes are attached to construct a certain grammatical form: So "were" + "**sing**" + "-ing" indicates the past continuous tense of the verb "to sing".

In Tamino, you can use the function `tf:stem` to retrieve occurrences of all word forms belonging to the same word stem. Similarly to `tf:phonetic`, it works only in the scope of one of the search functions:

```
let $text :=
 <chapter>
  <para>Die Bank eröffnete drei neue Filialen im Verlauf der letzten fünf ↩
Jahre.</para>
  <para>Ermüdet von dem Spaziergang setzte sich die alte Dame erleichtert auf die
    gepflegt wirkende Bank mitten im Stadtpark.</para>
  <para>Die aktuelle Bilanz der Bank zeigt einen Anstieg der liquiden Mittel im
    Vergleich zum Vorjahresquartal.</para>
  </chapter>
for    $a in $text//para
let    $check :=
       for    $value in ("Geld", "Bilanz", "Filiale", "monetär", "Aktie")
       return tf:containsNearText($a, 10, tf:stem($value), tf:stem("Bank"))
where  count($check[. eq true()]) > 0
return $a
```

This returns all `para` elements whose text contains at least one word which is related to a specific reading of the German word "Bank". The resulting document contains the first and the third `para` element, but not the second, since it does not contain any of the words defined in the sequence `("Geld", "Bilanz", "Filiale", "monetär", "Aktie")` in a distance of less than ten words from "Bank".

Tamino uses a set of rules to determine whether a word token belongs to some stem. There is a pre-defined rule set that works reasonably well for German. However, you can create your own set of rules. See the section **Rules for Searches Using Phonetic Values and Stemming** below. You can reach the pre-defined rules set using the following query:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
collection("ino:vocabulary")/ino:stemrules
```

Note: Do not use `tf:stem` standalone; use it only in the context of one of the following functions: `tf:containsText`, `tf:containsAdjacentText`, `tf:containsNearText`, `tf:createAdjacentTextReference`, `tf:createNearTextReference`, `tf:createTextReference`. The result of calling this function in any other context is unspecified and might change in a future Tamino version.

Rules for Searches Using Phonetic Values and Stemming

For queries that involve phonetics and stemming, Tamino internally uses the same mechanism, implemented as a finite-state machine, that rewrites the function argument according to a set of rules. These rules are described by XML schemas stored in the collection `ino:vocabulary` and have the names PHONRULES and STEMRULES:

This is the schema for PHONRULES:

```
<xs:complexType>
  <xs:sequence>
    <xs:element name="phonrule" minOccurs="1" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="phonStage" type="xs:integer" use="required" />
        <xs:attribute name="phonType" type="xs:string" use="required" />
        <xs:attribute name="phonReqs" type="xs:integer" use="required" />
        <xs:attribute name="phonMinChars" type="xs:integer" use="required" />
        <xs:attribute name="phonChars" type="xs:string" use="required" />
        <xs:attribute name="phonReplaceChars" type="xs:string" use="required" />
        <xs:attribute name="phonNextStage" type="xs:integer" use="required" />
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

This is the schema for STEMRULES:

```
<xs:complexType>
  <xs:sequence>
    <xs:element name="stemrule" minOccurs="1" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="stemStage" type="xs:integer" use="required" />
        <xs:attribute name="stemType" type="xs:string" use="required" />
        <xs:attribute name="stemReqs" type="xs:integer" use="required" />
        <xs:attribute name="stemMinChars" type="xs:integer" use="required" />
        <xs:attribute name="stemChars" type="xs:string" use="required" />
        <xs:attribute name="stemReplaceChars" type="xs:string" use="required" />
        <xs:attribute name="stemNextStage" type="xs:integer" use="required" />
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

So a set of rules consists of a series of `ino:phonrule` or `ino:stemrule` elements. The semantics of a rule are determined by its attributes, all of which are mandatory:

- **Location**
  The attributes `ino:phonType` and `ino:stemType` define the part of word affected by the rule: This is one of the values "SUFFIX", "INFIX" or "PREFIX"
- **Character Substitution**
  These attributes define the characters to look for and their replacement:
  - The attributes `ino:phonChars` and `ino:stemChars` define the sequence of characters to look for in a string
  - The attributes `ino:phonReplaceChars` and `ino:stemReplaceChars` define the substitution string.
- **State Machine Control**
  These attributes control the way the state machine is treating this rule:
  - The attributes `ino:phonStage` and `ino:stemStage` define the state in which the rule will be active. The attribute value is a number from 1 to *n*. In most of the rules there will be no state change.
  - The attributes `ino:phonNextStage` and `ino:stemNextStage` define the next state when the rule has fired. If the value is "0" the machine stops and the result is computed.
- **Restrictions**
  If a restriction is violated, the rule will not be executed.
  - The attributes `ino:phonMinChars` and `ino:stemMinChars` define the minimum length of the word fragment for the rule to be applied.
  - The attributes `ino:phonReqs` and `ino:stemReqs` define the minimum number of syllables in the word fragment for the rule to be applied. For `ino:phonReqs` this is practically always the value "0".

The order of the rules is significant. The following stem rules protect the substitution of "EAR":

```
<ino:stemrule ino:stemType='SUFFIX' ino:stemStage='2' ino:stemNextStage='0'
  ino:stemChars='EAR' ino:stemReplaceChars='EAR' ino:stemReqs='0' ↩
ino:stemMinChars='6' />
<ino:stemrule ino:stemType='SUFFIX' ino:stemStage='2' ino:stemNextStage='0'
  ino:stemChars='AR'  ino:stemReplaceChars=''    ino:stemReqs='0' ↩
ino:stemMinChars='5' />
```

Thesaurus

A thesaurus is a special kind of dictionary that is ordered by topic or semantic relationships. A regular dictionary uses a lexicographic order: for example, letter-based languages use the language's alphabet; ideographic languages use the base signs and the number of strokes. In contrast, a thesaurus is ordered by meaning: it helps you find words or phrases for general ideas. Semantic relationships let you explore words along two directions: horizontally by looking up variants with the same context of meaning (e.g., synonyms, antonyms) or vertically by finding broader, superordinate terms (hypernyms), and narrower, subordinate terms (hyponyms). In Tamino, this adds another dimension of text retrieval functionality: now you can retrieve contents not only by using the graphemic representation or syntactic variants of the search term, but also by using its semantic properties.

Tamino supports the most important aspects of a thesaurus: synonyms, hypernyms and hyponyms. There is no pre-defined thesaurus, so you can specify one tailored to the special vocabulary of your Tamino application scenario. You can define one or more thesauri in a single database. The collection `ino:vocabulary` holds thesaurus entries as `term` elements, each of which is assigned to a single thesaurus using the attribute `ino:thesaurus`. A `term` element can contain the following elements:

`termName`
: defines the name of the thesaurus entry (mandatory)

`synonym`
: defines a term which is synonymous to `termName`

`broaderTerm`
: defines a term which is superordinate to `termName` (hypernym)

`narrowerTerm`
: defines a term which is subordinate to `termName` (hyponym)

Example

To create a sample thesaurus with words having to do with animals, load the following data into the collection `ino:vocabulary` of an existing database. Please refer to the section *Loading Data into Tamino* for more information about loading data into Tamino.

```
<?xml version="1.0"?>
<term ino:thesaurus="animals"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2">
  <ino:termName>dog</ino:termName>
  <ino:synonym>canine</ino:synonym>
  <ino:synonym>pooch</ino:synonym>
  <ino:synonym>doggie</ino:synonym>
  <ino:synonym>bow-wow</ino:synonym>
  <ino:synonym>puppy-dog</ino:synonym>
```

```
  <ino:synonym>perp</ino:synonym>
  <ino:synonym>whelp</ino:synonym>
  <ino:broaderTerm>carnivore</ino:broaderTerm>
</term>
```

The example file dog.xml

```
<?xml version="1.0"?>
<term ino:thesaurus="animals"
      xmlns:ino="http://namespaces.softwareag.com/tamino/response2">
  <ino:termName>carnivore</ino:termName>
  <ino:broaderTerm>mammal</ino:broaderTerm>
</term>
```

The example file carnivore.xml

These two files establish the thesaurus "animals" for the given database. In the vertical direction the following hierarchy can be derived: A dog is a carnivore; a carnivore is a mammal. This is because the `ino:broaderTerm` contents ("carnivore") in the thesaurus entry for "dog" matches the `ino:termName` of another thesaurus entry, namely "carnivore".

Furthermore synonyms are defined for the entry "dog" that denote a dog using colloquial language, biological terms, pet names, etc.

Examples

Return all paragraphs in the specified document that contain a synonym of "dog":

```
let $doc := <doc>
  <p>Have you seen the large dog around the corner?</p>
  <p>On the farm nearby, a checkered whelp was playing on the ground with some ↩
cats.</p>
  <p>Also, some horses could be seen in the stable.</p>
 </doc>
for    $p in $doc/p
where  tf:containsText($p,tf:synonym("dog"))
return $p
```

As a result, the first two paragraphs are returned. Strictly speaking, only "whelp" is defined as a proper synonym, but Tamino follows the intuitive assumption that you also expect the term itself to be part of the result set. This holds for other thesaurus functions as well.

The following query returns all superordinate terms of "dog", for which you use the Tamino function `tf:broaderTerms`:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for    $p in collection("ino:vocabulary")/ino:term
where  tf:containsText($p/ino:termName,tf:broaderTerms("dog"))
return $p/ino:termName
```

Here, the two `ino:termName` instances for dog and carnivore are returned.

Pattern Matching

You can conveniently perform text searches with the help of search patterns. Tamino's text retrieval system allows for efficient queries using special characters that match one or more characters in a word. In Tamino XQuery the following functions support text search using pattern matching:

`tf:containsText`
`tf:containsAdjacentText`
`tf:containsNearText`
`tf:createTextReference`
`tf:createAdjacentTextReference`
`tf:createNearTextReference`

The tokenizer that is being used determines the pattern matching facilities that are available. The following table gives an overview of the characters that have a special meaning when used in one of the above functions:

Character	Tokenizer Availability	Effect
? (*maskcard*)	white space-separated	match a single character in a word
* (*wildcard*)	CJK white space-separated	match zero or more characters in a word
\ (*escapecard*)	white space-separated	cancel the special meaning of the following character

The default tokenizer ("white space-separated") supports all types of special characters, whereas the Japanese tokenizer only supports wildcard characters. The section **Wildcard Characters** provides details about the peculiarities when using the Japanese tokenizer.

The **table above** shows the default settings. However, with the white space-separated tokenizer you can use a different character for each of these special characters. If, for example, your data frequently uses the asterisk sign as a regular character, it is more convenient to redefine the wildcard character instead of having to escape the asterisks using the escapecard character every time they occur. See the section **Customizing Special Character Settings** for information on how to change these settings. The discussion here assumes the default settings as defined in `ino:transliteration` and, if not stated otherwise, the usage of the standard white space-separated tokenizer.

In the context of pattern matching, a word consists of a non-empty sequence of characters: for example, a wildcard character (default: asterisk) matches zero or more characters *in a word*, so that a single "*" represents a single word. If the search string contains more than one word, such as in the expression `tf:containsText($node, "word1 word2")` then it is treated as `tf:containsAdjacentText($node, 1, "word1", "word2")`.

The following sections contain more information about searching with any of these special characters and how to change the default setting:

- The Maskcard Character
- Wildcard Characters
- The Escapecard Character
- Customizing Special Character Settings

The Maskcard Character

The maskcard character, which by default is a question mark "?", stands for a single character in a word. A pattern `theat??` thus matches `theatre` as well as `theater`, but not `theatrical` since `??` only match `ri` and the rest (`cal`) is not matched.

Consider the following query:

```
let $text := text{"one two three four five six seven eight nine ten"}
return
(tf:containsText($text, "??"), tf:containsText($text, "t??"), tf:containsText($text, ↩
"two?three"))
```

The query returns a sequence of three items, each being a Boolean value that indicates whether the specified pattern matches the contents of the text node in `$text`. Attempting to match the first pattern `??` yields "false", since there are no numerals with only two letters. The second pattern `t??` matches all three-letter numerals beginning with `t`, namely `two` and `ten`, so "true" is returned. The last pattern fails again, although the pattern `two?three` seems to match the value "two three". However, since pattern matching is always performed on the basis of a word, the match does not succeed: the string "two three" is treated as two words delimited by the space character in between.

Note: Introducing the question mark as a maskcard character also has the effect that it is no longer classified as a delimiter character in the default transliteration.

Wildcard Characters

In contrast to the maskcard character, which matches exactly one character, the wildcard character matches zero or more characters in a word. By default, the wildcard character is an asterisk "*".

Consider the following query:

```
let    $text := text{"one, two"}
return tf:containsAdjacentText($text, 1, "one", "*", "*")
```

This query returns `false`, since `tf:containsAdjacentText` expects two word tokens adjacent to "one".

If you use the default tokenizer, i.e. the white space-separated tokenizer, then the wildcard character is always the asterisk "*" (Unicode value `U+002A`).

Using the Japanese Tokenizer

If you use the Japanese tokenizer, all of the following characters are recognized as wildcard characters:

Unicode Name	Code Value
ASTERISK	U+002A
ARABIC FIVE POINTED STAR	U+066D
ASTERISK OPERATOR	U+2217
HEAVY ASTERISK	U+2731
SMALL ASTERISK	U+FE61
FULL WIDTH ASTERISK	U+FF0A

Note: In contrast to the standard white space-separated tokenizer, this definition of wildcard characters is fixed and cannot be changed.

The Japanese tokenizer does not support wildcard characters in the middle of a word, since there are no explicit delimiter characters. So `"心*患"` will be treated as `"心*" adj "患"`.

The example queries below focus on the contents of the `patient/submitted/diagnosis` nodes to show the effect of performing search operations with or without wildcard characters on segmentation of Japanese words.

1.

Contents		心臓に問題がある		
Translation		problems with the heart		
Segmentation	Word Tokens	心臓	ある	問題
	Translation	heart (physical)	has	problem

2.

Contents		心臓麻痺	
Translation		heart attack	
Segmentation	Word Tokens	心臓	麻痺
	Translation	heart	paralysis

3.

Contents		心臓疾患	
Translation		heart disease	
Segmentation	Word Tokens	心臓	疾患
	Translation	heart	disease

4.

Contents		心臓血管疾患	
Translation		cardiovascular disease	
Segmentation	Word Tokens	心臓血管	疾患
	Translation	heart angio (cardiovascular)	disease

The following example queries all use the same query skeleton:

```
for    $a in input()/patient/submitted/diagnosis
where  <function-call>
return $a
```

XQuery Function Call	Matching Samples
`tf:containsText($a, "心臓")`	1, 2, 3
`tf:containsText($a, "心臓病")`	—
`tf:containsText($a, '心*")`	1, 2, 3, 4
`tf:containsText($a, '心臓*")`	1, 2, 3
`tf:containsText($a, "心臓疾患")`	3
`tf:containsAdjacentText($a, 1, "心臓", "疾患")`	
`tf:containsNearText($a, 1, "心臓", "問題")`	—
`tf:containsNearText($a, 1, "心*", "*患")`	3, 4

The Escapecard Character

With the help of the escapecard character you can negate any special meaning of the following single character. By default, the escapecard character is the backslash character "\". Use it if you want to look for any of the maskcard, escapecard or wildcard characters as literal characters.

Example 1: Escaping an Asterisk

```
let $text:= text{"** End of code **"}
return tf:containsText($text, "\*\*")
```

Here, the match succeeds if there is any two-letter word in `$text` that consists only of asterisks. This is true for the first and last words in `$text`.

Example 2: Escaping a Backslash

The following checks the path separator character that is used in `$path` and returns the result as plain text, ordered by platform:

```
{?serialization method="text" media-type="text/plain"?}
let $path := text{"C:\Program Files\Software AG\Tamino"}
return
( "Path Separators Used&#x0A;",
  "DOS/Windows&#x09;: ", tf:containsText($path, "*\\*"), "&#x0A;",
  "UNIX&#x09;: ", tf:containsText($path, "*/*"), "&#x0A;",
  "MacOS&#x09;: ", tf:containsText($path, "*:*")
)
```

In the pattern, the path separator character must be enclosed by the regular wildcard character, since it would not form a word on its own. Provided that the path separator character is defined as a regular character, the query reports for each platform whether its standard path separator character is used, although this example is certainly not a bullet-proof method. However, if the

path separator is not defined as a regular character, it will be interpreted as a delimiter. Using the default settings, the pattern in function call `tf:containsText($path, "*\\*")` will thus be interpreted as if `"* *"` were used, since the escapecard character tries to mask an invalid character and the text retrieval system uses a delimiter instead. This would yield "true", since there are two occurrences of two adjacent words separated by a space character ("Program Files" and "Software AG"). In cases like these you should ensure that the transliteration is appropriately defined according to what your application expects.

Note: In contrast to "*", the characters "\" and "?" are normally not classified as regular characters in the collection `ino:transliteration`. See the section Customizing Transliterations for instructions how to customize this special collection.

Customizing Special Character Settings

If you use the default tokenizer (white space-separated), you can customize the settings for the special characters used in pattern matching. They are declared as attributes to `ino:transliteration` in the special collection `ino:vocabulary`:

escapecard character	`ino:escapechar`
maskcard character	`ino:maskchar`
wildcard character	`ino:wildchar`

To change the value of any of these attributes, you can use a query similar to the following, which sets the value of the maskcard character to the default value:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
update insert attribute ino:maskchar {"?"}
into input()/ino:transliteration
```

You can check your changes with the following query:

```
declare namespace ino="http://namespaces.softwareag.com/tamino/response2"
for $a in input()/ino:transliteration/@*
return $a
```

Note that only attributes that have been modified are returned.

III Related Information

11 Related Information

- Internal Resources 108
- W3C Resources 108

The XQuery documentation focuses on XQuery as a language. However, as database queries are the principal means of communication with the database server, it might happen that you have a problem which arose because of a certain query. If you do not find the information in the XQuery documentation, you might have a look at the following sources of information, all of which also deal with aspects of querying a Tamino database:

Internal Resources

- Using the Tamino Interactive Interface to Query a Database with Tamino XQquery: This section describes how to query a database object using the Tamino Interactive Interface.
- Efficient Queries: XQuery: Here you will find some advice for speeding up your queries.
- Querying Using X-Machine Commands: This section explains how to directly query a database using HTTP and X-Machine command verbs.
- XML Schema Datatypes: This section explains the data types defined in the XML Schema specification which are used by XQuery.
- Storing Non-XML Objects in Tamino: This section explains how to represent and query objects that are in some other format, such as binary data.
- Character Encoding of XML Objects: This section discusses the character encoding of XML objects when querying them directly using HTTP and X-Machine command verbs.
- Migrating from X-Query to Tamino XQuery (in the section *Migrating Applications* of the *Migration Guide*): This section gives some advice for application developers concerning changes from the previous implementation called X-Query to Tamino XQuery.

W3C Resources

- The XQuery 1.0 language, as specified by the W3C, consists of the following documents:
  - **XQuery 1.0: An XML Query Language**
    The official XQuery recommendation of the W3C, which forms part of the basis of Tamino XQuery.
  - **XQuery 1.0 and XPath 2.0 Data Model (XDM)**
    The official XQuery recommendation of the W3C that describes the underlying data model used for XQuery 1.0 and XPath 2.0.
  - **XSLT 2.0 and XQuery 1.0 Serialization**
    The official XQuery recommendation of the W3C that describes how to serialize an instance of the data model into a sequence of octets.

- **XQuery 1.0 and XPath 2.0 Functions and Operators**
  The official XQuery recommendation of the W3C, which forms part of the basis of Tamino XQuery.

- **XML Path Language (XPath) 2.0**: The official recommendation of the W3C, which forms part of the basis of XQuery.
- **XML Query Use Cases**: Some use cases relevant for XQuery.
- Working Drafts for doing text retrieval with XQuery:
  - **XQuery and XPath Full Text 1.0 Requirements**
  - **XQuery and XPath Full Text 1.0 Use Cases**
- **Namespaces in XML 1.0**: This W3C recommendation defines basic language elements that are used in XPath.

Index

A

atomization, 30

C

character
  special
    escapecard, 102
    maskcard, 99
    wildcard, 100
collation
  query, 82

D

delete
  node
    in Tamino XQuery, 17, 39

I

insert
  node
    in Tamino XQuery, 18, 38

P

path expression
  in Tamino XQuery, 27
pattern matching, 98

R

rename
  node
    in Tamino XQuery, 18, 40
replace
  node
    in Tamino XQuery, 40

S

sequence
  in Tamino XQuery, 24
  singleton, 24
special character
  escapecard, 102
  maskcard, 99
  wildcard, 100

T

text retrieval
  in Tamino XQuery, 15
  tokenization, 86
tokenizer, 98
type promotion, 30

W

wildcard
  in pattern matching, 100

X

XQuery
  concepts
    atomization, 30
    calling through a web service, 49
    constructor, 8, 26
    CRUD usage of a doctype as a web service, 61
    expression, 24
    FLWOR expression, 9, 33
    path expression, 27
    sequence, 24
    sort, 12
    type, 29
    type promotion, 30
    update operations, 37
    valid documents on update, 42
  filter, 10, 35
  FLWU expression, 41
  join, 14
  namespace, 76
  path expressions
    axis, 27
    node test, 28
    predicate, 29
  retrieve object from database, 25
  serialize query result, 80
  text retrieval, 15
  update, 16
    delete, 17, 39
    insert, 18, 38
    rename, 18, 40
    replace, 40