XML Stream Guide

UNIFACE V8.2

10117048201-00
Revision 0
Jun 2001
XML

UNIFACE V8.2
XML Stream Guide

Revision 0

24-hour online customer support and learning

MyUNIFACE is an Internet-based support and learning environment which provides real-time access to a wealth of UNIFACE product and technical information. Features include online product documentation, technical tips and know-how, up-to-date platform availability, product fixes, course information, online training, and live communication with fellow developers. You can obtain full access privileges for MyUNIFACE by completing an online registration form (customer license information is required) at http://myuniface.com/.

For the latest version of the documentation always check the UNIFACE Library on the MyUNIFACE site.

Your suggestions and comments about UNIFACE documentation and course material are highly valued. Please send your reactions to:

Compuware Europe B.V.
Delivery Methods & Practices
P. O. Box 12933
1100 AX Amsterdam
The Netherlands

e-mail: DM&P-Hotline@nl.compuware.com
fax: +31 (0)20 311-6213

Contents

2 Modifications to the Repository for XML streams

3 UNIFACE support for XML and DTD syntax

4 XML transformations

5 Handling communication between components

6 Proc statements and functions

Preface

This guide describes three-tier development using XML stream parameters.

Audience

The guide is written for all UNIFACE developers and managers working in a multitier or component-based development paradigm.

This guide assumes some understanding of the XML standard issued by the World Wide Web Consortium.

How to use this guide

The guide describes three-tier development, the XML stream parameters, the UNIFACE DTD Editor, and the Proc statements and functions that handle XML streams.

Conventions

The XML standard, UNIFACE and component-based development use certain terms in slightly different ways.

Entity

In XML, an entity is a named character string defined in a DTD. This string can be used in an XML document by using escape characters combined with the name of the entity.

In UNIFACE, an entity is an object in the application model consisting of data and default code.

To distinguish between these two usages, the guide refers to *XML entities* and *UNIFACE entities*.

This guide also uses the term *predefined entities*, which is defined in the XML standard.

Attribute

In XML, an attribute is a property of an XML element. For example, a COUNTRY element might have a has_state attribute, as follows:

```
<COUNTRY has_state="no">Malta</COUNTRY>

<COUNTRY has_state="yes">United States of America</COUNTRY>
```

In component-based development, the term attribute refers to data that is exposed by a component.

This guide uses attribute exclusively in the sense intended by the XML standard.

This guide also refers to *processing information attributes*, which are XML attributes generated by UNIFACE to place state and validation information in XML streams.

Chapter 1 Three-tier development

1.1 Three-tier architecture

The three-tier software architecture is a strict and formal split of an application into distinct parts, each part performing a specific function. The tiers described in a three-tier architecture are the presentation tier, the business logic tier, and the data access tier.

The function of each tier is as follows:

- Presentation tier—this tier holds all the components responsible for the user interface. These components are typically UNIFACE Server Pages, forms, and reports, and non-UNIFACE components, such as JavaServer Pages or Microsoft's Active Server Pages.
- Business logic tier—this tier holds all the components that handle business rules and task-specific behavior. In UNIFACE, the business tier is composed of session services for the centralization of complex business rules affecting multiple entities. Optionally, this tier contains entity services for the centralization of simple business rules for single entities. In addition, services and 3GL components can be found here.

  Network and middleware access are encapsulated by the middleware drivers and the UNIFACE Router.
- Data access tier—this tier contains physical database structures captured in the relational business object model. UNIFACE ensures physical data access by encapsulating SQL in its DBMS drivers.

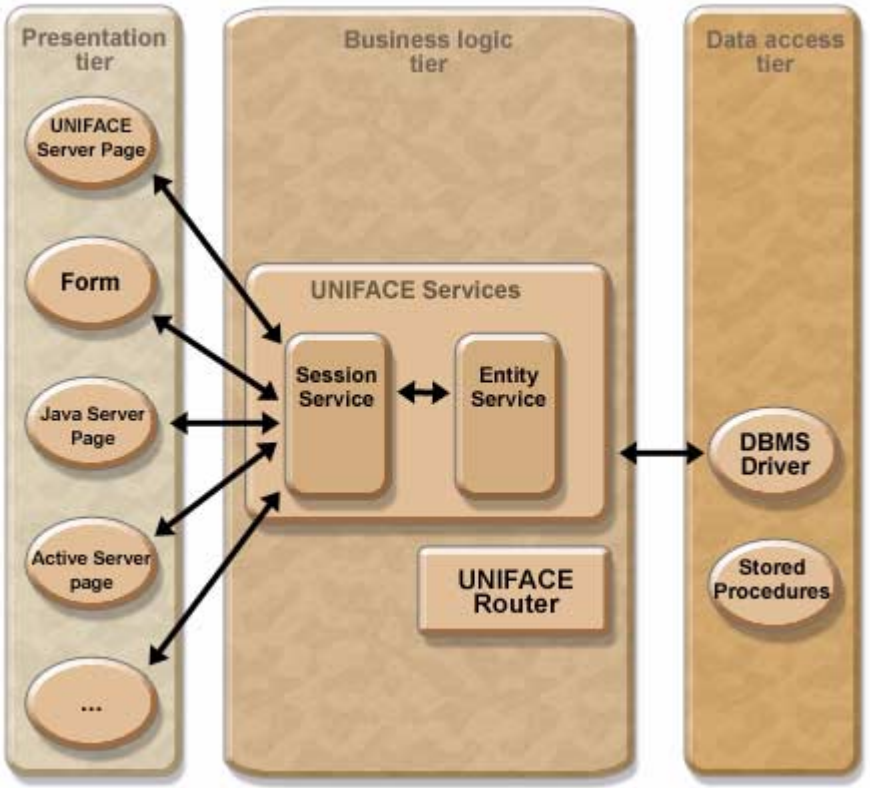
1.1.1 Advantages of the three-tier software architecture

The three-tier software architecture has advantages in the following areas:

- Scalability and deployment flexibility—component roles are specialized, improving maintainability, networking, and I/O overheads.
- Component roles are clearly defined within a three-tier framework. This provides a good basis for component-based development and re-usability.
- Stateless communication between components in the presentation tier and those in the business logic tier is ensured by means of XML.
- Infrastructure independence is enhanced by the use of a three-tier architecture. This is because presentation and data access—areas that are often infrastructure-dependent—are separated from the application's business logic.
- A specific set of skills is required for the development of each tier, so tiers can be developed independently of each other. For example, the thin presentation tier allows front-end experts to do their work

without being affected by developments taking place in the business logic tier.

Figure 1-1 Three-tier software architecture.



1.1.2 Requirements of three-tier development

UNIFACE fully supports the following requirements of three-tier application development:

- Component-based development
- Support for distributed architectures
- Support for disconnected record sets

Component-based development

Each application tier is component-based. This means that each tier is composed exclusively of replaceable components that communicate via well-defined signatures. It is therefore essential to construct components for each tier using CBD methodology.

UNIFACE fully supports component-based development.

Support for distributed architectures

Multitier applications are ideally deployable on *n*-server, *n*-client networks, and the presentation layer should be readily deployable across the Internet.

UNIFACE fully supports distributed architectures. UNIFACE introduces XML streams that enable transparent communication with Internet-based rich client components.

Support for disconnected record sets

Presentation layer components should not have a role in database I/O as this role is handled by the data layer.

This implies that data in the presentation layer does not have a connection to the database. Records in the presentation layer are therefore referred to as disconnected record sets. When data is returned to the business layer, there must be a mechanism for reconnecting the data to existing occurrences. For example, the business layer must be able to identify new occurrences, modified occurrences, and occurrences marked for deletion. Support for disconnected record sets is essential for fully separating the presentation tier from the business tier.

UNIFACE supports disconnected record sets by including processing information with XML streams. This processing information is available to any application or component that can parse XML streams. Additionally, the processing information is accessible by the Proc functions `$occcrc`, `$occstatus`, `$fieldproperties`, and `$occproperties`.

1.1.3 Presentation tier

The presentation tier is the logical group of components in an application that provide a user interface, such as UNIFACE Server Pages, forms, reports, as well as other non-UNIFACE components such as JavaServer pages and Microsoft's Active Server Pages.

Presentation tier components have the following characteristics:

- Allow users to interact with the application.
- Do not process data or handle business rules.
- Do not directly access databases or other storage media.

Validation and business logic

Declarative validation on the presentation tier in a three-tier application reduces network and server overhead, because it can ensure that substantially correct data is submitted to the business tier. Early validation of data on the presentation tier is also user-friendly.

However, procedural validation on the presentation tier is not recommended because the enforcement of business rules is centralized by the business tier. Furthermore, the capabilities of presentation components to enforce business rules can be limited. For example, browsers can choose to disable JavaScript code.

Validate carefully on the presentation tier for the following reasons:

- Combining business logic with user-interface issues complicates code and debugging and defeats the purpose of three-tier development.
- Distributing logic across client machines makes applications difficult to deploy and maintain.

1.1.4 Business logic tier

The business logic tier consists of the group of components that implements business rules.

Components that belong to the business logic tier centralize business logic, and separate this logic from the user interface and from the data access tier. This simplifies maintenance and the integration of new components and subsystems.

Business logic tier components have the following characteristics:

- Responsibility for referential integrity by maintaining relations and record sets for a transaction or task.
- Communication with the components in the presentation tier using disconnected record sets. This ensures a clear separation between presentation and application logic. You can implement communication by using UNIFACE XML stream parameters.
- Encapsulation of physical data access via the network by means of the UNIFACE Router and a set of middleware drivers.

In UNIFACE, the responsibility for the business logic tier is handled by two UNIFACE component types: the session service and, optionally, the entity service. The session service handles complex business rules affecting multiple entities. The optional entity service handles simple business rules and is used for strategic reasons in distributed deployment environments that require centralized execution of simple business rules and centralized control of data access.

1.1.5 Data access tier

The data access tier can be seen as a simple 'wrapper' for the databases and storage devices, which retrieves data and transforms it into a suitable format for the rest of the application.

These functions are fulfilled by the UNIFACE infrastructure. The data access tier is all of the UNIFACE functionality that handles data storage and retrieval.

The data access tier consists of the following:

- Physical database structures captured in the relational business object model
- DBMS drivers that encapsulate physical data access

Data-access tier functionality passes data to the business logic tier using a standard data interchange format. This hides the complexity of the data storage medium from the business and presentation tiers. As a result, UNIFACE is platform- and location independent.

UNIFACE provides comprehensive support for a wide range of databases and distributed architectures. The UNIFACE infrastructure of database drivers combined with the UNIFACE Router provides a complete data tier for UNIFACE applications.

1.1.6 Disconnected record sets

A disconnected record set is a package of data that represents a group of occurrences of a UNIFACE entity (optionally including any inner entities). Data state is not maintained by a connection to the data source. Instead, the disconnected record set contains processing information that allows reconnection of the disconnected data to its source.

Disconnected record sets are handled in UNIFACE using `xmlstream` parameters. UNIFACE Server Pages generate HTML that emulates disconnected record sets.

For example, it is the responsibility of the business tier to enforce business rules. Presentation tier components have to submit data to the business tier for processing, and business tier components must reconnect this data to other data in the business and data tiers before processing can occur.

Note: Web deployment of the presentation layer always requires disconnected record sets, because no state is maintained between Web client and Web server. Use of disconnected record sets throughout an application enables Web deployment without compromising the existing application infrastructure.

1.2 XML streams

XML streams are data streams using an XML format. XML streams enable loosely coupled connections between components, and provide a standardized format for data transport between components. UNIFACE can also attach processing information to data in an XML stream to allow the data state to be transported with the data.

1.2.1 Loosely coupled connections

Components are connected by the operations they expose to each other, and these connections can be loosely or tightly coupled. Components are loosely coupled if you can make changes to the internal structure of a component without having to make changes to the components with which it communicates. Components are tightly coupled if changes to one component necessitates changes to other related components.

Loosely coupled connections are a good practice in component-based development in general, and this is particularly true for three-tier development. Loosely coupled connections give great flexibility to the architecture and implementation of each tier.

A loosely coupled connection is defined solely by the data interchanged between the components, and is independent of the internal state or structure of the components themselves. Thus, the structure of the data streams between components is the crucial consideration when building for loosely coupled connections:

- If the data stream can be described rigorously, a component can declare its data requirements to the outside world.
- If the stream can be processed and transformed easily, the data in the stream can be loaded into the component regardless of how the structure of the stream or the component changes.
- If the stream can be processed by generally available, reusable software, the stream can be passed to 3GL components.

XML and loosely coupled connections

XML meets the requirements stated above:

- XML uses DTDs for structural definitions. Components that use XML to exchange data can declare that they can process data conforming to a given DTD, and other components can produce data targeted at the DTD without requiring any knowledge of the target component.
- XML is supported by other technologies and standards, for example, XSL, that enable transformation of XML into a wide range of output formats.
- Third-party XML parsers and processors are readily available on most infrastructures.

1.2.2 XML stream

An XML stream is a packet of XML data sent as a parameter from one component to another. The data is processed to comply with the XML 1.0 standard.

UNIFACE can transfer field data, data relationships, and processing information by XML stream parameters.

Field data

UNIFACE converts data in fields to XML element values. This can require some conversion of data types, and substitution of characters reserved for XML syntax. These tasks are handled by the Proc `xmlload`, `xmlsave`, and `retrieve/reconnect` statements.

The table describes how UNIFACE converts data from UNIFACE data types to XML element values in an XML stream. In general, data is converted to the UNIFACE String data type, and the string is then loaded into the XML stream using the procedure described below for String data:

Table 1-1 Conversion of UNIFACE data types to PCDATA in XML streams. ***part 1 of 2***

UNIFACE data type	Converted to...	Description
String	String, with substitution of predefined entities	No conversion takes place, except: • substitution of predefined entities for quote ("), apostrophe ('), angle bracket (< and >) and ampersand (&) characters. • profile characters and subfield separators are converted to a backslash (\) keystroke combination. For example, GOLD * is converted to *.[1]
Special string		As for String.
Raw data	String, with substitution of predefined entities	The field is converted into a string using an internal base 64 encoding.

Table 1-1 Conversion of UNIFACE data types to PCDATA in XML streams. ***part 2 of 2***

UNIFACE data type	Converted to...	Description
Numeric	String, with substitution of predefined entities	The format of the string is based on the field's Display format, or on the format specified in the Format trigger.
Floating decimal point	String, with substitution of predefined entities	The format of the string is based on the field's Display format, or on the format specified in the Format trigger.
Date	String, with substitution of predefined entities	The format of the string is based on the field's Display format, or on the format specified in the Format trigger.
Time	String, with substitution of predefined entities	The format of the string is based on the field's Display format, or on the format specified in the Format trigger.
Combined date and time	String, with substitution of predefined entities	The format of the string is based on the field's Display format, or on the format specified in the Format trigger.
Boolean	String	"F" (false) or "T" (true).
Image (all types)	String, with substitution of predefined entities	The field is converted into a string using an internal base 64 encoding.
Linear date	String, with substitution of predefined entities	The format of the string is based on the field's Display format, or on the format specified in the Format trigger.
Linear time	String, with substitution of predefined entities	The format of the string is based on the field's Display format, or on the format specified in the Format trigger.
Linear date and time	String, with substitution of predefined entities	The format of the string is based on the field's Display format, or on the format specified in the Format trigger.

Table notes:

1. `xmlload` does not convert profile character and subfield separator escape sequences to the corresponding profile character or subfield separator if the `xmlload/noprofile` switch is specified.

The table gives replacement texts for each predefined entity specified in the XML standard. An XML parser replaces the entities with the replacement texts after parsing the XML source.

Table 1-2 Predefined entities.

Predefined entity	Replacement text
>amp;	>
<amp;	<
&amp;	&
&apos;	'
&quot;	"

Data relationships

UNIFACE entities are represented as XML elements in XML streams. The fields of an entity are represented as child elements of the entity's element.

Inner entities are represented as child elements of the outer entity.

XML streams do not make a distinction between database and non-database fields, or between fields and entities defined in a component or in an application model.

Note: The structure of an XML stream (including the names, nesting, and order of all elements) is declared by a DTD, which is defined in your application model.

Note: All field and entity data is treated in an identical manner by XML streams. Data is saved into, and loaded from, XML streams in the same way, regardless of whether the entity or field occurs in your application model.

Processing information

UNIFACE can include processing information in the XML stream, describing the modification and validation status of the occurrences, and also enabling reconnection of the data to its source. For more information, see section 1.2.4 *Attributes in XML streams*.

The Proc functions `$occcrc`, `$occstatus`, `$occproperties` and `$fieldproperties` can be used to access processing information loaded from an XML stream. The statement `retrieve/reconnect` requires processing information in order to reconnect data to a database.

1.2.3 Processing information

Processing information is information about data—whether data is new or already stored in a database, or whether data is valid or nonvalid.

Processing information is used in all versions of UNIFACE to manage exchanges of data between the database and components. This occurs transparently, so that a programmer generally does not have to handle data state. UNIFACE XML streams expose this information to programmers and 3GL components in the form of XML attributes.

For example, when an occurrence is deleted on a UNIFACE form with direct database access, UNIFACE marks the occurrence for deletion in the component's data structure—but the database is not modified. Instead, the request to delete the occurrence is stored as *processing information* in the component's data structure. When the user stores the data, UNIFACE deletes the occurrence from the database and from the component's data structure. In default UNIFACE I/O, the processing information used by UNIFACE to manage changes in data between the component data structure and the database is hidden from the programmer.

XML streams simply add this information to the data when it is transformed into XML format. Components receiving the XML stream extract the processing information from the XML stream when loading the stream's data into component fields. This enables interchange of both data and data state between components. This mechanism also enables UNIFACE to interchange data state with 3GL components.

1.2.4 Attributes in XML streams

UNIFACE uses XML attributes to store meta information about occurrences in an XML stream, and allows you to define additional attributes.

Meta information attributes

The attributes UNIFACE can generate to store state information in an XML stream are described below. UNIFACE automatically generates the attributes if they are specified in the DTD used by the stream.

The table shows the attributes generated by UNIFACE for XML streams.

Table 1-3 UNIFACE-generated attributes in XML streams. ***part 1 of 2***

Attribute	Allowed values	Valid for...[1]	Required by...[2]	Description
`id`	UNIFACE-generated base-64 encoded string	entity elements	`retrieve/ reconnect`	UNIFACE merges occurrences with matching ID attributes. `retrieve/ reconnect` also calculates ID attributes for database occurrences, enabling merging of XML data with database occurrences.
`crc`	UNIFACE-generated hexadecimal string, or a user-defined string set using `$occcrc`	entity elements	`retrieve/ reconnect`	UNIFACE only merges XML occurrences with database occurrences if the `crc` attribute of the XML occurrence matches the CRC checksum calculated for the database occurrence.
`status`	`new`—the occurrence is new, and does not represent a database occurrence			If `status="new"`, `retrieve/ reconnect` creates a new occurrence to hold the reconnected record.

Table 1-3 UNIFACE-generated attributes in XML streams. ***part 2 of 2***

Attribute	Allowed values	Valid for...[1]	Required by...[2]	Description
	`del`—the occurrence is marked for deletion			If `status="del"`, `retrieve/ reconnect` marks the matching occurrence in the hitlist for deletion.
	`est`—the occurrence may be reconnectable to a database occurrence	entity elements	`retrieve/ reconnect`	If `status="est"`, `retrieve/ reconnect` attempts to update the existing occurrence with data from the XML stream. If an existing occurrence is not found in the database or the component, a new occurrence is created to hold the reconnected record.
`valerr`	UNIFACE or user-defined error message	entity elements and field elements	Remote validation	This attribute holds validation error messages. `xmlload` fires the On Error trigger for each field or occurrence with a `valerr` attribute.

Table notes:

1. Elements in an XML stream are either mapped to entities or fields painted on a UNIFACE component. *Entity elements* are mapped to UNIFACE entities, and *field elements* are mapped to UNIFACE fields.

2. XML streams can be used without any of these attributes, but the attributes are required by some Proc. In particular, `retrieve/ reconnect` has to resolve the origin and state of occurrences before it can merge XML stream data with database data.

id

`id` is a unique identifier based on the primary key of the occurrence. If an occurrence is new, the value of `id` is based on the occurrence's internal identifier in the component's data structure.

Note: In some cases the `id` attribute could be reserved by other software. To handle these situations, UNIFACE allows you to define a `uid` attribute, which has the same purpose and behavior as the `id` attribute.

crc

`crc` is a cyclical redundancy checksum (CRC) based on the field values at the time the occurrence was saved into an XML stream. When the XML stream is reconnected to the database, the CRC value is recalculated and compared with the CRC value in the XML stream. The CRC values must match or the occurrence is not reconnected. (A mismatch indicates that the data has been changed in the database since the disconnected record set was created). This results in behavior identical to optimistic locking.

status

`status` stores the modification status of an occurrence. The following values are allowed for `status`:

- `"est"`—the occurrence exists in the database
- `"new"`—the occurrence is new
- `"del"`—the occurrence is marked for deletion

valerr

If a data validation error occurred while reconnecting data from an XML stream to a database, the validation errors can be written to an XML stream sent back to the calling component. These validation errors are stored in the `valerr` attribute.

User-defined attributes

You can define your own attributes by declaring them in the DTD for the XML stream. UNIFACE supports `FIXED` attributes, that is, attributes with a fixed value. For more information about the syntax for declaring attributes, see section 3.1 *Attribute declarations*.

1.2.5 Document Type Definition (DTD)

A Document Type Definition (DTD) defines rules for the structure of an XML stream. DTDs are defined in the XML standard, issued by the World Wide Web Consortium.

A DTD is also a UNIFACE Repository object, stored in the UCDTYP.DICT table. The DTD Repository object includes DTD definitions and a default mapping between the elements defined by the DTD and component fields and entities.

Compilation of DTDs

Compilation of DTD objects updates the ULANA table. If a DTD specifies a file name in its properties, compilation of that DTD also creates a text file for that DTD with a **.dtd** file extension. These files conform to the XML standard for DTDs.

The /urr command line switch copies the DTD declarations, default mapping, and other properties from the ULANA table to the URR file.

The default mapping between XML elements defined by the DTD and UNIFACE fields is compiled in the ULANA table, and is not stored in the DTD text file.

XML structures you can define with a DTD

An XML DTD can specify the following XML structures:

- What XML elements are allowed in an XML stream.
- The multiplicity of each element. An element can be set to occur:
  - once only
  - zero or once
  - any number of times
  - once or more times
- The content of an element. An element can contain data, other elements, or a combination of data and elements.
- The order in which elements can occur. Elements can be constrained to occur in a specific sequence, or they can be allowed to occur in any order.
- The attributes of an element.

Caution: DTDs allow data structures that do not correlate to relational database or UNIFACE data structures. Ensure that your DTDs always meet the following criteria:

DTD checklist

XML DTDs allow data structures that are not relevant to UNIFACE.

Ensure that your DTD meets the following requirements:

- Elements that represent fields:
  - *must* have PCDATA as their content model
  - *must* be a child of an element representing the field's entity
  - *must* occur once within each instance of the parent element

- Elements that represent entities:
  - *must* contain elements only
  - *can* be set to occur exactly once, 0-n times, 1-n times, or 0-1 times within each instance of their parent element
- Always:
  - *avoid* mixed content (this means that you must never allow an element to contain PCDATA and elements)

1.3 DTD Editor

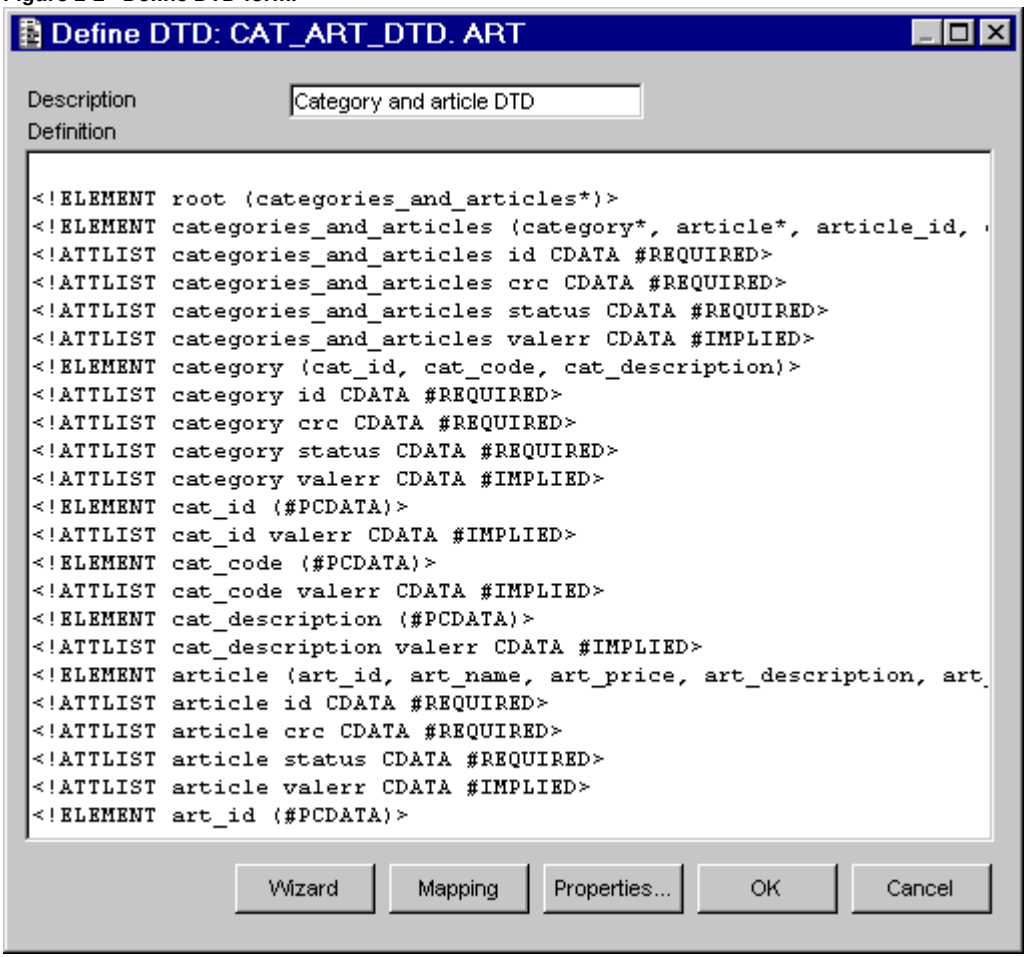
UNIFACE incorporates a DTD Editor to create and modify XML DTDs for XML streams.

1.3.1 DTD Editor

The DTD Editor is a tool in the UNIFACE Development Environment that allows you to create and edit Document Type Definitions (DTDs). For more information on starting the DTD Editor, see section 5.1.1 *Start the DTD Editor*. DTDs are sets of rules that define the structure of an XML stream. For more information about DTDs, see section 1.2.5 *Document Type Definition (DTD)*.

The DTD Editor consists of several forms, including the Define DTD form, the DTD Wizard, the Define DTD Properties form and the DTD: Default Mapping form. The DTD Editor starts by displaying the Define DTD form.

Figure 1-2 Define DTD form.



The DTD Editor displays the DTD for editing. For more information on creating and editing DTDs with the DTD Editor, see section 5.1 *Using the DTD Editor*. The DTD Editor provides the following utilities:

- Load from file—create a DTD in your Repository from a DTD in your file system.

- Load component structure—create a DTD from the data structure painted on a component.
- Properties—define additional properties for the DTD. For example, you can define a URI location (an Internet address) for the DTD so that non-UNIFACE components can locate the DTD.
- Mapping—define a default map between XML elements and UNIFACE fields and entities, so that UNIFACE can transport data to and from the XML stream.
- Validate—check that the DTD is syntactically correct.
- DTD Wizard—create and manipulate DTDs using a graphical interface. For more information about the DTD Wizard, see section 1.3.2 *DTD Wizard*.

Menu options

The options available on the File menu include:

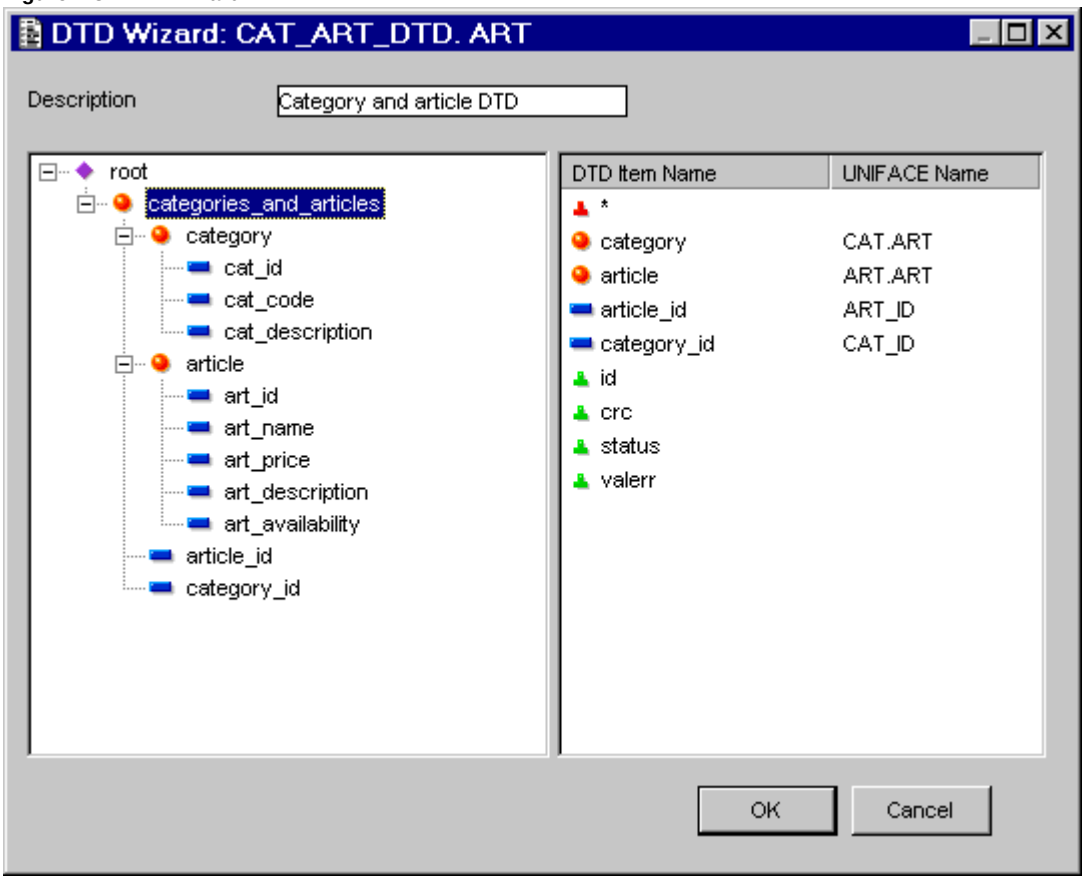
- New DTD
- Open
- Save
- Duplicate DTD
- Delete DTD
- Load Component Structure
- Load from File
- Save to File
- Validate
- Compile DTD

Note: The Save to File option saves the DTD as a file using the DTD format specified in the XML standard. Save to File does not store your changes in the Repository, and it does not copy UNIFACE-specific information about the DTD to the file. For example, additional properties and mapping defined for the DTD are not included in the file created by Save to File.

1.3.2 DTD Wizard

The DTD Wizard is part of the DTD Editor. The DTD Wizard simplifies the task of writing and maintaining DTDs by presenting DTDs graphically, using familiar UNIFACE icons. In particular, the DTD Wizard assists in the task of creating DTDs based on application model objects.

Figure 1-3 DTD Wizard.



The DTD Wizard is divided into two panes:

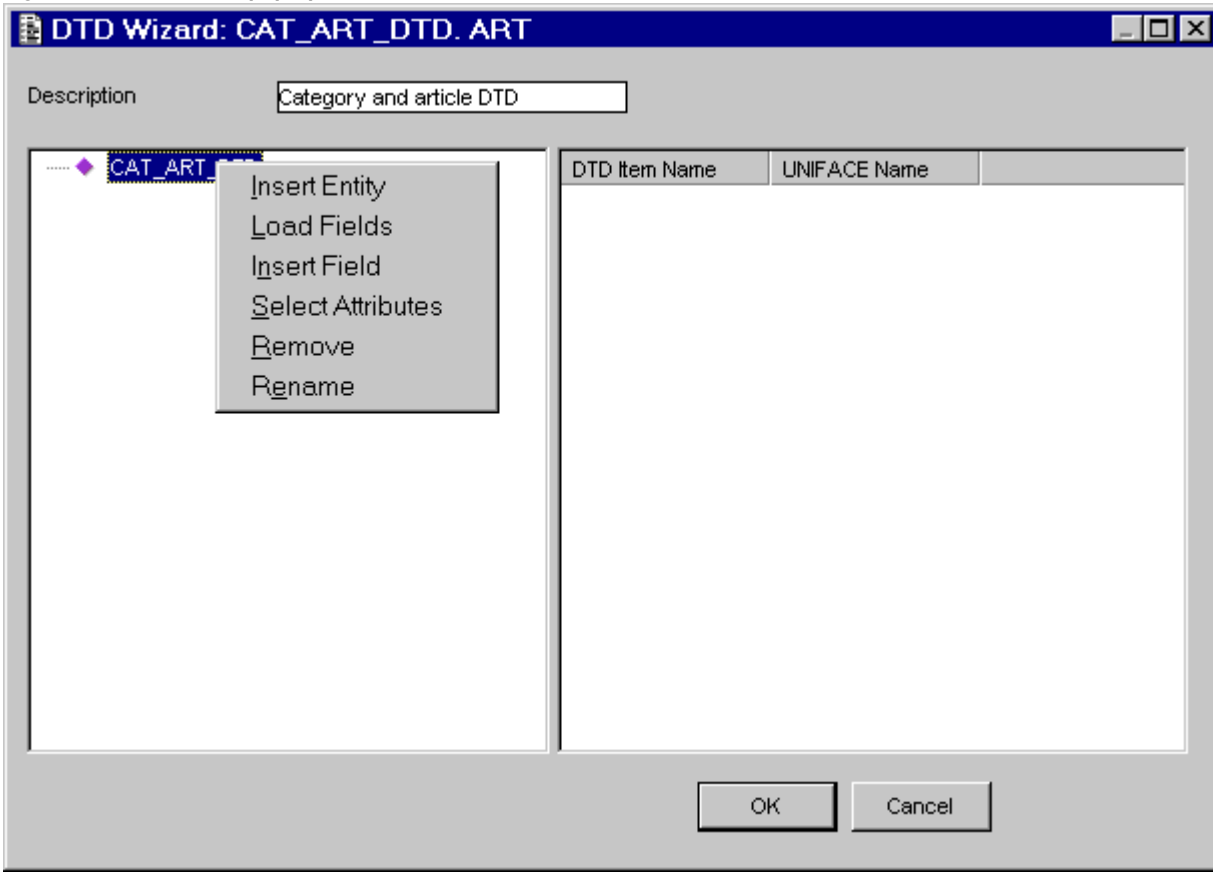
- The DTD Tree is displayed on the left-hand side of the Wizard. The DTD Tree shows all the element declarations in the DTD as a tree

structure, with each element declaration represented as a node in the tree. You can select items in the DTD Tree by clicking on them.

- The right-hand side of the DTD Wizard shows a list of all the DTD declarations linked to the current node in the DTD Tree. The list view also shows how the element names are mapped to UNIFACE field and entity names.

Using the DTD Wizard, you can create syntactically valid DTDs without any knowledge of DTD syntax. To add, remove, rename, or otherwise modify an object in your DTD, right-click on the object in DTD Tree and select an action from the pop-up menu.

Figure 1-4 DTD Wizard pop-up menu.



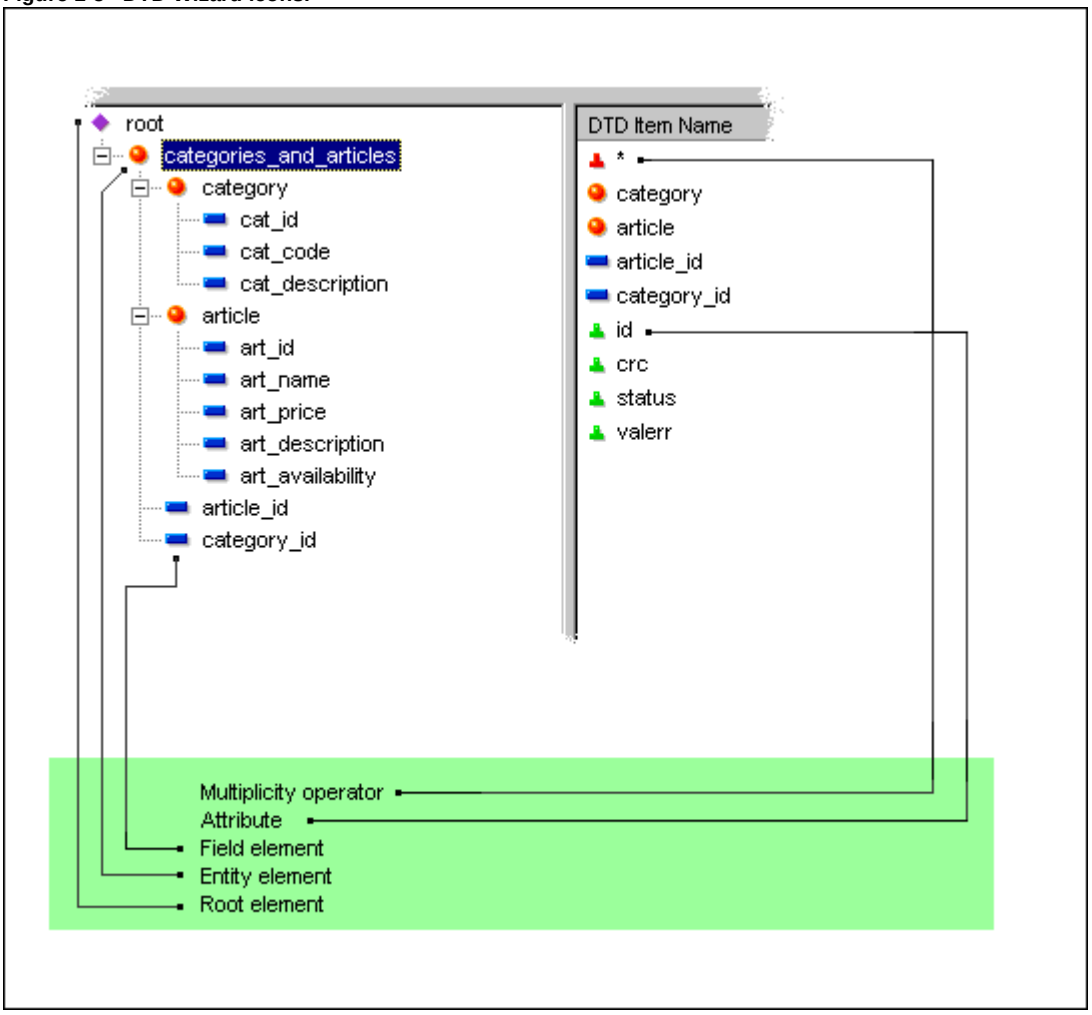
DTD Tree

DTDs declare elements that serve different roles in the XML stream. Some elements represent UNIFACE entities, and others represent fields. One element is the root element of the stream—it encloses all the other elements in the stream, but it does not represent any UNIFACE object.

The DTD Tree uses icons to show the role of the elements declared by the DTD:

- Root element—all DTDs define an element that encloses all other elements in an XML stream. This element is called the root element. The root element is not mapped to any UNIFACE object.
- Entities—elements that represent UNIFACE entities are displayed using the standard UNIFACE entity icon.
- Fields—elements that represent UNIFACE fields are displayed using the standard UNIFACE field icon.
- Root element—this element is displayed as a purple icon.

Figure 1-5 DTD Wizard icons.



List view

The list view displays information about the currently selected node in the DTD Tree. In addition to showing icons described for the DTD tree, the list view displays the following icons:

- Attributes—these are represented by a green icon.
- The multiplicity operator (which determines how many times an element can occur in the XML stream) is represented by a red icon.

Table 1-4 Multiplicity operators.

Operator	Description
*	The element may occur zero, once, or many times within its parent element.
?	The element may occur zero or once within its parent element.
+	The element may occur once or many times within its parent element.
No operator	The element must always occur once within its parent element.

*Note: The DTD Wizard always sets the multiplicity operator for all entity elements to * (the element may occur zero to many times in the stream). Field elements are always constrained to appear exactly once per occurrence of their enclosing entity element.*

Note: If it is necessary to change the multiplicity operator for an element, you have to edit the DTD in the Define DTD form of the DTD Editor.

1.4 UNIFACE processing of XML streams

This section describes how UNIFACE creates XML streams, and how UNIFACE extracts data from XML streams.

1.4.1 Saving data into an XML stream

UNIFACE components can transfer field values (and their modification and validation state) into XML streams.

The XML stream is stored in a variable or parameter with xmlstream data type.

xmlsave

The xmlsave statement saves the data from fields painted on a component into an XML stream. The xmlsave statement does the following:

- String data in each field is converted to comply with the XML 1.0 standard. This includes replacing angle brackets (< and >), ampersand (&), and quotation marks (' and ") with appropriate escape sequences. In the XML 1.0 standard, these escape sequences are called predefined entities.

Refer to table 1-2 for replacement texts for each predefined entity specified in the XML standard. An XML parser replaces the entities with the replacement texts after parsing the XML source.

- Raw and image data types are converted into strings using a base 64 encoding.
- Numeric data is converted to string data, using the display format of the field.

Refer to table 1-1 to see how UNIFACE converts data from UNIFACE data types to XML element values in an XML stream. In general, data is converted to the UNIFACE String data type, and the string is then loaded into the XML stream using the procedure described below for String data:

The original field values on a component are unchanged by xmlsave. Validation triggers are not fired by xmlsave. The active path is not changed by xmlsave.

Saving state information in an XML stream

If a DTD specifies meta information attributes for an element, meta information about the modification and validation state is included in the XML stream for that element.

You can include state with selected UNIFACE entities in an XML stream by specifying state attributes in your DTD only for those entities that require state information.

Triggers fired by xmlsave

The following triggers are fired by the xmlsave statement:

- *Pre Save Occurrence*

The Pre Save Occurrence trigger is fired immediately before an occurrence is saved from a component into an `xmlstream`. The occurrence is available and can be examined.

This trigger tunes the execution of `xmlsave`. You can use this trigger to customize the process of saving a component into an `xmlstream`. For example, an occurrence can be excluded from the save, or the value for a derived field can be calculated.

- *Post Save Occurrence*

  The Post Save Occurrence trigger is fired immediately after an occurrence is saved from a component into an `xmlstream`.

  This trigger tunes the execution of `xmlsave`. You can use it to customize the process of saving a component into an `xmlstream`.

1.4.2 Loading data from XML streams

The `xmlload` statement extracts data from an XML stream and places the data in fields painted on the component. The `xmlload` statement uses a DTD and an element-to-field map to do this conversion.

Element values are converted into UNIFACE field values, as described in the following table.

Refer to table 1-1 to see how UNIFACE converts data from UNIFACE data types to XML element values in an XML stream. In general, data is converted to the UNIFACE String data type, and the string is then loaded into the XML stream using the procedure described below for String data:

Loading state from an XML stream

UNIFACE uses attributes to determine the state of data in an XML stream. If these attributes are specified in the DTD and are encountered in the XML stream, the state specified in the attributes is applied to occurrences loaded from the stream. If no state is defined in the stream, UNIFACE treats all occurrences as new.

Data validation error messages can be present in the XML stream. If UNIFACE encounters these, the On Error trigger is fired for each validation error included in the stream. For example, the On Error trigger could change the display color of a field to indicate a validation error.

For more information about XML stream state attributes, see section 1.2.4 *Attributes in XML streams*.

Triggers fired by xmlload

The `xmlload` statement fires the following triggers:

- *Pre Load Occurrence*

  The Pre Load Occurrence trigger is fired immediately before an occurrence is loaded from an `xmlstream` into a component. The new occurrence is not yet available and cannot be accessed.

  This trigger tunes the execution of `xmlload` and `xmlsave`. You can use this trigger to customize the process of loading an `xmlstream` into a component.

- *Post Load Occurrence*

  The Post Load Occurrence trigger is fired immediately after an occurrence is loaded from an `xmlstream` into a component. The new occurrence is available and can be accessed.

  This trigger tunes the execution of `xmlload`. You can use it to customize the process of loading an `xmlstream` into a component. For example, an occurrence can be discarded, or the value for a derived field can be calculated.

1.4.3 Null values in XML streams

Null fields are saved into XML streams as empty elements. This means that a null field is indistinguishable from an empty string.

Null numeric fields

A null numeric field is saved into an XML stream as an empty element, in the same way as any other data type. When the XML stream is loaded into a component, the empty element is converted into a zero numeric value.

Null values and absent elements

Fields in occurrences created by `xmlload` that do not have an XML stream element mapped to them are treated as null fields. That is, when `retrieve/reconnect` merges the occurrences created from the XML stream with database occurrences, the null fields do not overwrite the database fields.

Included entities are an exception to this behavior. For more information, see section 1.4.4 *Included entities and XML streams*.

1.4.4 Included entities and XML streams

During reconnection of disconnected record sets, occurrences of included entities are completely overwritten by the data in the disconnected record.

Caution: If there is no element in the XML stream for a given field of an included entity, that field is emptied. This means that if all the data of an included entity is not placed in an XML stream, data loss can occur when the XML stream is reconnected. This restriction only applies to included entities. For more information on how UNIFACE handles partial records in XML streams, see section 1.4.3 Null values in XML streams.

Therefore, if any data from an included entity is required in an XML stream, it is essential to include all the fields of the included entity in the XML stream.

1.4.5 Application of DTDs to an XML stream

UNIFACE generates XML streams using the structural definitions in the DTD specified by the `xmlsave` statement.

UNIFACE does not validate XML streams against the DTD declared in the `variables` or `params` blocks.

UNIFACE does not validate XML streams against the DTD specified in the `xmlload` statement.

Note: Future versions of UNIFACE could feature validation of XML streams against the DTDs specified in Proc. To be certain of future compatibility, ensure that your XML streams are valid against the DTDs specified by these statements.

1.4.6 Default DTD mapping and mapping defined on a component

UNIFACE supports mapping of field and entity names to elements in XML streams. This enables UNIFACE to transport values between XML streams and component fields. For example, UNIFACE requires a mapping structure to transport values between an XML element named `personName` and a field named NME.PERSON.

A default mapping can be defined in your application model, and mappings can also be defined locally in Proc, as associative lists. For more information, see section 5.2.3 *Define a local mapping for a DTD* and section 5.1.7 *Define a default mapping for a DTD*.

The default mapping is applied to an XML stream by `xmlload` if the `/incldefmap` switch is specified. The local mapping is applied if a mapping list is included as an argument for `xmlload`.

If both the local and default mappings are specified, the local mapping is combined with the default mapping. If the default and local mappings conflict, then the local mapping takes precedence.

If a mapping scheme addresses fields that are not available on a component, or elements that are not in the XML stream, that part of the mapping is ignored.

1.4.7 DTD mapping lists

DTD mapping lists map the values of elements in a DTD stream to fields on a component.

Default DTD mapping lists can be defined in the DTD Editor. DTD mapping lists can also be defined in Proc, for use in the Proc statements `xmlload` and `xmlsave`.

DTD mapping lists are UNIFACE associative lists. Each list item has the following syntax:

"*ElementName=TargetName*;..."

Where:

- *ElementName*—is the name of an element in the DTD. *ElementName* is case-sensitive.
- *TargetName*—is the name of a field or entity. The fields or entities do not have to be present on the target component. Values that cannot be mapped to a painted field are ignored. *TargetName* is not case-sensitive.

Order of DTD mapping list items

When creating a mapping list, the order of the list items can be significant if you are mapping data to UNIFACE entities with overlapping field names. In this situation, you should ensure that mapping list follows the order in which elements first appear in the XML stream.

For example, if you are mapping an XML stream to the entities COUNTRY and PERSON, and both these entities have a field NAME, then the order of the mapping list is significant. If COUNTRY data appears first in the XML stream, then your mapping list should specify a mapping for NAME.COUNTRY before specifying a mapping for NAME.PERSON.

Chapter 2 Modifications to the Repository for XML streams

The Repository has been extended to store information about DTDs and the default mapping of data between XML streams and components.

A new entity, UCDTYP.DICT, has been added to the DICT model in the meta dictionary. This section describes the parts of the meta dictionary that have been modified to support XML streams.

UCDTYP.DICT

UCDTYP contains information on DTDs in application models. Each occurrence is identified by the name of the DTD, and the application model.

Table 2-1 Fields for entity UCDTYP.DICT.

Field Name	Type	In DB?	Description / Comment	Interface / Syntax / Layout
UTIMESTAMP	E	Y	Last Update	I: @STAMP S: L:
UCOMPSTAMP	E	Y	Last Compilation	I: @STAMP S: L:
UDTDNAME	SS	Y	Short Name of DTD	I: @NAME32 S: L:
UMODELNAME	SS	Y	Application Model Name	I: @NAME32 S: L:

Table 2-1 Fields for entity UCDTYP.DICT.

Field Name	Type	In DB?	Description / Comment	Interface / Syntax / Layout
UVERS	S	Y	Version Number	I: C12 S: L:
UDESCR	SS	Y	Description	I: @DESCR S: L:
UDTDTYPE	S	Y	DTD Type Reserved for future use; currently the default, and only allowed value, is DTD, meaning an XML standard-compliant DTD	I: C3 S: L:
UDTDFILE	SS	Y	Name of DTD file The file to which the DTD can be mirrored on disk	I: C40 S: L:
UDEFMAP	SS	Y	Default Mapping Associative list of mappings	I: @B196 S: L:
UDTDREF	SS	Y	Reference	I: @B193 S: L:
UCOMMENT	SS	Y	Comments	I: @B194 S: L:
UDEFINTION	SS	Y	DTD Data Textual representation of the DTD Data	I: @B195 S: L:

Table 2-2 Keys for entity UCDTYP.DICT.

Key	Type	Key field name
1	Primary	UDTDNAME, UMODELNAME

Table 2-3 Foreign keys for entity UCDTYP.DICT.

Foreign key field name	One entity	Index	Relationship name
UMODELNAME	UCSCH	N	U_UCSCH_UCDTYP_DICT

Chapter 3 UNIFACE support for XML and DTD syntax

This is a description of UNIFACE support for DTD and XML syntax as defined in the XML standard.

3.1 Attribute declarations

UNIFACE supports a subset of the attribute declarations that are available in the XML standard. UNIFACE supports the attribute declarations required for inter-component communication using XML streams, namely declarations for attributes for processing information, and declarations for namespaces.

The syntax supported by UNIFACE is described below, and a description of the full syntax is also given for comparison.

Attribute declaration syntax as supported by UNIFACE (expressed in UNIFACE conventions)

You can use the following attribute declarations in your DTDs:

- `<!ATTLIST` *EntityElement* `id CDATA #REQUIRED>`
- `<!ATTLIST` *EntityElement* `crc CDATA #REQUIRED>`
- `<!ATTLIST` *EntityElement* `status CDATA #REQUIRED>`
- `<!ATTLIST` *EntityElement* `valerr CDATA #IMPLIED>`
- `<!ATTLIST` *FieldElement* `valerr CDATA #IMPLIED>`
- `<!ATTLIST` *ElementName AttributeName* "*AttributeValue*" `#FIXED>`

Where:

- *EntityElement* is the name of an element mapped to a UNIFACE entity
- *FieldElement* is the name of an element mapped to a UNIFACE field
- *ElementName* is either an *EntityElement* or *FieldElement*
- *AttributeName* is the name of the attribute
- *AttributeValue* is the value of the attribute

Use #FIXED attributes to declare namespaces for your XML element names.

XML standard syntax (expressed in UNIFACE conventions)

The syntax for attribute declarations specified in the XML standard is:

<!ATTLIST *ElementName AttributeName TokenizedType* | CDATA | *ValueList* {#REQUIRED|#FIXED|#IMPLIED} *DefaultValue*>

Where:

- *ElementName*—is the name of the element to which the attribute belongs
- *Attributename*—is the name of the attribute
- *CDATA*-is string data
- *ValueList*—is a list of optional values, with the following structure: ($Value_1$ | $Value_2$ | $Value_3$ ...)
- *TokenizedType*—is, for example, a unique ID number, or the name of an entity declared within the XML document
- *REQUIRED*—declares the attribute is always present with a value
- *#FIXED*—declares that the attribute is always present and always has the default value
- *#IMPLIED*—declares that no default value is provided for the attribute
- *DefaultValue*—is the default value of the attribute. A default value cannot be specified for #REQUIRED or #IMPLIED attributes.

The XML standard also allows an alternative syntax for attribute declarations, whereby all attributes are declared for an element in one <!ATTLIST> declaration. This alternative syntax is not supported by UNIFACE.

3.2 Element declarations

UNIFACE supports a subset of the element declarations available in the XML standard. UNIFACE supports the element declarations required for intercomponent communication using XML streams, namely declarations for root elements, elements representing entities, and elements representing fields.

The syntax supported by UNIFACE is described below, and a description of the full XML standard syntax is given for comparison.

3.2.1 Root element declarations

All XML streams have a root element that contains all the other elements in the stream.

You can declare the root element of your DTD as follows:

`<!ELEMENT` *RootName* `(`*OuterEntity*{*MultiplicityOperator*}`)>`

where:

- *RootName* is the name of the root element of the entity. The root element is not mapped to any UNIFACE object, so it is conventional to give the root element the same name as the DTD or to name it `ROOT`.
- *OuterEntity* is the name of the element mapped to the outermost UNIFACE entity in the XML stream.
- *MultiplicityOperator* is a single character that defines how many times an element can occur in the stream. The available options are:
  - ?—the element can occur zero or one times in the stream.
  - *—the element can occur zero, once or many times in the stream.
  - +—the element can occur once or many times in the stream.
  - *No operator*—the element must occur exactly once in the stream.

3.2.2 Element declarations for UNIFACE entities

UNIFACE entities are represented as XML elements in an XML stream. An element that represents an entity contains elements that represent the inner entities and fields of that entity.

You can declare elements for entities in your DTDs by using the following syntax:

<!ELEMENT *EntityName* ({*InnerEntity*$_1$ {*MultiplicityOperator*} | *FieldName*$_1$ }{, *InnerEntity*$_2$ {*MultiplicityOperator*} | *FieldName*$_2$ }...{, *InnerEntity*$_n$ {*MultiplicityOperator*} | *FieldName*$_n$ })>

Where:

- *EntityName*—is the name of the element mapped to the UNIFACE entity.
- *InnerEntity*—is the name of an element mapped to an inner entity.
- *FieldName*—is the name an element mapped to a field of the UNIFACE entity.

3.2.3 Element declarations for UNIFACE fields

UNIFACE fields are represented as XML elements in an XML stream. An element that represents a field contains PCDATA (string-type data), and no child elements.

<!ELEMENT *FieldName* (#PCDATA)>

Where *FieldName* is the name of an element representing a field. *FieldName* must be specified in the declaration of an entity element.

3.2.4 XML standard syntax (expressed in UNIFACE conventions)

The XML standard's syntax for element declarations is:

<!ELEMENT *ElementName* (*Content*)>

Where:

- *ElementName*—is the name of the element
- *Content*—is the content allowed for that element. This can be one of the following:
  - EMPTY—specifies that the element has no content.
  - ANY—specifies that there are no restrictions on the content of the element.
  - PCDATA—specifies that the element contains parsed character data (string data).
  - *ContentList*—this is a list of child elements and PCDATA sections that the element can contain.

ContentList has the following format:

{(}*element*$_1$ *MultiplicityOperator connector element*$_2$ *MultiplicityOperator... connector element*$_n$ *MultiplicityOperator*{)}*MultiplicityOperator*}

Where:

- *element*—is the name of a child element, or is a nested *ContentList*
- *connector*—is either a comma (,), or a pipe (|).

3.3 Miscellaneous DTD and XML syntax support

This is a description of UNIFACE support for a range of DTD and XML syntax issues.

3.3.1 Comments

The DTD syntax for comments is not supported by UNIFACE. Instead, place comments about the DTD in the Comments field in the Define DTD Properties form.

UNIFACE does not generate comments in XML streams. If comments are encountered in an XML stream, the comments are ignored by `xmlload`.

3.3.2 White space

The DTD Editor supports the use of white space (tabs and spaces) for DTDs. All white space is replaced by a single space character when the DTD is edited in the DTD Wizard.

The XML standard allows hard return characters in white space. This is not supported by the DTD Editor or by the DTD Wizard.

3.3.3 Processing Instructions

Processing instructions are not generated by UNIFACE, and are not processed.

If processing instructions are encountered in an XML stream, the comments are ignored by `xmlload`.

3.3.4 Unique element names

The XML standard allows an element to be declared once and to be used in several elements. For example, a `paragraph` element could be defined and used as part of the content of `chapter` and `preface` elements. This is not supported by UNIFACE. An element can only be used within the content of a single parent. The names of child elements must be unique within the DTD.

3.3.5 XML entities

XML entities are not supported by UNIFACE, with the exception of the Predefined entities specified in the XML standard.

The table gives replacement texts for each predefined entity specified in the XML standard. An XML parser replaces the entities with the replacement texts after parsing the XML source.

Table 3-1 Predefined entities.

Predefined entity	Replacement text
`>`	>
`<`	<
`&`	&
`&apos;`	'
`&quot;`	"

Parameter entities, which are used exclusively in DTDs, are not supported.

3.3.6 Document subdeclarations

The document subdeclaration is an extension to the DTD, stored in the XML document itself. The document subdeclaration is generally used to store definitions of XML entities.

Document subdeclarations are not used by UNIFACE, and are ignored during processing of XML streams.

3.4 Sample DTDs and XML streams

3.4.1 A basic DTD and XML stream

The following DTD is a highly simplified sample, that lacks processing information attributes and namespace declarations. Processing information attributes are required to reconnect disconnected record sets to existing data for processing or storage in a database. Namespace declarations are not required, but are a good practice for ensuring that your elements names are unique in an XML stream.

The DTD defines an XML stream for a single outer occurrence of country data, with zero, one, or many occurrences of state (province) data. Due to the lack of processing information, an XML stream based on this DTD is not suitable for disconnected record sets, but is suitable for static data that is not maintained.

The DTD is based on a more sophisticated DTD, described in section 3.4 *Sample DTDs and XML streams*.

```
<!ELEMENT Root (Country)>
<!ELEMENT Country (Country:Country_Id, State*,
Country:Code, Country:Name, Country:Has_State)>
<!ELEMENT Country:Country_Id (#PCDATA)>
<!ELEMENT State (State:State_Id, State:Country_Id,
State:Code, State:Name)>
<!ELEMENT State:State_Id (#PCDATA)>
<!ELEMENT State:Country_Id (#PCDATA)>
<!ELEMENT State:Code (#PCDATA)>
<!ELEMENT State:Name (#PCDATA)>
```

```
<!ELEMENT Country:Code (#PCDATA)>
<!ELEMENT Country:Name (#PCDATA)>
<!ELEMENT Country:Has_State (#PCDATA)>
```

An XML stream generated using this DTD

The following XML stream is generated from the same data as the XML stream in section 3.4 *Sample DTDs and XML streams*. The differences between these streams is due to the differences in the DTDs.

The DTD declares that the `Root` element contains a single occurrence of `Country`, so the XML stream generated using this DTD only contains one occurrence of `Country`.

```
<?xml version="1.0"?>
<Root>
 <Country>
  <Country:Country_Id>7Z9RN42HWF3H</Country:Country_Id>
  <State>
   <State:State_Id>7Z9RNICU20O9</State:State_Id>
   <State:Country_Id>7Z9RN42HWF3H</State:Country_Id>
   <State:Code>GAU</State:Code>
   <State:Name>Gauteng</State:Name>
  </State>
  <State>
   <State:State_Id>7Z9RNICU25M2</State:State_Id>
   <State:Country_Id>7Z9RN42HWF3H</State:Country_Id>
   <State:Code>MPU</State:Code>
   <State:Name>Mpumalanga</State:Name>
  </State>
  <Country:Code>ZA</Country:Code>
  <Country:Name>South Africa</Country:Name>
  <Country:Has_State>T</Country:Has_State>
 </Country>
</Root>
```

3.4.2 Sample DTD and XML stream

The following DTD defines an XML stream for country and state data. Each country can have zero, one or many states (provinces). The DTD includes a default mapping to two entities, COUNTRY.ORD and STATE.ORD.

The DTD uses namespaces to ensure that element names are unique, and uses the processing information attributes provided by UNIFACE for all elements in the XML stream. Using this DTD, data can be sent to components as a disconnected record set, and then reconnected to existing records for processing or storage.

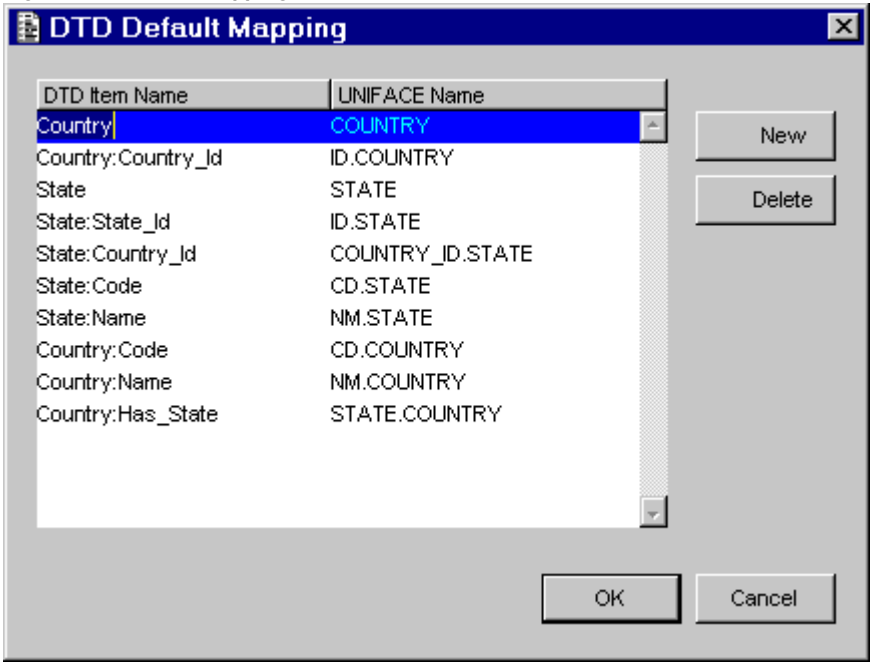
```
<!ELEMENT Root (Country*)>
<!ELEMENT Country (Country:Country_Id, State*, Country:Code, Country:Name,
Country:Has_State)>
<!ATTLIST Country id CDATA #REQUIRED>
<!ATTLIST Country crc CDATA #REQUIRED>
<!ATTLIST Country status CDATA #REQUIRED>
<!ATTLIST Country valerr CDATA #IMPLIED>
<!ATTLIST Country xmlns:Country CDATA #FIXED "http://acme.com/Country">
<!ELEMENT Country:Country_Id (#PCDATA)>
<!ATTLIST Country:Country_Id valerr CDATA #IMPLIED>

<!ELEMENT State (State:State_Id, State:Country_Id, State:Code, State:Name)>
<!ATTLIST State id CDATA #REQUIRED>
<!ATTLIST State crc CDATA #REQUIRED>
<!ATTLIST State status CDATA #REQUIRED>
<!ATTLIST State valerr CDATA #IMPLIED>
<!ATTLIST State xmlns:State CDATA #FIXED "http://acme.com/State">
<!ELEMENT State:State_Id (#PCDATA)>
<!ATTLIST State:State_Id valerr CDATA #IMPLIED>
<!ELEMENT State:Country_Id (#PCDATA)>
<!ATTLIST State:Country_Id valerr CDATA #IMPLIED>
<!ELEMENT State:Code (#PCDATA)>
<!ATTLIST State:Code valerr CDATA #IMPLIED>
<!ELEMENT State:Name (#PCDATA)>
<!ATTLIST State:Name valerr CDATA #IMPLIED>

<!ELEMENT Country:Code (#PCDATA)>
```

```
<!ATTLIST Country:Code valerr CDATA #IMPLIED>
<!ELEMENT Country:Name (#PCDATA)>
<!ATTLIST Country:Name valerr CDATA #IMPLIED>
<!ELEMENT Country:Has_State (#PCDATA)>
<!ATTLIST Country:Has_State valerr CDATA #IMPLIED>
```

Default mapping

The default mapping for this DTD maps the XML elements to COUNTRY and STATE entities in the ORD application model.

Figure 3-1 Default mapping for a DTD.



An XML stream generated using this DTD

The following XML stream was generated using this DTD. For an example of a different XML stream generated from the same data, see section 3.4.1 *A basic DTD and XML stream*.

```
<?xml version="1.0"?>
<Root>
 <Country xmlns:Country="http://acme.com/Country"
id="fS9JQzdaOVJONDJIV0YzSA=="
```

```
crc="E343DACC" status="est">
  <Country:Country_Id>7Z9RN42HWF3H</Country:Country_Id>
  <State xmlns:State="http://acme.com/State"
id="fS9JQzdaOVJOSUNVMjBPOQ=="
crc="631C4F4B" status="est">
   <State:State_Id>7Z9RNICU20O9</State:State_Id>
   <State:Country_Id>7Z9RN42HWF3H</State:Country_Id>
   <State:Code>GAU</State:Code>
   <State:Name>Gauteng</State:Name>
  </State>
  <State xmlns:State="http://acme.com/State"
id="fS9JQzdaOVJOSUNVMjVNMg=="
crc="8005F953" status="est">
   <State:State_Id>7Z9RNICU25M2</State:State_Id>
   <State:Country_Id>7Z9RN42HWF3H</State:Country_Id>
   <State:Code>MPU</State:Code>
   <State:Name>Mpumalanga</State:Name>
  </State>
  <Country:Code>ZA</Country:Code>
  <Country:Name>South Africa</Country:Name>
  <Country:Has_State>T</Country:Has_State>
 </Country>
 <Country xmlns:Country="http://acme.com/Country"
id="fS9JQzdaOVJONDJIV0tOSQ=="
crc="1AE5F06B" status="est">
  <Country:Country_Id>7Z9RN42HWKNI</Country:Country_Id>
  <State xmlns:State="http://acme.com/State"
id="fS9JQzdaOVJOSUNVMkRTSA=="
crc="47FBA802" status="est">
   <State:State_Id>7Z9RNICU2DSH</State:State_Id>
   <State:Country_Id>7Z9RN42HWKNI</State:Country_Id>
   <State:Code>CH</State:Code>
   <State:Name>Chombe</State:Name>
  </State>
  <Country:Code>ZIM</Country:Code>
  <Country:Name>Zimbabwe</Country:Name>
  <Country:Has_State>T</Country:Has_State>
```

```
 </Country>
</Root>
```

3.4.3 Element declarations for entities

The following DTD declares an entity COUNTRY with the fields: ID, CD, NM, and STATE. COUNTRY contains an inner entity, STATE, which has the following fields: ID, CD, and NM. The declaration takes place in the following steps:

- Declare the element as the child of an existing element.
- Define the content of the element. You must define all the elements that are allowed to occur within the start and end tags of the element.

The relevant lines in the DTD are emphasized in the following DTD extract:

```
<!ELEMENT Root (Country*)>
<!ELEMENT Country (Country:Country_Id, State*, Country:Code,
Country:Name, Country:Has_State)>
<!ATTLIST Country id CDATA #REQUIRED>
<!ATTLIST Country crc CDATA #REQUIRED>
...
<!ELEMENT State (State:State_Id, State:Country_Id, State:Code,
State:Name)>
...
```

Note: The DTD uses logical names for the fields and entities. For more information about the DTD in this example, including information about the mapping structure used for the COUNTRY and STATE entities, see section 3.4.2 Sample DTD and XML stream.

<!ELEMENT Root (Country*)>

This line declares that `Country` is the only child element of the root element, named `Root`. The * operator indicates that the `Country` element can occur zero, once, or many times in the stream.

<!ELEMENT Country (Country:Country_Id, State*, Country:Code, Country:Name, Country:Has_State)>

This line declares the content of the `Country` element.

It also declares that the State element is a child of Country. The * operator for State indicates that State elements can occur zero, once, or many times inside each Country element.

Note: The names of the fields of Country are prefixed with Country: *to ensure that the names of elements are unique in the DTD.*

<!ELEMENT State (State:State_Id, State:Country_Id, State:Code, State:Name)>

This line declares the content of the State element. State does not have any inner entities, so the content of State is exclusively a list of elements representing the fields of State.

3.4.4 Unique element names and namespaces

The names of all elements in an XML stream must be unique. For example, the entities STATE and COUNTRY both have an ID field. A different element must be used for ID.STATE and ID.COUNTRY, as shown in the DTD below: ID.STATE is represented by State:State_Id, and ID.COUNTRY is represented by Country:Country_Id.

For more information about the DTD used in this example, see section 3.4.2 *Sample DTD and XML stream*.

Namespaces

As an additional step to ensure that all element names are unique in the DTD, the elements representing fields are prefixed with the name of their parent element (that is, the name of the element representing their entity).

You can use this naming convention informally, by defining the element names with an *Entity*: prefix, or you can formally define the entity name prefix as a namespace using the syntax shown below. For more information about namespaces and the XML standard, see the World Wide Web Consortium.

```
<!ELEMENT Root (Country*)>

<!ELEMENT Country (Country:Country_Id, State*, Country:Code, Country:Name,
Country:Has_State)>

<!ATTLIST Country id CDATA #REQUIRED>
```

```
<!ATTLIST Country crc CDATA #REQUIRED>
<!ATTLIST Country status CDATA #REQUIRED>
<!ATTLIST Country valerr CDATA #IMPLIED>
<!ATTLIST Country xmlns:Country CDATA #FIXED "http://acme.com/
Country">
<!ELEMENT Country:Country_Id (#PCDATA)>
<!ATTLIST Country:Country_Id valerr CDATA #IMPLIED>

<!ELEMENT State (State:State_Id, State:Country_Id, State:Code, State:Name)>
<!ATTLIST State id CDATA #REQUIRED>
<!ATTLIST State crc CDATA #REQUIRED>
...
```

3.4.5 DTDs and mapping

The following DTDs describe XML streams that can be used with the CAT.ART entity.

UNIFACE can require additional information to map elements in a DTD to field and entity names, if the element names do not match qualified or unqualified field and entity names. This information is defined in a mapping list.

CAT.ART

CAT.ART is an entity describing categories of products in a catalog. The entity has the following fields:

- ID—a technical primary key.
  For example, "qwERTy12345".
- CD—a semantic candidate key.
  For example, "MSCPUMPS".
- DESCR—a description field.
  For example, 'Multi-speed centrifugal pumps and water extraction systems'.

BASICDTD.ART

This DTD has been generated by the DTD Wizard in the Development Environment.

No mapping is required between the DTD and the fields of CAT.ART because the element names match field and entity names.

```
<!ELEMENT BASICDTD (CAT*)>
<!ELEMENT CAT (ID, CD, DESCR)>
<!ATTLIST CAT id CDATA #REQUIRED>
<!ATTLIST CAT crc CDATA #REQUIRED>
<!ATTLIST CAT status CDATA #REQUIRED>
<!ATTLIST CAT valerr CDATA #IMPLIED>
<!ELEMENT ID (#PCDATA)>
<!ATTLIST ID valerr CDATA #IMPLIED>
<!ELEMENT CD (#PCDATA)>
<!ATTLIST CD valerr CDATA #IMPLIED>
<!ELEMENT DESCR (#PCDATA)>
<!ATTLIST DESCR valerr CDATA #IMPLIED>
```

BASIC_NO_ATTSDTD.ART

If the CAT.ART entity is read only (for example, the entity is set to No Updates in the application model), there is no need to include any of the UNIFACE-generated attributes with the XML stream. In this case, the BASIC_NO_ATTSDTD.ART DTD could be used:

```
<!ELEMENT BASIC_NO_ATTSDTD (CAT*)>
<!ELEMENT CAT (ID, CD, DESCR)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT CD (#PCDATA)>
<!ELEMENT DESCR (#PCDATA)>
```

RENAMEDDTD.ART

All the elements in this DTD have been renamed. This can be done to enhance the readability of the DTD, or to separate the DTD from component field and entity names.

```
<!ELEMENT RENAMEDDTD (category*)>
<!ELEMENT category (category_id, category_code, description)>
<!ATTLIST category id CDATA #REQUIRED>
<!ATTLIST category crc CDATA #REQUIRED>
<!ATTLIST category status CDATA #REQUIRED>
<!ATTLIST category valerr CDATA #IMPLIED>
<!ELEMENT category_id (#PCDATA)>
```

```
<!ATTLIST category_id valerr CDATA #IMPLIED>
<!ELEMENT category_code (#PCDATA)>
<!ATTLIST category_code valerr CDATA #IMPLIED>
<!ELEMENT description (#PCDATA)>
<!ATTLIST description valerr CDATA #IMPLIED>
```

Mapping is required between elements in this DTD and the fields of CAT.ART because the element names do not match the field names:

```
$MAPPING$ = "category=CAT.ART;category_id=ID.CAT;
category_code=CD.CAT;description=DESCR.CAT"
```

This mapping can be used as follows:

```
xmlload XML_PARAMETER, "DTD:RENAMEDDTD.ART", $MAPPING$
```

CAT.PRODUCTS

CAT.PRODUCTS is an entity in another application model with the same purpose as CAT.ART. CAT.PRODUCTS has the following fields:

- PK—the technical primary key
- ID—the category code
- CAT_TEXT—a description field
- CAT_ICON—a small icon used to identify a category

Note: If a component containing CAT.PRODUCTS received an XML stream using the BASICDTD.ART DTD, the ID element would be incorrectly mapped to ID.CAT. No other data would be loaded from the XML stream.

To correctly retrieve the data from an XML stream that uses BASICDTD.ART, define a mapping of XML element names to the entity and field names:

```
$MAPPING$ = "CAT=CAT.PRODUCTS;ID=PK;CD=ID;DESCR=CAT_TEXT"
```

This mapping can be used as follows:

```
xmlload XML_PARAMETER, "DTD:BASICDTD.ART", $MAPPING$
```

Chapter 4 XML transformations

The following topics describe how the structure of an XML stream can be manipulated and transformed by the use of XSLT (Extensible Stylesheet Language for Transformations).

XSLT enables you to access the data in any XML stream, and also provides a standard technique for converting XML data into other formats, such as text, HTML, and lists. XSLT also enables you to convert XML data produced by UNIFACE into other XML formats. This provides support for XML standards, such as B2B messaging standards.

4.1 XSLT (XSL Transformations)

The Extensible Stylesheet Language for Transformations (XSLT) is an XML-based language developed by the World Wide Web Consortium. XSLT provides a standard language for expressing transformations to the structure of XML streams. Given a well-formed XML stream as input, XSLT can be used to construct an output, either in XML or other character-based output formats.

The following transformations are typical of the uses of XSLT:

- Transforming XML into HTML for display on the Web
- Transforming XML into text formats, such as comma-separated lists
- Transforming XML into a new XML structure. This includes:
  - Altering nodes in the XML stream. For example, converting elements to attributes, or adding or removing a node.
  - Changing the precedence of items in the XML stream. For example, sorting the content of an XML stream alphabetically.

4.2 UNIFACE XSLT tools and components

UNIFACE provides component USYSXSLT to transform XML streams at run time.

XSLT files can be edited and tested using the XSLT Workbench.

4.3 How XSLT works

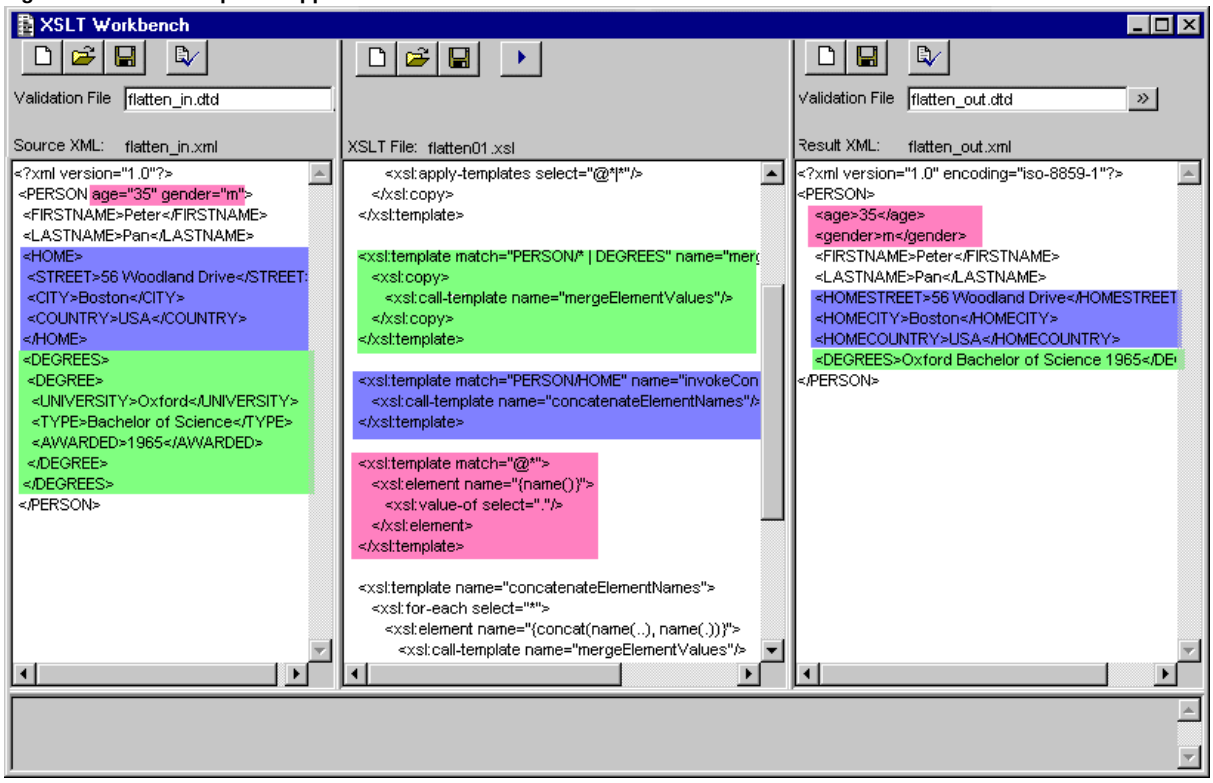
XSLT is a rule-based language. An XSLT processor matches patterns in the input XML stream with rules in the XSLT stylesheet, and applies an output template specified by the rule. For example, to find an attribute named `myAttribute` and output the attribute's value in an element named `<myElement>`, you could use the following template:

```
<xsl:template match="@myAttribute"> <!--Apply this template to myAttribute. -->
   <xsl:element name="myElement">  <!--Create element myElement. -->
      <xsl:value-of select="."/>   <!--Insert
the value of myAttribute.-->
   </xsl:element>
</xsl:template>                    <!--End
of template.-->
```

This XSLT template creates the following fragment of XML:

...<myElement>*AttributeValue*</myElement>...

Figure 4-1 XSLT templates applied to an XML stream.



The following transformations are highlighted in the illustration:

- The age and gender attributes are converted to elements <age> and <gender>.
- In the source XML, element <HOME> contains <STREET>, <CITY>, and <COUNTRY>. In the Result XML pane, the <HOME> element is removed, and the child elements of <HOME> are output as <HOMESTREET>, <HOMECITY> and <HOMECOUNTRY>.
- The content of source element <DEGREES> is output as a single element value. The same template also outputs the <FIRSTNAME> and <LASTNAME> elements (an XSLT template can apply to many parts of a source XML stream).

4.3.1 Learning more about XSLT

XSLT is a rich language with many statements and functions. The language is unique in several respects, such as its treatment of variables, the data types allowed, and the handling of processing flow. As such it is recommended to consult specialist sources devoted to XSLT.

The specification for XSLT is issued by the World Wide Web Consortium.

For examples of XSLT stylesheets, see XMLTRANSFORM in the UNIFACE Library.

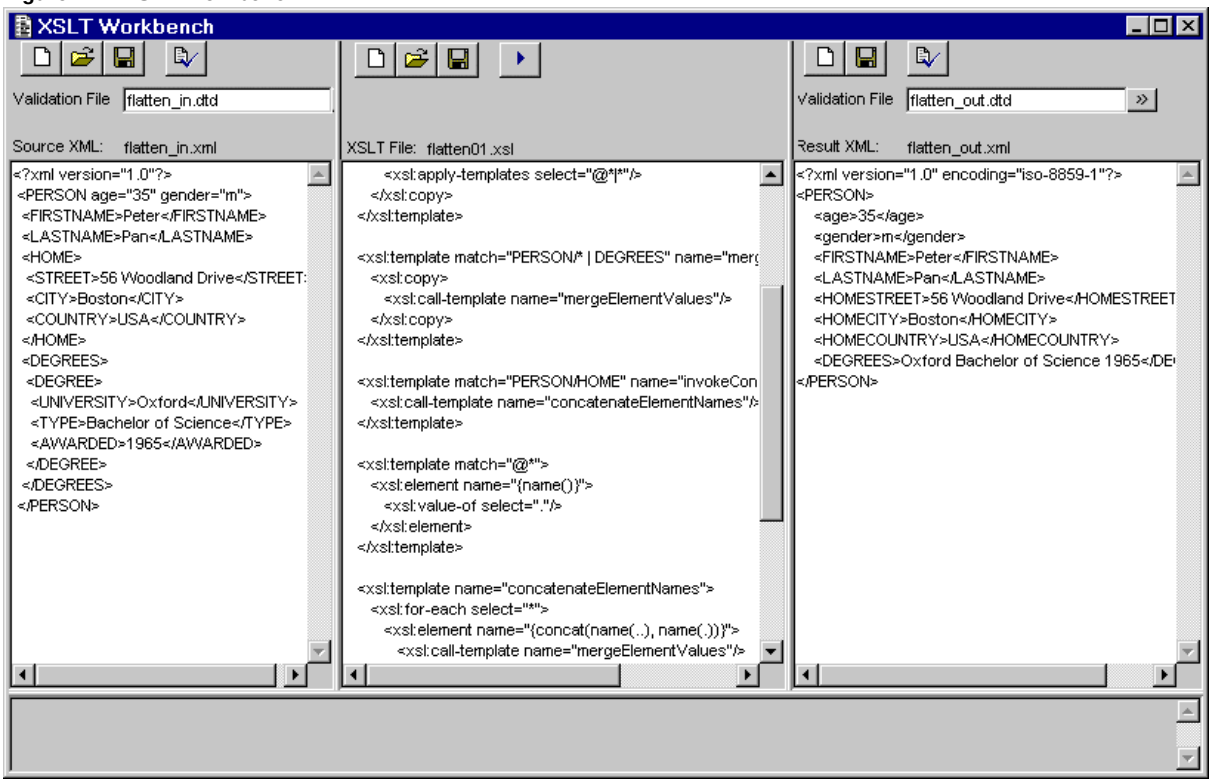
4.4 XSLT Workbench

The XSLT Workbench is a tool for creating and testing XSLT stylesheets. The XSLT Workbench is divided into the following sections:

- *Source XML*—displays the test data for testing the XSLT file

- *XSLT File*—displays the XSLT file
- *Result XML*—displays the XML resulting from running the transformation

Figure 4-2 XSLT Workbench.



Source XML pane

Use this part of the XSLT Workbench to create, load, edit and save XML files (with, by default, an **`.xml`** file name extension). When an XSLT file is tested, the XML file in this pane is used as an input stream.

Clicking Validate validates the Source XML, using the validation file specified in the Validation File field.

XSLT File pane

Use this part of the XSLT Workbench to create, load, edit, and save XSLT files (with, by default, an **.xsl** file name extension). This pane also provides button Test XSLT to activate the XSLT processor component USYSXSLT.

The *Test XSLT* button transforms the *Source XML* using the *XSLT File*, and display the results in the *Result XML* pane. (The USYSXSLT component is supplied with the XML stream loaded into the Workbench, and the path to the XSLT file.)

Result XML pane

This pane displays the result of a transformation. Normally the result is another XML file, but could be another text-based format. This pane allows you to save the transformation output as a file, and to clear the screen.

Clicking Validate validates the Result XML, using the validation file specified in the Validation File field.

Note: When using menu commands to save, close or open files, remember that the menu command applies to the pane that currently has focus.

4.5 USYSXSLT

USYSXSLT is a UNIFACE component that provides services for the transformation of XML streams using XSLT stylesheets.

USYSXSLT uses the Xalan XSLT processor from the Apache Software Foundation. Xalan is a Java-based application that implements the XSLT specification issued by the World Wide Web Consortium.

System requirements

USYSXSLT has the following system requirements:

- Operating system—UNIX, or Microsoft Windows (versions 95, 98, 2000 and NT)
- Java runtime environment—Sun Java Runtime, versions 1.2.2 or higher.

For your convenience, the Sun Java Runtime environment is included on your UNIFACE CD (Microsoft Windows installations only).

Note: On Microsoft Windows systems, ensure that the path to JVM.dll is declared in the PATHS environment variable.

Signature

USYSXSLT provides operation XMLTRANSFORM for transforming XML streams.

Remote execution of USYSXSLT

You can specify remote execution of the USYSXSLT component in your assignment file. For more information, see Assignments for remote services and reports in the UNIFACE Library.

4.6 Basic XSLT techniques—examples

The following examples show the techniques required in most common XSLT transformations:

4.6.1 XSLT—rename elements and attributes

In many cases, elements and attributes have the same semantic meaning in two different DTDs but are named differently. You must therefore create an XSLT script to rename the elements and attributes of one DTD to match the naming conventions in another DTD.

Input stream

The input stream uses the element `actor` and attribute `role`:

```
<?xml version="1.0" ?>
<actor role="innocent bystander">Peter</actor>
```

Output stream

The output stream requires the element `person` and attribute `type`:

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

```
<person type="innocent bystander">Peter</person>
```

XSLT stylesheet

Renaming `actor` to `person`, and `role` to `type`, can be done with the following XSLT stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" />
<xsl:template match="/" name="rootTemplate" >
   <xsl:apply-templates select="actor" />
</xsl:template>
<xsl:template match="actor" name="actor-to-person" >
   <xsl:element name="person">
      <xsl:attribute name="type">
         <xsl:value-of select="@role" />
      </xsl:attribute>
      <xsl:value-of select="." />
   </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

Literal result elements

Template `actor-to-person` can be written in a much more compact form, using literal result elements. Literal result elements are any non-XSLT elements inside a template body. Literal result elements are simply copied to the output stream by the XSLT processor.

You can also set attribute values for literal result elements, using curly braces ({ and }) to indicate that the attribute value is not literal, but derived. These techniques are shown in the following alternative `actor-to-person` template:

```
<xsl:template match="actor" name="actor-to-person" >
   <person type="{@role}" >
      <xsl:value-of select="." />
   </person>
</xsl:template>
```

4.6.2 XSLT—change attributes to elements

Since UNIFACE does not read data from attributes[1], it is almost always necessary to map an attribute to an element before an `xmlload`, and to map an element to an attribute after an `xmlsave`.

Input stream

The input stream has an attribute `role` (of the element `actor`):

```
<?xml version="1.0" ?>
<actor role="innocent bystander">Peter</actor>
```

Ouput stream

The output stream requires elements `type` and `name` (as a child of element `person`):

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
 <name>Peter</name>
 <type>innocent bystander</type>
</person>
```

XSLT stylesheet

Use the following XSLT stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="xml" />
<xsl:template match="/" name="root" >
   <xsl:apply-templates select="actor" />
</xsl:template>
<xsl:template match="actor" name="actor-to-person" >
   <person>
      <name>
         <xsl:value-of select="." />
      </name>
```

1. With the exception of the processing information attributes—`id`, `crc`, `status` and `valerr` —that UNIFACE creates to support disconnected record sets.

```
   <type>
      <xsl:value-of select="@role" />
   </type>
   </person>
</xsl:template>
</xsl:stylesheet>
```

4.6.3 XSLT—reorder elements

In the simplest case, reordering elements means putting element B before element A. In this example, you are restructuring the tree by creating or removing nodes. This kind of reordering is needed when an attribute is mapped to an element and the text node of the attribute's element should be mapped to a child element.

Input stream

The following XML stream stores the actor name as the content of element <actor>:

```
<?xml version="1.0" ?>
<actor role="innocent bystander">Peter</actor>
```

Output stream

The output stream requires that the actor name should be contained in an element <name>:

```
<?xml version="1.0" encoding="UTF-8"?>
<person><name>Peter</name>
<type>innocent bystander</type></person>
```

XSLT stylesheet

Use the following XSLT stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="xml" />
<xsl:template match="/" name="root" >
   <xsl:apply-templates select="actor" />
```

```
</xsl:template>
<xsl:template match="actor" name="actor-to-person" >
   <person>
   <name>
      <xsl:value-of select="." />
   </name>
   <type>
      <xsl:value-of select="@role" />
   </type>
   </person>
</xsl:template>
</xsl:stylesheet>
```

4.6.4 XSLT—suppress empty elements or attribute

An element always overwrites the value of the field to which it is mapped in UNIFACE. If no element is mapped to a field, the current value of the field is preserved, even if other fields in the same occurrence are modified by data from the stream. Therefore, it can be useful to exclude empty elements from a stream if you want to preserve default values, or merge subsets of records.

The simplest method is to suppress the output when the source element or attribute is empty.

Input stream

The attribute `role` in the following XML stream should be suppressed in the output stream when it is empty. Otherwise it should be mapped to the element `type`.

```
<?xml version="1.0" ?>
<actors>
   <actor role="">Peter</actor>
   <actor role="star">Greta</actor>
</actors>
```

Output stream

The following output is required:

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
   <person><name>Peter</name></person>
   <person><name>Peter</name><type>star</type></person>
</persons>
```

Note: The attribute `role` *has been suppressed in the first occurrence of* `<person>`*, while it has been mapped to element* `type` *in the second occurrence.*

XSLT stylesheet

Use the following XSLT stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="xml" />
<xsl:template match="/" name="root" >
    <persons>
    <xsl:apply-templates select="//actor" />
    </persons>
  </xsl:template>
<xsl:template match="actor[@role='']"
              name="supress-output-when-empty" >
   <person>
   <name>
      <xsl:value-of select="." />
   </name>
   </person>
</xsl:template>
<xsl:template match="actor" name="actor-to-person" >
   <person>
   <name>
      <xsl:value-of select="." />
   </name>
   <type>
      <xsl:value-of select="@role" />
   </type>
```

```
   </person>
</xsl:template>
</xsl:stylesheet>
```

4.6.5 XSLT—implement exclusive OR relationship

A DTD can specify that it expects element A *or* element B as a child of another element, but not both. For example, a security implementation only accepts an encrypted password or a codeword, but not both. This constraint is described by the following DTD:

```
<!ELEMENT security          (encryptedpassword | codeword) >
<!ELEMENT encryptedpassword ( #PCDATA ) >
<!ELEMENT codeword          ( #PCDATA ) >
```

The XSLT stylesheet must be able to either handle the `encryptedpassword` and `codeword` elements differently, or transform them to the same output item. In this example, both of these alternative elements are transformed to the same output element.

Input stream

The following source XML stream conforms to the DTD:

```
<?xml version="1.0" ?>
<security>
  <encryptedpassword>4</encryptedpassword>
</security>
```

You should validate the XML stream against the DTD to confirm the XML stream contains either a password or a codeword. Use the Proc statement `xmlvalidate` for this purpose.

Output stream

The following output is required:

```
<?xml version="1.0" encoding="UTF-8"?>
<ENTITY.MODEL><FIELD>4</FIELD></ENTITY.MODEL>
```

Note: This output stream can be loaded by a UNIFACE component without the need for specifying a mapping between element and field names, if the entity named ENTITY.MODEL is painted on the component, and the ENTITY field list includes a field named FIELD.

XSLT stylesheet

Use the following XSLT stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="xml" />
<xsl:template match="/" name="root-template" >
   <ENTITY.MODEL>
      <xsl:apply-templates select="security/*" />
   </ENTITY.MODEL>
</xsl:template>
<xsl:template match="encryptedpassword |codeword"
              name="xor-to-FIELD.ENTITY.MODEL" >
 <FIELD>
    <xsl:value-of select="." />
 </FIELD>
</xsl:template>
</xsl:stylesheet>
```

4.7 XSLT applied to UNIFACE—examples

The following examples show some applications of XSLT within UNIFACE:

4.7.1 B2B XSLT stylesheets

UNIFACE uses XSLT stylesheets to transport data to and from B2B messages. **`cxmlmes2u.xsl`** converts Ariba B2B messages for processing by UNIFACE. **`u2cxmlmes.xsl`** converts UNIFACE B2B message into Ariba format. These XSLT stylesheets are provided with your UNIFACE installation.

These stylesheets also provide examples of the application of XSLT to inter-component communication. This example describes some of the techniques used in these stylesheets.

cxmlmes2u.xsl

This stylesheet removes the need to do default or local mapping of element names to field names, by transforming the input XML stream to a structure conforming to UNIFACE's default naming structure for elements: <FIELD.ENTITY.MODEL>. For example, the following template from **cxmlmes2u.xsl** converts attribute domain to element <AT_DOMAIN.TCREDENTIAL.U_V8B2B>:

```
<xsl:template name="to_domain">
  <AT_DOMAIN.TCREDENTIAL.U_V8B2B>
    <xsl:value-of select="//Header/To/Credential/@domain"/>
  </AT_DOMAIN.TCREDENTIAL.U_V8B2B>
</xsl:template>
```

The stylesheet uses an identity transformation template to copy sub-trees from the input XML stream to the output stream. This template applies to all nodes in the input stream, so if there is no other template with a more exact match for the current node, the identity transformation template is applied instead (the XSLT standard defines rules for the precedence of templates, so a template matching a wildcard, such as "*", has lower precedence than a template matching a specific element name, such as "From").

```
<!-- Identity transformation template -->
  <xsl:template match="*|@*|comment()|processing-instruction()|text()">
    <xsl:copy>
      <xsl:apply-templates select="*|@*|comment()|processing-
instruction()|text()"/>
    </xsl:copy>
  </xsl:template>
</xsl:transform>
```

The identity transformation template is particularly useful for B2B stylesheets, because this allows the application to convert and process the B2B envelope, and then process the B2B body in a separate phase.

u2cxmlmes.xsl

XSLT stylesheets are not 'reversible' so a separate stylesheet is defined to convert the data from the format used by UNIFACE B2B to a format suitable for Ariba. The following template transfers element values to the attributes required by Ariba:

```
<!-- Composed element template -->
  <xsl:template match="CXML.U_V8B2B">
```

```
    <cXML>
      <xsl:attribute name="version">
        <xsl:value-of select="AT_VERSION/text()"/>
      </xsl:attribute>
      <xsl:attribute name="xml:lang">
        <xsl:value-of select="AT_LANGUAGE/text()"/>
      </xsl:attribute>
...
```

4.7.2 Transform an XML stream

This example uses an XSLT file to *normalize* an XML stream so that all the information in the stream can be accessed by the `xmlload` statement. This normalization can be necessary because of the divergence between the tree structure inherent in XML and the relational model on which UNIFACE is based. For more information, see section 1.2.5 *Document Type Definition (DTD)* .

The input XML stream `I_INPUT` has the following content:

```
<?xml version="1.0"?>
<team id="dfgty4yghr67" crc= "" status="new" >
 <id>10001</id>
 <name>Williams</name>
  <driver id="1223werwr44353456456" crc= "" status="new" age="35" gender="m">
   <id>10002</id>
   <name>Amal</name>
  </driver>
 <car id="dyt4645yrhg567" crc= "" status="new" >
   <id>10003</id>
  <name>Unknown</name>
  <manufacturer>Ford</manufacturer>
 </car>
</team>
```

This stream contains the following information that UNIFACE cannot load directly into components:

- Information in attributes (such as `age="35"`), which UNIFACE does not interpret.
- The `id` and `name` elements occur as the children of elements `team`, `driver`, and `car`. Elements occurring as children of more than one other element are also not supported by UNIFACE.

Finally, elements `team`, `driver` and `car` have UNIFACE processing information attributes that must be preserved in the output stream.

The target output XML stream, `I_NORMALIZED` is the following (transformed items are in **`bold`**):

```
<?xml version="1.0" encoding="utf-8" ?>
<team crc="" id="dfgty4yghr67" status="new">
 <team-id>10001</team-id>
 <team-name>Williams</team-name>
  <driver crc="" id="1223werwr44353456456" status="new">
   <age>35</age>
   <gender>m</gender>
   <driver-id>10002</driver-id>
   <driver-name>Amal</driver-name>
  </driver>
 <car crc="" id="dyt4645yrhg567" status="new">
   <car-id>10003</car-id>
  <car-name>Unknown</car-name>
  <manufacturer>Ford</manufacturer>
 </car>
</team>
```

The following XSLT instructions are used to achieve this transformation:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<!-- Copy elements to the output stream. -->
   <xsl:template match="*" name="main">
      <xsl:copy>

         <!-- The following attributes are to be kept as attributes. -->
         <!-- First they are copied to the output stream, and then -->
         <!-- template disableTemplates is applied to the attributes -->
         <!-- to prevent application of template elementsToAttributes. -->
```

```
         <xsl:copy-of select="@id | @crc | @valerr | @status"/>

         <!-- Copy elements, rename non-unique elements, convert attributes to
elements. -->
         <xsl:apply-templates select="@*|node()"/>

      </xsl:copy>
   </xsl:template>

<!-- Handle attributes by converting them to child elements. -->
   <xsl:template match="@*" name="elementsToAttributes">
      <xsl:element name="{name()}">
         <xsl:value-of select="."/>
      </xsl:element>
   </xsl:template>

<!-- Make selected element names unique by concatenating parent name. -->
<!-- Modify the 'match' attribute to change the list of elements -->
<!-- requiring concatenation using the XPATH //*/NewName for each element. -->
   <xsl:template match="//*/name | //*/id" name="concatenateElementNames">
      <xsl:element name="{concat(name(..), '-', name())}">
         <xsl:value-of select="."/>
      </xsl:element>
   </xsl:template>

<!-- Disable output of these items, excepting output as the result of an
xsl:copy -->
<!-- or xsl:copy-of instruction. -->
   <xsl:template match="@id | @crc | @valerr | @status" name="disableTemplates"/
>

</xsl:stylesheet>
```

The following Proc can be used to effect the normalization of the input XML stream (assuming that the XSLT instructions are contained in a file **`normalize.xsl`**):

```
operation XNORMALIZE
params
   numeric  I_STATUS          : OUT
```

```
   string   I_STATUSCONTEXT    : OUT
   string   I_INPUT            : IN
   string   I_ARGUMENTS        : IN
   string   I_NORMALIZED       : IN
endparams

activate "USYSXSLT".XMLTRANSFORM (I_INPUT, "normalize.xsl", I_NORMALIZED,
I_ARGUMENTS, I_STATUS, I_STATUSCONTEXT)
; Handle activation errors HERE, by evaluating $status and $procerror.
; Handle errors that occured during validation HERE, by
; evaluating the expressions of I_STATUS and I_STATUSCONTEXT.
end ; operation XNORMALIZE
```

Note: The XSLT instructions in this example are generic; this means the stylesheet can be applied to any XML stream containing non-unique child element names, or data in attribute values. However, mixed content (where an element contains a mixture of text data and child elements) is not handled by this transformation. To use this stylesheet on an XML stream containing mixed content, add the elements with mixed content to the `match` *attribute of template* `disableTemplate`*s. This excludes the mixed content from the output XML stream.*

4.7.3 Get values from an XML stream using XSLT

It is possible to use XSLT to get single values or groups of values from an XML stream. The following XSLT instructions can be used for this:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="text"/>
<xsl:strip-space elements="*"/>

<!-- Disable output of these nodes (recursive). -->
   <xsl:template match="*" name="outputNothing">
    <xsl:apply-templates select="*"/>
   </xsl:template>

<!-- Output the value of this node. -->
```

```
   <xsl:template match="TargetNode" name="outputValueOf">
      <xsl:value-of select="TargetValue"/>
   </xsl:template>

</xsl:stylesheet>
```

The <xsl:output> element specifies that the output format of the stylesheet is text, not XML. This means that XML entities are replaced by their replacement texts, so & is output as &. The <xsl:strip-space> element controls which white space nodes are removed from the XML stream. This example removes all white space nodes.

Template outputNothing recursively steps through all the elements in the XML stream, and assigns nothing to output. This template produces empty output.

Template outputValueOf matches a specific element or set of elements specified by *TargetNode*, and outputs related values specified in *TargetValue*. The match attribute of outputValueOf is more exact than that of outputNothing, so the XSLT processor applies outputValueOf template preferentially.

The following XML stream is processed using this stylesheet:

```
<?xml version="1.0"?>
<A>
 <A1>abc</A1>
 <A2>def</A2>
  <B>
   <B1>ghi</B1>
   <B2>jkl</B2>
  </B>
 <C>
  <C1>mno</C1>
  <C1>pqr</C1>
  <C1>stu</C1>
 </C>
 <C>
  <C1 att="goodbye ">vwx</C1>
  <C1 att="cruel ">yz1</C1>
  <C1 att="world">234</C1>
 </C>
 <C att="hello " att2="world">
```

```
  <C1>567</C1>
  <C1>8</C1>
  <C1>9</C1>
 </C>
 <D><!-- comment. -->Mixed content <D1>element</D1> example.</D>
</A>
```

The following table shows the values output from the XML stream after it is processed using the XSLT stylesheet.

Table 4-1 Conversion of UNIFACE data types to PCDATA in XML streams. ***part 1 of 2***

TargetNode and TargetValue	Return value
TargetNode: `*` *TagetValue*: `.` Return the value of all elements. Attribute values, comments, and processing instructions are not matched by *TargetNode*, and are therefore not returned.	`abcdefghijklmnopqrstuvwxyz1 23456789Mixed content element example.`
TargetNode: `C1` *TargetValue*: `.` Return the values of all <C1> elements.	`mnopqrstuvwxyz123456789`
TargetNode: `C1[1]` *TargetValue*: `.` Return the value of the first <C1> child of any element (this matches three <C1> elements).	`mnovwx567`
TargetNode: `C[2]/C1[3]` *TargetValue*: `.` Return the value of the third <C1> child of the second <C> element.	`234`
TargetNode: `D | D1` *TargetValue*: `.` Return the value of mixed content element <D> and child element <D1>.	`Mixed content element example.`

Table 4-1 Conversion of UNIFACE data types to PCDATA in XML streams. ***part 2 of 2***

TargetNode and TargetValue	Return value
TargetNode: `C1` *TargetValue*: `./@att` Return the values of the `att` attributes of the <C1> elements.	`goodbye cruel world`
TargetNode: `C[3]` *TargetValue*: `./@*` Output the value of the first attribute of the third <C> element (the *TargetValue* matches all attributes of the <C> element, but <xsl:value-of> only outputs the first node that matches *TargetValue*).	`hello`
TargetNode: `D` *TargetValue*: `./text()[2]` Output the second text node of element <D>. *TargetNode* matches all <D> elements in the stream, so a more exact expression would be required for *TargetNode* if the stream contained more than one <D> element.	`example.`
TargetNode: `D` *TargetValue*: `./comment()` Output the first comment child of element <D>.	`comment.`

For an example of an operation wrapping this stylesheet, see section 4.7.4 *Operation GETXMLITEM*.

4.7.4 Operation GETXMLITEM

This operation calls an XSLT stylesheet and returns the value of a node in an XML stream. The parameters can specify one node or several nodes.

GETXMLITEM uses $replace to insert parameters XNODE and XVALUE into an XSLT stylesheet, and uses component USYSXSLT to process the XSLT stylesheet. For more information about the XSLT stylesheet used by this operation, and examples of input and output data, see section 4.7.3 *Get values from an XML stream using XSLT*.

```
operation GETXMLITEM

; Get an item or set of items from an XML stream
; using XPATH patterns.
; Modify an XSLT stylesheet using the $replace
; function, and then activate USYSXSLT.

params
   numeric I_STATUS        : OUT
   string  I_STATUSCONTEXT : OUT
   string  I_XML           : IN
   string  XNODE           : IN
   string  XVALUE          : IN
   string  I_VALUE         : OUT
endparams

variables
   string  XSLTFILE
endvariables

fileload "getitem.xsl", XSLTFILE

XSLTFILE = $replace(XSLTFILE, 1, "TargetNode", XNODE)
XSLTFILE = $replace(XSLTFILE, 1, "TargetValue", XVALUE)

filedump XSLTFILE, "getitemtemp.xsl"

activate "USYSXSLT".XMLTRANSFORM(I_XML, "getitemtemp.xsl", " ", I_VALUE,
I_STATUS, I_STATUSCONTEXT)

end ; operation GETXMLITEM
```

4.8 Validation—examples

It is often necessary to validate an XML stream before or after transformation.

4.8.1 Validate an XML stream

You can use the Proc statement `xmlvalidate` to validate an XML stream, even if the DTD for the XML stream is not in your Repository.

The following operation `XVALIDATE` validates an XML stream using `xmlvalidate`:

```
operation XVALIDATE
params
   numeric I_STATUS          : OUT
   string  I_STATUSCONTEXT   : OUT
   string  I_DTD             : IN
   string  I_XML             : IN
endparams

; I_DTD is a file path; use the argument /file
; to indicate this to xmlvalidate.
xmlvalidate/file I_XML, I_DTD
I_STATUS = $procerror
I_STATUSCONTEXT = $procerrorcontext
end ; operation XVALIDATE
```

Chapter 5 Handling communication between components

Components communicate via the parameters of their operations. UNIFACE provides a range of data types for parameters, including occurrence and entity parameters, and basic data types such as numeric and string parameters.

UNIFACE provides XML stream parameters, which enable components to exchange structured data and disconnected record sets using XML. The structure of an XML stream is described by a DTD, defined in the DTD Editor. The structure of an XML stream can be transformed using XSLT.

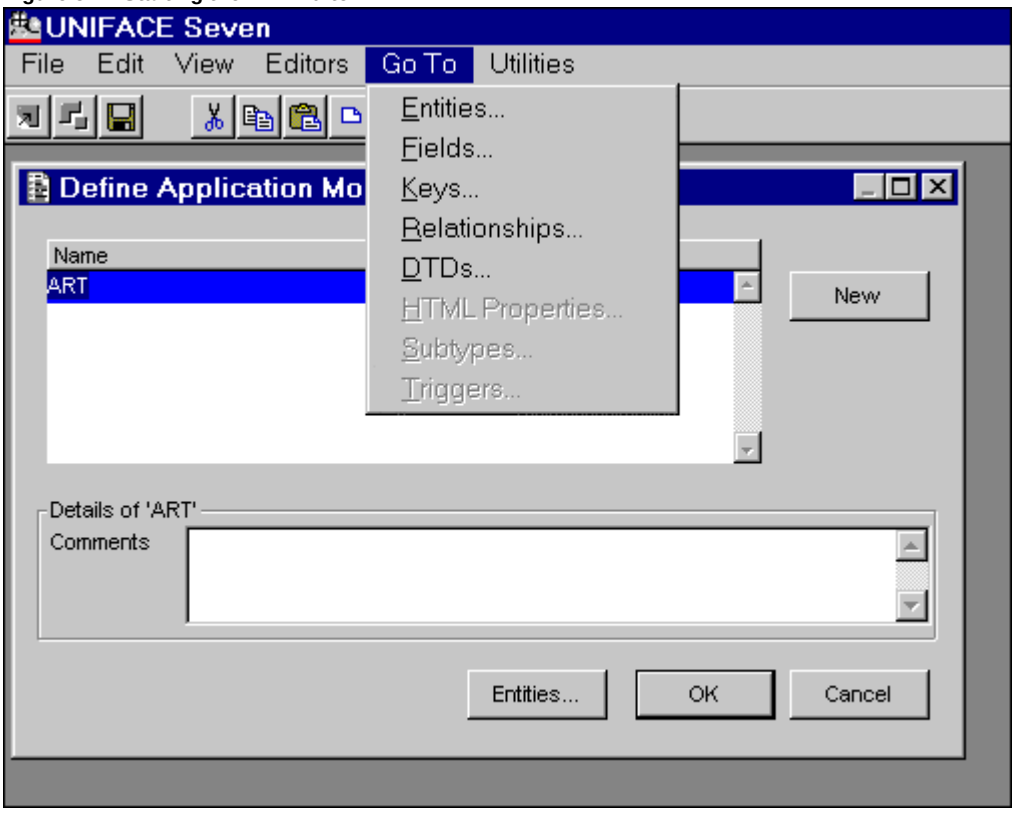
UNIFACE provides Handle parameters for exchanging data between components by reference. Handles enable components to communicate using operations defined on their entities and occurrences.

5.1 Using the DTD Editor

The following is a description of how to create and edit DTDs using the DTD Editor and the DTD Wizard.

5.1.1 Start the DTD Editor

Figure 5-1 Starting the DTD Editor.

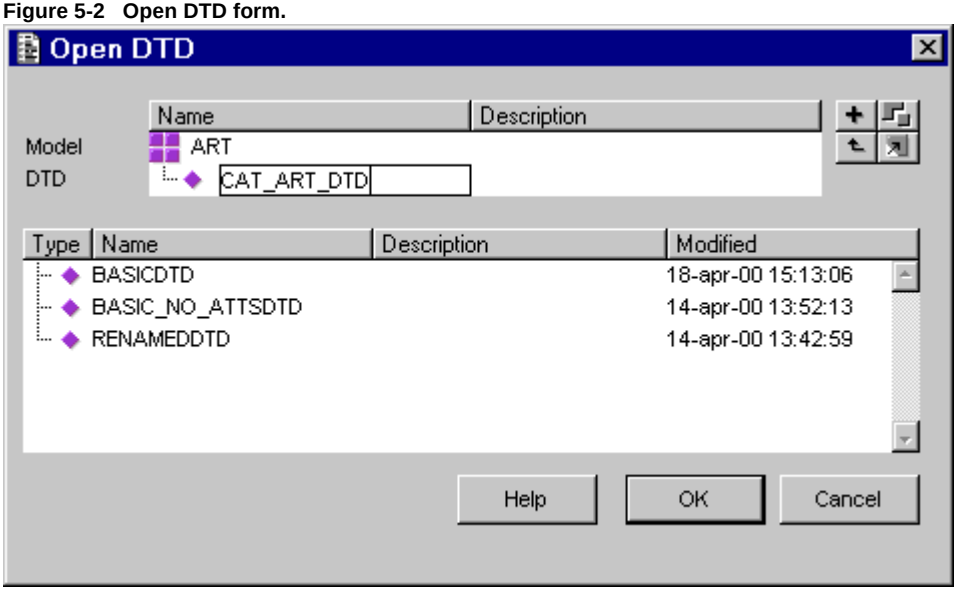


To start the DTD Editor, do the following:

1. Start the UNIFACE Model Editor.
2. Select an application model.

3. Click Go To—>DTDs. This opens the Open DTD form.

Figure 5-2 Open DTD form.



4. Select a DTD or enter the name of a new DTD.
5. Click OK. This opens the Define DTD form of the DTD Editor.

5.1.2 Generate a DTD from a component's structure

To generate a DTD from a component's painted structure of entities and fields, do the following:

1. Start the DTD Editor.
2. Create a new DTD, or open an existing DTD.
3. Select File—>Load Component Structure.
4. Select the component from the list, and click OK.

This generates a DTD that defines an XML stream with a structure corresponding to the field and entity structure from the component.

Generating a DTD from the component's structure overwrites all declarations already stored in the DTD.

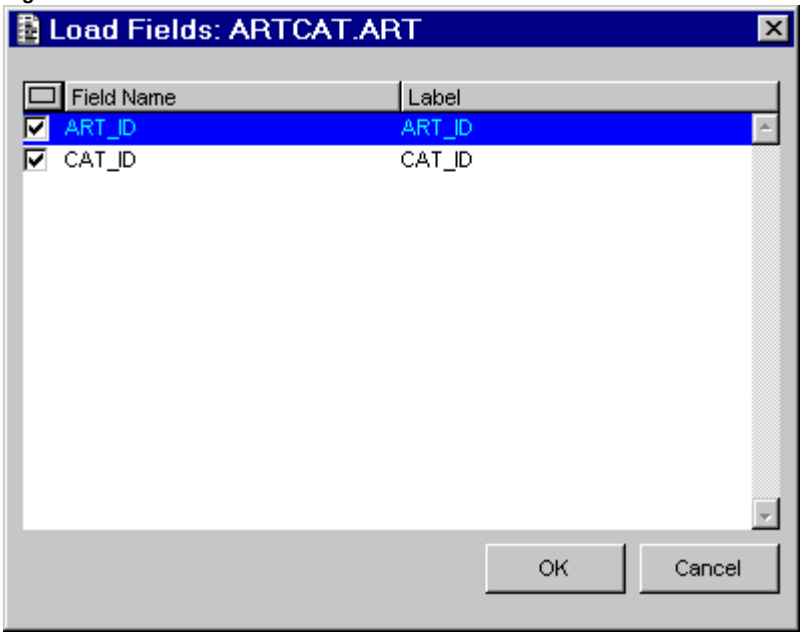
Note: Fields that are not painted on the component are not added to the DTD. Use the DTD Wizard to add other fields from your application model.

5.1.3 Generate DTDs from entities

To generate a DTD from entities defined in your application model, do the following:

1. Start the DTD Editor.
2. Create a new DTD, or open an existing DTD.
3. Click Wizard. This starts the DTD Wizard.
4. Select the root element of the tree.
5. Right-click, and select Insert Entity from the pop-up menu. Select an entity from the list.
6. Click OK.
7. Add fields to the entity by selecting Load Fields from the pop-up menu. Select the fields you need from the list and click OK.

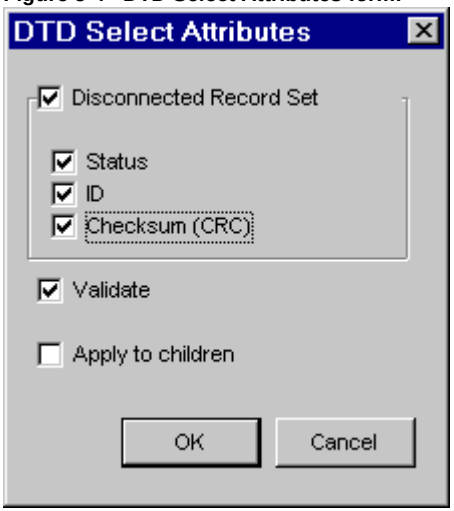
Figure 5-3 Load Fields form.



You can also insert fields individually by selecting Insert Field from the pop-up menu. This gives you the option of entering field names that are not defined for the entity.

8. Add attributes to the elements you have generated by selecting the elements and selecting Select Attributes from the pop-up menu.

Figure 5-4 DTD Select Attributes form.



9. Compile the DTD.

This generates a DTD that defines a structure corresponding to the entity selected in steps 5, 6, and 7.

You can also insert an entity inside another entity. Select the element representing the outer entity, and repeat steps 5 to 8.

Note: You can only insert entities defined in an application model when you are working in the DTD Wizard. To define elements for objects that are not in your application model, edit the DTD in the Define DTD form.

5.1.4 Load a DTD from a file

To load a DTD from a file, do the following:

1. Start the DTD Editor.
2. Click File—>Load from file...

3. Enter the name of the DTD (including the path to the file), or browse to the file.
4. Click Open to load the DTD file into the DTD Editor.

Caution: Clicking Open overwrites the current DTD definition.

5. Select File—>Save or File—>Compile DTD to store the DTD in your Repository.

5.1.5 Define relationships in XML streams

Entities are represented as elements in XML, and inner entities are represented as elements nested within the start and end tags of the outer entity.

To define inner-outer entity relationships in an XML stream, do one of the following:

- Create a component with the relationships painted using frames-within-frames. Generate the DTD from the component structure, as described in section 5.1.2 *Generate a DTD from a component's structure*.
- Load the inner entities using the DTD Wizard, as described in section 5.1.3 *Generate DTDs from entities*.

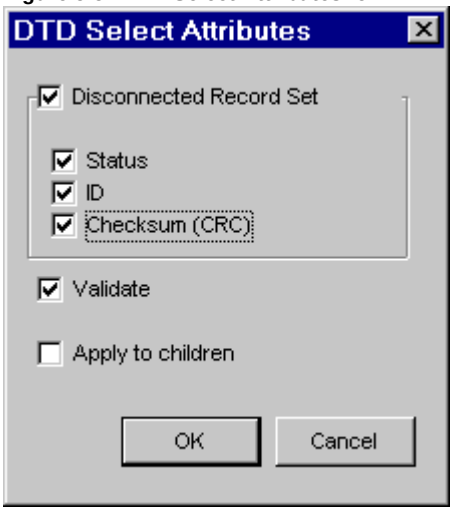
5.1.6 Select attributes for elements

To select attributes for elements, do the following:

1. Open the DTD in the DTD Editor.
2. Click Wizard to start the DTD Wizard.
3. For each element that requires attributes, do the following:
   - Select the element in the DTD Tree on the DTD Wizard form.
   - Right-click on the element, and select Select Attributes from the pop-up menu.

- Select the attributes you require on the Select Attributes form.

Figure 5-5 DTD Select Attributes form.



- Click OK to accept the changes, or cancel the changes by clicking Cancel.

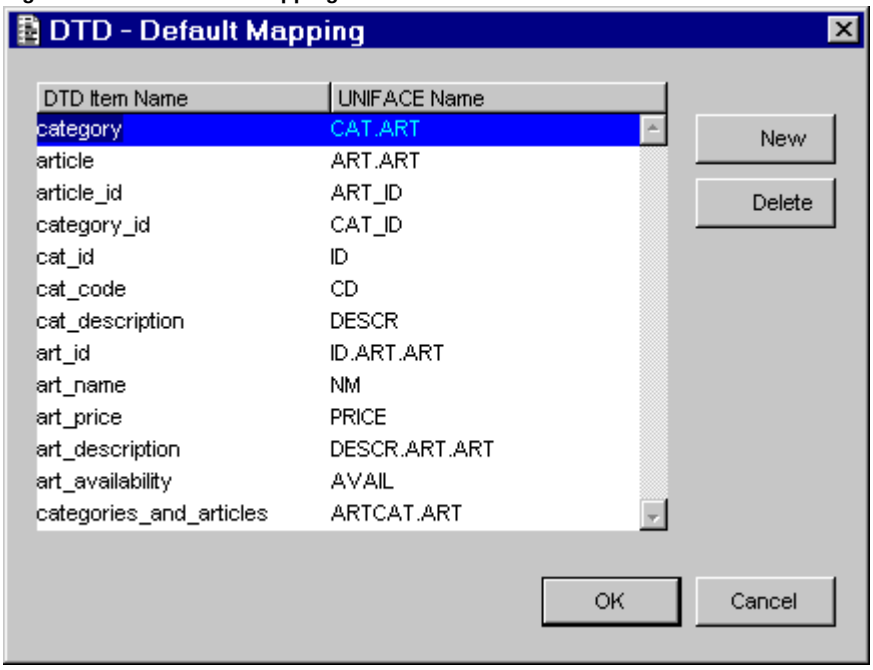
5.1.7 Define a default mapping for a DTD

It can be necessary to tell UNIFACE how to map the elements in an XML stream to fields and entities. You can do this by defining a default mapping for the DTD or by defining a local mapping on a component. For more information about creating a local mapping, see section 5.2.3 *Define a local mapping for a DTD*.

To define a default mapping for a DTD, do the following:

1. Open the DTD in the DTD Editor.
2. Click Mapping. This opens the DTD Default Mapping form.

Figure 5-6 DTD Default Mapping form.



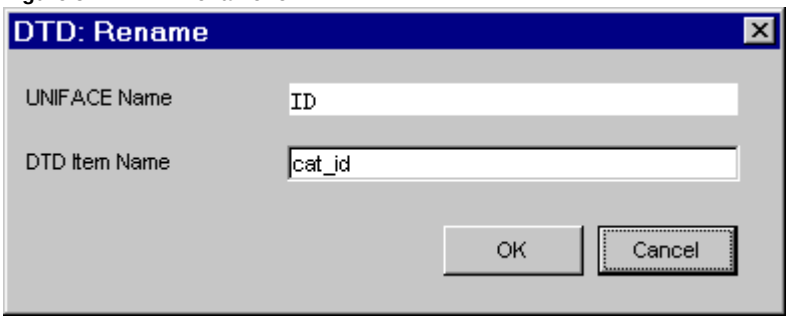
3. For each element you want to map to a UNIFACE field or entity, do the following:
   - Enter the name of the element in the DTD Item Name column.
   - Enter the name of the target field or entity in the UNIFACE Name column. It can be a good practice to use qualified names, such as *Field.Entity*, and *Entity.Model*. If you qualify entity names with model names, then you need to specify a local mapping if the XML stream is also mapped to an identical entity in a different application model.

     For example, if the default mapping specifies that element `country` is mapped to the entity COUNTRY.ORD, you need to specify a local mapping to map the data to the entity COUNTRY.ORD_P.
4. Click OK to accept the mapping definitions, or Cancel to discard the changes.
5. Select File—>Save to store the changes in your Repository.

Creating a default mapping in the DTD Wizard

Whenever you rename an element in the DTD Wizard, a corresponding entry in the default mapping is created or updated.

Figure 5-7 DTD: Rename form.



5.1.8 Define a DTD manually

To define a DTD manually, do the following:

1. An XML stream has the following structure:
   RootDef
   EntityDefs
   FieldDefs
   AttributeDefs
   where:
   - *RootDef* is the definition of the root node of the XML stream
   - *EntityDefs* are the definitions of elements that represent UNIFACE entities
   - *FieldDefs* are the definitions of elements that represent the fields of the UNIFACE entities
   - *AttributeDefs* are the definitions of attributes for elements

   The following steps describe how to define root, entity and field elements, and attributes.
2. All XML streams have a root node, which encloses all the other elements in the stream. The definition of the root element is always the first definition in the DTD (see step 1). Define the root element as follows:
   `<!ELEMENT` *RootName* `(`*OuterEntity*{*MultiplicityOperator*}`)>`

where:

- *RootName* is the name of the root element of the entity. The root element is not mapped to any UNIFACE object, so it is conventional to give the root element the same name as the DTD or to name it ROOT.
- *OuterEntity* is the name of the element mapped to the outermost UNIFACE entity in the XML stream.
- *MultiplicityOperator* is a single character that defines how many times an element can occur in the stream. The available options are:
  - ?—the element can occur zero or one times in the stream.
  - *—the element can occur zero, once or many times in the stream.
  - +—the element can occur once or many times in the stream.
  - *No operator*—the element must occur exactly once in the stream.

Define a DTD manually-root element declaration

At this stage, your DTD should look similar to the following:

```
<!ELEMENT ROOT (myentity*)>
```

The DTD defines the root element ROOT, which has one child element myentity.

3. Define elements for each of the UNIFACE entities in the XML stream (the *EntityDefs* in step 1). To do this, declare an element containing elements that represent the fields and inner entities of the entity. Define each entity as follows:

   <!ELEMENT *EntityName* ({$InnerEntity_1$ {*MultiplicityOperator*} | $FieldName_1$ }{, $InnerEntity_2$ {*MultiplicityOperator*} | $FieldName_2$ }...{, $InnerEntity_n$ {*MultiplicityOperator*} | $FieldName_n$ })>

   Where:

   - *EntityName*—is the name of the element mapped to the UNIFACE entity.
   - *InnerEntity*—is the name of an element mapped to an inner entity.
   - *FieldName*—is the name an element mapped to a field of the UNIFACE entity.

Define a DTD manually-element declarations

At this stage, your DTD should look similar to the following:

```
<!ELEMENT ROOT (myentity*)>
```

```
<!ELEMENT myentity (field1, field2, field3, innerentity)>
<!ELEMENT innerentity (fielda, fieldb, fieldc)>
```

The DTD defines elements for two entities, `myentity` and `innerentity`, and declares elements for their fields.

4. Define elements for all the fields of the entities in your DTDs. Define each field as follows:

   `<!ELEMENT` *FieldName* `(#PCDATA)>`

   Where *FieldName* is the name of an element representing a field. *FieldName* must be specified in the declaration of an entity element.

Define a DTD manually-field declarations

At this stage, your DTD should look similar to the following:

```
<!ELEMENT ROOT (myentity*)>
<!ELEMENT myentity (field1, field2, field3, innerentity)>
<!ELEMENT innerentity (fielda, fieldb, fieldc)>
<!ELEMENT field1 (#PCDATA)>
<!ELEMENT field2 (#PCDATA)>
<!ELEMENT field3 (#PCDATA)>
<!ELEMENT fielda (#PCDATA)>
<!ELEMENT fieldb (#PCDATA)>
<!ELEMENT fieldc (#PCDATA)>
```

The DTD defines elements for the fields of `myentity` and `innerentity`. Note that the only difference between field definitions is the element name.

5. Define attributes for the elements you declared in steps 3 and 4.

   You can use the following attribute declarations:

   - `<!ATTLIST` *EntityElement* `id CDATA #REQUIRED>`
   - `<!ATTLIST` *EntityElement* `crc CDATA #REQUIRED>`
   - `<!ATTLIST` *EntityElement* `status CDATA #REQUIRED>`
   - `<!ATTLIST` *EntityElement* `valerr CDATA #IMPLIED>`
   - `<!ATTLIST` *FieldElement* `valerr CDATA #IMPLIED>`
   - `<!ATTLIST` *ElementName AttributeName* "*AttributeValue*" `#FIXED>`

   Where:

   - *EntityElement* is the name of an element mapped to a UNIFACE entity

- *FieldElement* is the name of an element mapped to a UNIFACE field
- *ElementName* is either an *EntityElement* or *FieldElement*
- *AttributeName* is the name of the attribute
- *AttributeValue* is the value of the attribute

Define a DTD manually—attribute declarations

At this stage, your DTD should look similar to the following (all attributes have been declared):

```
<!ELEMENT ROOT (myentity*)>
<!ELEMENT myentity (field1, field2, field3, innerentity)>
<!ELEMENT innerentity (fielda, fieldb, fieldc)>
<!ELEMENT field1 (#PCDATA)>
<!ELEMENT field2 (#PCDATA)>
<!ELEMENT field3 (#PCDATA)>
<!ELEMENT fielda (#PCDATA)>
<!ELEMENT fieldb (#PCDATA)>
<!ELEMENT fieldc (#PCDATA)>
<!ATTLIST myentity       id     CDATA #REQUIRED>
<!ATTLIST innerentity    id     CDATA #REQUIRED>
<!ATTLIST myentity       crc    CDATA #REQUIRED>
<!ATTLIST innerentity    crc    CDATA #REQUIRED>
<!ATTLIST myentity       status CDATA #REQUIRED>
<!ATTLIST innerentity    status CDATA #REQUIRED>
<!ATTLIST myentity       valerr CDATA #IMPLIED>
<!ATTLIST innerentity    valerr CDATA #IMPLIED>
<!ATTLIST field1         valerr CDATA #IMPLIED>
<!ATTLIST field2         valerr CDATA #IMPLIED>
<!ATTLIST field3         valerr CDATA #IMPLIED>
<!ATTLIST fielda         valerr CDATA #IMPLIED>
<!ATTLIST fieldb         valerr CDATA #IMPLIED>
<!ATTLIST fieldc         valerr CDATA #IMPLIED>
```

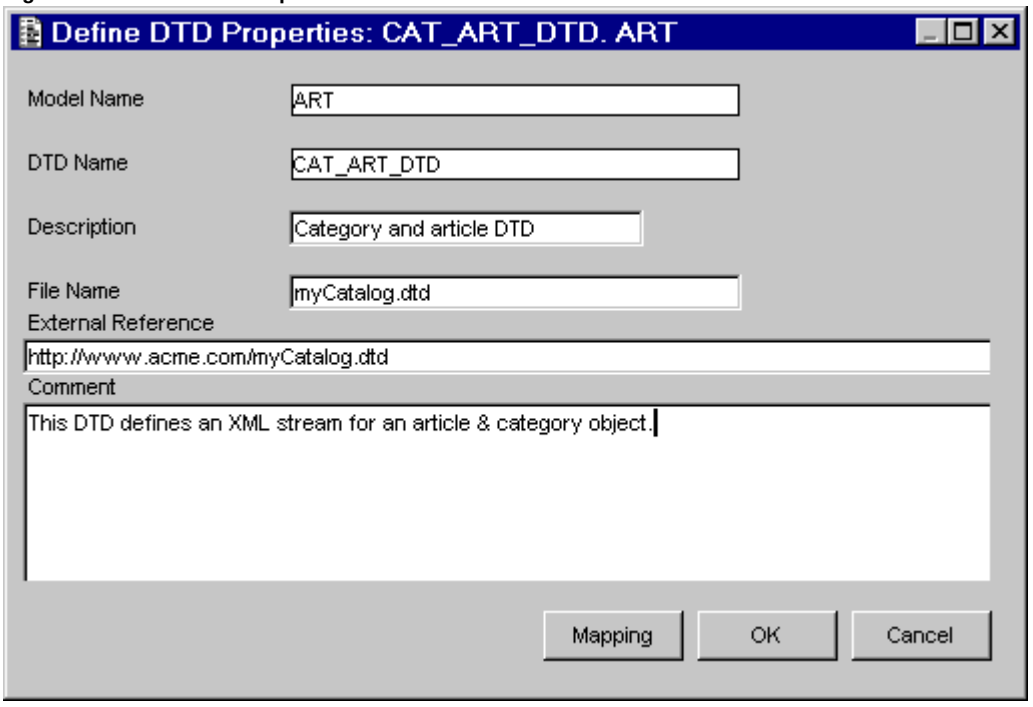
This DTD uses the processing information attributes required by UNIFACE for disconnected record sets. For more information, see section 1.2.4 *Attributes in XML streams* and section 1.1.6 *Disconnected record sets*. For an example of a DTD that uses user-defined attributes, see section 3.4.2 *Sample DTD and XML stream*.

6. Define a default mapping between elements defined in the DTD and UNIFACE fields and entities.
7. Validate the DTD by selecting File—>Validate.
8. Save the DTD by selecting File—>Save, or File—>Compile DTD.

5.1.9 Define additional DTD properties

To go to the DTD Properties form, click Properties on the Define DTD form. This opens the Define DTD Properties form.

Figure 5-8 Define DTD Properties form.



The following properties can be defined on the Define DTD Properties form:

- *Description*—is a short description of the DTD and its purpose.
- *File Name*—is the name of the DTD file created during compilation of the DTD.

- *External Reference*—is a URI reference to the DTD, such as **`http://www.acmedtds.com/mydtd`**. This URI is placed in the XML stream by the Proc statement `xmlsave/ref`. A 3GL component receiving an XML stream can use this reference to locate the DTD.
- *Comment*—is additional information about the DTD.
- *Default Mapping*—is the default mapping between elements defined in the DTD and fields and entities on a UNIFACE component.

External Reference and File Name are properties that support 3GL components, by providing alternative ways of supplying the DTD to a component. These properties are not required, and are ignored by UNIFACE components.

5.1.10 Compile a DTD

To compile a DTD, do the following:

1. Open the DTD in the DTD Editor, as described in section 5.1.1 *Start the DTD Editor*.
2. Select File—>Compile DTD.

Alternatively, use the command line switch `/dtd` to compile your DTDs. For more information, see /dtd in the UNIFACE Library.

5.2 Handling XML streams

The XML stream parameter type allows you to send disconnected record sets between components. The structure of an XML stream is defined by the rules specified in a DTD.

5.2.1 Load a DTD from a file

To load a DTD from a file, do the following:

1. Start the DTD Editor.
2. Click File—>Load from file...

3. Enter the name of the DTD (including the path to the file), or browse to the file.
4. Click Open to load the DTD file into the DTD Editor.

Caution: Clicking Open overwrites the current DTD definition.

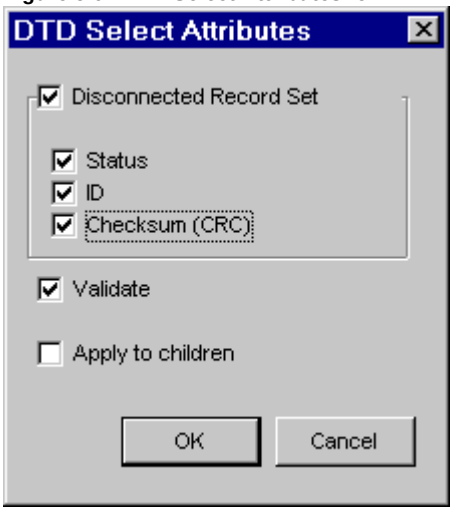
5. Select File—>Save or File—>Compile DTD to store the DTD in your Repository.

5.2.2 Select attributes for elements

To select attributes for elements, do the following:

1. Open the DTD in the DTD Editor.
2. Click Wizard to start the DTD Wizard.
3. For each element that requires attributes, do the following:
   - Select the element in the DTD Tree on the DTD Wizard form
   - Right-click on the element, and select Select Attributes from the pop-up menu
   - Select the attributes you require on the Select Attributes form.

Figure 5-9 DTD Select Attributes form.



- Click OK to accept the changes, or cancel the changes by clicking Cancel.

5.2.3 Define a local mapping for a DTD

To define a local mapping for an XML stream, do the following:

1. Define a local or component variable to contain the mapping information.
2. DTD mapping lists are UNIFACE associative lists. Each list item has the following syntax:

   "*ElementName*=*TargetName*;..."

   Where:

   - *ElementName*—is the name of an element in the DTD. *ElementName* is case-sensitive.
   - *TargetName*—is the name of a field or entity. The fields or entities do not have to be present on the target component. Values that cannot be mapped to a painted field are ignored. *TargetName* is not case-sensitive.

5.2.4 Create and send an XML stream

To create an XML stream, do the following:

1. Place the data destined for the XML stream into component fields. This can involve retrieving the data from a database, or creating and modifying field values by Proc.
2. Define a variable or parameter to contain the XML stream. For more information on the syntax for defining `xmlstream` variables and parameters, see variables and params in the UNIFACE Library.
3. Save the data into the `xmlstream` variable or parameter using the `xmlsave` statement. The `xmlsave` statement uses switches and arguments to manipulate the construction of the XML stream. For more information, see chapter 6 *Proc statements and functions*.

   The `xmlsave` statement creates the structure of the XML stream according to the rules specified in a DTD. For more information about defining a DTD, see section 5.1 *Using the DTD Editor*.
4. Send the XML stream as an `OUT` parameter.

5.2.5 Receive an XML stream

To receive an XML stream, do the following:

1. Define an operation with an XML stream as an `IN` parameter. For more information on the syntax for defining `xmlstream` parameters, see params in the UNIFACE Library.
2. Load the data from the XML streams into the component's data structure using the `xmlload` Proc statement. The `xmlload` statement uses switches and arguments to control the transfer of data from the XML stream to the component. For more information, see chapter 6 *Proc statements and functions*.
3. After the `xmlload` statement has executed, the data in the XML stream is available for processing.

5.2.6 Reconnect data from an XML stream

You can send data from a component as a disconnected record set. To do this, send the data as an XML stream using the `id`, `crc`, and `status` attributes. These attributes allow the data to be reconnected to component and database data. For more information, see section 1.1.6 *Disconnected record sets*, and section 1.2.4 *Attributes in XML streams*.

To reconnect data from an XML stream to component and database data, do the following:

1. Load the data from the XML stream using `xmlload`.
2. Use `retrieve/reconnect` to reconnect the data from the XML stream to existing data in the component and in the database (if the data has been loaded into database entities). For more information, see chapter 6 *Proc statements and functions*.

5.2.7 Do remote validation

You can use XML streams to pass data to another component for validation. This process is known as remote validation. The benefit of remote validation is that validation logic can be centralized and located in dedicated validation components.

To do remote validation, do the following:

1. Send the data as an XML stream to the validation component. The XML stream must include the `id,crc,status` and `valerr` attributes for all occurrences and fields in the stream.
2. Load the data from the XML stream into the validation component, and reconnect the data to component and database data, using `retrieve/reconnect`.
3. Validate the data. If necessary, modify the On Error triggers to set values for the `errormsg` property of `$occproperties` and `$fieldproperties`. For more information, see $fieldproperties and $occproperties in the UNIFACE Library.
4. Send the data back to the initial component as an XML stream using the `id`, `crc`, `status`, and `valerr` attributes. Values for `errormsg` are placed in the `valerr` attribute by the `xmlsave` statement.
5. Load the data from the XML stream into the initial component. Each `valerr` attribute encountered by `xmlload` fires an occurrence-level or field-level On Error trigger, with `$status` = 142.

5.2.8 Make DTDs available to non-UNIFACE components

UNIFACE components always read DTD data from the URR file. As 3GL components cannot access the URR file, UNIFACE provides alternate means to pass DTD information to non-UNIFACE components.

To make a DTD available to non-UNIFACE components, do one of the following:

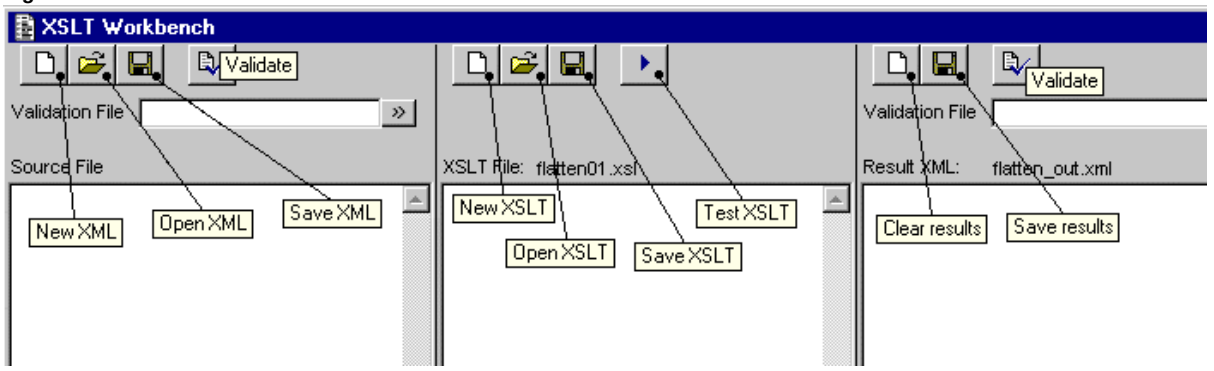
- Open the DTD in the DTD Editor, click Properties, and enter a file name for the DTD in the File Name field. This saves the DTD to disk as a text file, which can be accessed by 3GL.
- Open the DTD in the DTD Editor, click Properties, and enter a location for the DTD in the External Reference property field. The Proc statement `xmlsave/ref` can then be used to create an XML stream that includes a reference to this location. The External Reference must be in a format supported by the 3GL.
- `xmlsave/dtd` can be used to include the DTD in the XML stream, ensuring that it is available to 3GL components.

Note: Only the DTD declarations are made available to non-UNIFACE components. Additional DTD properties, such as the default mapping between UNIFACE entities and fields and the DTD, are not provided to non-UNIFACE components.

5.3 Develop and test XSLT stylesheets

The XSLT Workbench provides an environment for developing and testing XSLT stylesheets.

Figure 5-10 Buttons on the XSLT Workbench.



1. Select Utilities—>Edit XSLT to start the XSLT Workbench.
2. Click Open XSLT to open an XSLT file (with, by default, an **.xsl** file name extension), or create a new XSLT file by entering the XSLT instructions in the XSLT File pane of the Workbench.
3. To test the XSLT file, you can click Load XML to load an existing test XML file (with, by default, an **.xml** file name extension), or you can enter XML directly in the Source XML pane. You can validate the source XML stream by selecting a DTD file in the Validation File field, and clicking Validate.

   Click Test XSLT to test the XSLT file using the Source XML file as the input XML stream. The output XML stream is displayed in the Result XML pane. You can save the result stream to a file by clicking Save Results. You can validate the result XML stream by selecting a DTD file in the Validation File field, and clicking Validate.

Chapter 6 Proc statements and functions

This section describes the Proc statements and functions that you can use to handle XML stream parameters.

$occcrc

Name

$occcrc

Set or return the CRC checksum of an occurrence.

Synopsis

$occcrc(*EntityName*)

$occcrc(*EntityName*) = *CheckSum*

Where:

- *EntityName* is an entity name
- *CheckSum* is an eight-character hexadecimal string

Use

Allowed in all component types.

Description

$occcrc returns an eight-character hexadecimal string that represents the current values of an occurrence's fields. If the occurrence's field values change, the CRC checksum calculation yields a different result.

XML streams

CRC checksum values are required for disconnected record sets, and are produced automatically by UNIFACE when loading data from a database, or when creating or loading data from XML streams.

CRC checksums are stored in the crc processing information attribute in XML streams, if the crc attribute is specified in the DTD used by the XML stream. They are used by retrieve/reconnect to determine if an occurrence can be updated by an XML stream. retrieve/reconnect does not update an occurrence with data from an XML stream unless the CRC value in the stream matches the value of $occcrc for the occurrence.

Disabling CRC checks during retrieve/reconnect

If you set the value of $occcrc for an occurrence to "00000000", retrieve/reconnect does not carry out a CRC check before merging data from the XML stream into the occurrence.

$occcrc returns an eight-character string with hexadecimal display format. $occcrc can be set by $occcrc, and by xmlload (which sets $occcrc to equal the value of the crc attribute for the occurrence in the XML stream).

If the CRC value has not been set by $occcrc or by xmlload, $occcrc returns one of the following:

- The CRC checksum as calculated by the database driver—for database occurrences only
- An empty string—for non-database occurrences

If $occcrc is not equal to an eight-character hexadecimal string, $procerror is set to the error constant <UPROCERR_RANGE>.

$occproperties

Name

`$occproperties`

Return or set the properties of an occurrence.

Synopsis

`$occproperties(`*EntityName*`)`

`$occproperties(`*EntityName*`)` = *PropertyList*

Use

Allowed in all component types.

Description

`$occproperties` returns or sets the properties of an occurrence using an associative list.

PropertyList contains a UNIFACE associative list of *Key*=*Property* pairs. The following keys are used:

- `errormsg`—occurrence-level validation error messages.
  This can contain default UNIFACE validation error messages, or it can contain user-defined error messages.
- `subclass`—a style subclass used by a UNIFACE Server Page to present validation errors.

$occstatus

Name

$occstatus

Return or set the reconnect status of an occurrence.

Synopsis

$occstatus(*EntityName*)

$occstatus(*EntityName*) = *ReconnectStatus*

Where:

- *EntityName* is the name of an entity
- *ReconnectStatus* is the status of the occurrence for reconnection to existing data by the retrieve/reconnect statement.

Use

Allowed in all component types.

Description

Each occurrence in a component is either a new occurrence not yet stored in a database, an existing occurrence from a database, or is marked for deletion. The $occstatus function allows you to get or set this status for each occurrence in the component, if the value of $occstatus has previously been set for the occurrence by $occstatus or xmlload.

The value of $occstatus is used to set the value of the status attribute in XML streams.

$occstatus returns one of the following values:

- ""—$occstatus has not been set to a value for the occurrence. This can mean one of the following:
  - There is no status attribute in the XML stream; xmlload could not set a value for $occstatus when it created the occurrence.
  - The occurrence was not created by xmlload, and has not had $occstatus set in Proc.
  - $occstatus has been set to "" in Proc.
- "est"—the occurrence exists in the database
- "new"—the occurrence is new, that is, it does not exist in the database, or the occurrence originates from a non-database entity
- "del"—the occurrence is marked for deletion

retrieve/reconnect

Name

`retrieve/reconnect`

Reconnect data loaded from an XML stream with the occurrences in a database or component.

Synopsis

`retrieve/reconnect` {*EntityName*}

Where *EntityName* is the name of an entity painted on the component. *EntityName* can be a literal string, variable, or constant.

Use

Allowed in all component types.

Description

`retrieve/reconnect` resolves the occurrence state information stored in the processing information attributes of the XML stream, and fires all validate triggers. The procedure followed for each occurrence depends on the value of the `status` attribute for each occurrence in the XML stream.

status="new"

The XML stream declares that the occurrence does not exist in the database, or that the occurrence originates from a non-database entity, with `status="new"`. To reconnect the disconnected record, UNIFACE creates a new occurrence to contain the data in the disconnected record.

UNIFACE does the following for all disconnected records with `status="new"`:

1. A new occurrence is created.
2. The fields of the disconnected occurrence are merged with the occurrence created in step 1.
3. The disconnected record is deleted, and the `retrieve/reconnect` process continues with the next disconnected record.The process does not stop after a validation error or CRC mismatch.

After `retrieve/reconnect` has executed, it is possible that two occurrences with equal primary keys exist. This situation is handled by the default Proc in the validate triggers.

status="est"

The XML stream declares that the occurrence exists in the database with `status="est"`. To reconnect the disconnected record, UNIFACE must find the occurrence matching the disconnected record, and update it with the data in the disconnected record.

UNIFACE does the following for all disconnected records with `status="est"`:

1. UNIFACE searches for the occurrence in the component.
2. If the occurrence does not exist in the component, UNIFACE attempts to retrieve the occurrence from the database (if the component has a database connection).
3. If the occurrence is still not found, a new occurrence is created. The On Error trigger (`$error` = 2013) is fired, and the `retrieve/reconnect` process continues at step 5.
4. If the occurrence is located in step 1 or 2, UNIFACE compares the CRC value of the disconnected record with the CRC value for the occurrence, and does one of the following:
   - If the CRC values match, UNIFACE can update the occurrence with the values in the XML stream. The process continues at step 5.
   - If the CRC values do not match, the occurrence has been modified after the XML stream was created. The disconnected record cannot be reconnected, and the On Error trigger (`$error` = 2012) is fired for the occurrence. The process continues at step 7.
5. The fields of the disconnected record are merged with the occurrence found in steps 1, 2 or 3.
6. If cautious locking is used, the occurrence is locked.
7. The disconnected record is deleted, and the `retrieve/reconnect` process continues with the next disconnected record. The process does not stop after a validation error or CRC mismatch.

Note: Data is not merged for up entities with empty WRITE_UP and DELETE_UP triggers.

The reconnection can fail (for example, if an attempt is made to update a primary key field), and the On Error trigger will fire on the original occurrence.

status="del"

`status="del"` declares that the disconnected record should be deleted from the database and/or the component. To reconnect the disconnected record, UNIFACE must find the occurrence matching the disconnected record, and delete it.

UNIFACE does the following for all disconnected records with `status="del"`:

1. UNIFACE searches for the occurrence in the component.
2. If the occurrence does not exist in the component, UNIFACE attempts to retrieve the occurrence from the database.
3. If the occurrence is still not found, a new occurrence is created. The On Error trigger (`$error` = 2013) is fired, and the `retrieve/reconnect` process continues at step 5.
4. If the occurrence is located in steps 1 or 2, UNIFACE compares the CRC value of the disconnected record with the CRC value for the occurrence:
   - If the CRC values match, UNIFACE marks the occurrence as deleted. (The occurrence is only deleted when the component stores its data.) If cautious locking is used, the occurrence is locked.
   - If the CRC values do not match, the disconnected record cannot be reconnected, and the On Error trigger (`$error` = 2012) is fired for the occurrence. The occurrence is *not* marked for deletion.
5. The disconnected record is deleted, and the `retrieve/reconnect` process continues with the next disconnected record. The process does not stop after a validation error or CRC mismatch.

No status attribute

If no `status` attribute is defined in the XML stream, then all occurrences are treated as new (that is, all occurrences are regarded as having `status="new"`).

Fields not in the XML stream

An XML stream can include elements for a subset of the fields of an entity. During reconnection, field values are modified if the field is represented by an element in the XML stream (and if the record has `status="est"`). If the field is present in the XML stream as an empty element, the field is emptied during `retrieve/reconnect`.

If the field is not represented by an element in the XML stream, the field is not modified.

Caution: Included entities are an exception. The fields of included entities are emptied if the field is not represented by an element within the stream. All the fields of the included entity must be included in the XML stream, otherwise data is lost during reconnection.

retrieve/reconnect and nondisconnected occurrences

`retrieve/reconnect` processes disconnect occurrences (occurrences loaded into a component with the `xmlload` statement). Other occurrences in the component are not affected.

The values returned in `$status` following `retrieve/reconnect` are:

- 0, if the occurrence was successfully reconnected.
- >0, the number of errors `retrieve/reconnect` encountered. In this context, an error is defined as the number of times an On Error trigger returned a negative value.

Receiving an XML stream

The following code shows an operation that receives an XML stream, and loads the data from the XML into the component's data structure.

```
operation XMLIN
; This operation receives and
; reconnects an XML stream.

params
    xmlstream [DTD:ABCDTD.ABC]  MYSTREAM  : IN
endparams

clearxmlload MYSTREAM, "DTD:ABCDTD.ABC"
retrieve/reconnect
...
```

xmlload

Name

xmlload

Load data from an XML stream into occurrences painted on the component.

Synopsis

xmlload{/incldefmap}{/noprofile} *XMLvariable*, *DTDname*{, *DTDmapping*}

Where:

- /incldefmap—instructs xmlload to use the default DTD mapping defined in the DTD Editor.

  /noprofile—escape sequences for profile characters and subfield separators are not converted to the corresponding profile character of subfield separator during xmlload.
- *XMLvariable* is the field, variable, or parameter containing the XML stream.
- *DTDname* is the DTD used to validate the XML stream.

  DTDname is a literal string, variable, or constant using the following format:

  {DTD:}*Name*.*Model*

  Where:

  - DTD:—specifies that the XML stream is defined using a DTD (this is to ensure compatibility with future developments in the XML standard).
  - *Name* is the name of the DTD as specified in the application model.
  - *Model* is the name of the DTD's application model.
- *DTDmapping* is a UNIFACE list mapping elements to field names.

  For more information on default and component mapping for XML streams, see section 1.4.6 *Default DTD mapping and mapping defined on a component*.

Use

Allowed in all component types.

Description

xmlload transfers data from an XML stream into a component. The data is loaded directly into the component's data structure. xmlload does not interpret or initiate validation of data, so the data loaded by xmlload can include duplicates of occurrences already within the component, as well as occurrences marked for deletion.

Note: Occurrences marked for deletion are not accessible in Proc and are not displayed on forms. When data is stored to a database, occurrences marked for deletion in the component's data structure are deleted from the database. For more information, see store in the UNIFACE Library.

Note: To remove duplicates of occurrences and validate data from an XML stream, use `retrieve/reconnect`.

For more information on the conversion of data between UNIFACE and XML streams, see Section 1.4 *UNIFACE processing of XML streams*. For more information about how data, profile characters, and subfield separators are converted by the xmlload and xmlsave statements, see section 1.2 *XML streams*.

Mapping data between elements and fields and entities

The structure of the XML stream is defined by the DTD specified in *DTDName*.

UNIFACE maps field values to XML elements in the stream using a combination of the following mapping methods:

- Local mapping, contained in *DTDMapping* (highest priority). This is a mapping structure defined in Proc, as an associative list of element names and UNIFACE field and entity names. For more information, see section 1.4.7 *DTD mapping lists*.
- Default mapping (defined in the application model for each DTD). The default mapping is only used if the /incldefmap switch is used.
- Element and field/entity name matching (lowest priority). For example, UNIFACE maps data from fields to elements with the same name.

These mapping techniques work in parallel, with local mapping overriding all other mappings for a given element or field, and name matching only being applied to those elements for which no other mapping is defined.

The values returned in $status following xmlload are:

- -1, if xmlload could not load the XML stream (see $procerror for more information).

- 0, if xmlload loaded the XML stream successfully.
- >0, if xmlload loaded the XML stream, but could not find all the field and entity names specified in the default and local mappings. For each field specified in the mapping but not found in the component, $status is incremented by 1. More information is available in the message frame if the assignment setting $TESTMODE_COMPONENTS is set.

Table 6-1 Values commonly returned by $procerror for xmlload.

Value	Error constant	Meaning
-1500	<UXMLERR_DTD_NOTFOUND>	A DTD could not be located.
-1501	<UXMLERR_DTD_INVALID>	There is a syntax error in the DTD.
-1502	<UXMLERR_GENERATION>	An error occurred during generation of an XML stream.
-1503	<UXMLERR_PARSE>	An error occurred during parsing of an XML stream.

Creating and sending an XML stream

The following operation creates an XML stream from a component's data structure, and sends the XML stream as an OUT parameter.

```
operation XMLOUT
; This operation saves data to XML.

params
    xmlstream [DTD:ABCDTD.ABC]  MYSTREAM  : OUT
endparams

clear
retrieve
xmlsave MYSTREAM, "DTD:ABCDTD.ABC"
...
```

Receiving an XML stream

The following code shows an operation that receives an XML stream, and loads the data from the XML into the component's data structure.

```
operation XMLIN
; This operation receives and
```

```
; reconnects an XML stream.

params
    xmlstream [DTD:ABCDTD.ABC]  MYSTREAM  : IN
endparams

clearxmlload MYSTREAM, "DTD:ABCDTD.ABC"
retrieve/reconnect
...
```

xmlsave

Name

xmlsave

Place component data in an XML stream.

Synopsis

xmlsave{/mod}{/one}{/dtd | /ref}{/incldefmap}{/root}
XMLvariable, *DTDname*{, *DTDmapping*}

Where:

- /mod—includes only modified occurrences in the XML stream
- /one—includes only current outer occurrence (with all inner occurrences) in the XML stream
- /dtd—includes the DTD in the XML stream
- /ref—includes the URI location of the DTD in the XML stream
- /incldefmap—instructs xmlsave to use the default DTD mapping defined in the DTD Editor
- /root—excludes the XML version declaration from the saved output
- *XMLvariable* is the field, variable, or parameter for the XML stream
- *DTDname*—is the DTD used for the XML stream

  DTDname is a literal string, variable, or constant using the following format:

  {DTD:}*Name*.*Model*

  Where:

  - DTD:—specifies that the XML stream is defined using a DTD (this is to ensure compatibility with future developments in the XML standard).
  - *Name* is the name of the DTD as specified in the application model.
  - *Model* is the name of the DTD's application model.
- *DTDmapping* is a UNIFACE list mapping XML elements to UNIFACE fields and entities

Use

Allowed in all component types.

Description

`xmlsave` creates an XML stream from the data in a component. The stream is built from the complete hitlist, including occurrences currently marked for deletion. Occurrences and fields are selected from the data based on the mapping and switches used by the `xmlsave` statement.

For more information on data conversion between UNIFACE and XML streams, see section 1.4 *UNIFACE processing of XML streams*.

Mapping data between elements and fields and entities.

The structure of the XML stream is defined by the DTD specified in *DTDname*.

UNIFACE maps field values to XML elements in the stream using a combination of the following mapping methods:

- Local mapping, contained in *DTDmapping* (highest priority). This is a mapping structure defined in Proc, as an associative list of element names and UNIFACE field and entity names. For more information, see section 1.4.7 *DTD mapping lists*.
- Default mapping (defined in the application model for each DTD). The default mapping is only used if the `/incldefmap` switch is used.
- Element and field/entity name matching (lowest priority). For example, UNIFACE maps data from fields to elements with the same name.

These mapping techniques work in parallel, with local mapping overriding all other mappings for a given element or field, and name matching only being applied to those elements for which no other mapping is defined.

UNIFACE uses the following naming convention to generate element names in XML streams if no mapping information is specified by Proc statement `xmlsave`.

Table 6-2 Default generation of element names in XML streams.

Rule	UNIFACE item	XML stream item	Example element
Entity	Entity	*Entity.Model*	<UCDTYP.DICT>
Field[1]	Field	*Field*	<UDESCR>
Non-unique field[1]	Non-unique field	*Field.Entity.Model*	<UDESCR.UFORM.DICT>

Table notes:

1. UNIFACE applies the *non-unique field* naming rule when two or more entities in the same DTD have fields with identical names. In these cases, the first field added to the stream is generated using the *field* rule, subsequent fields using the same name in other entities are generated using the *non-unique field rule*.

The values returned in $status following xmlsave are:

- -1, if xmlsave could not create the XML stream (see $procerror for more information).
- 0, if xmlsave created the XML stream successfully.
- >0, if xmlsave created the XML stream, but could not find all the field and entity names specified in the default and local mappings. For each field specified in the mapping but not found in the component, $status is incremented by 1. More information is available in the message frame if the assignment setting $TESTMODE_COMPONENTS is set.

Table 6-3 Values commonly returned by $procerror for xmlsave.

Value	Error constant	Meaning
-1500	<UXMLERR_DTD_NOTFOUND>	A DTD could not be located.
-1501	<UXMLERR_DTD_INVALID>	There is a syntax error in the DTD.
-1502	<UXMLERR_GENERATION>	An error occurred during generation of an XML stream.
-1503	<UXMLERR_PARSE>	An error occurred during parsing of an XML stream.

Creating and sending an XML stream

The following operation creates an XML stream from a component's data structure, and sends the XML stream as an OUT parameter.

```
operation XMLOUT
; This operation saves data to XML.

params
    xmlstream [DTD:ABCDTD.ABC]  MYSTREAM  : OUT
endparams

clear
retrieve
xmlsave MYSTREAM, "DTD:ABCDTD.ABC"
...
```

Receiving an XML stream

The following code shows an operation that receives an XML stream, and loads the data from the XML into the component's data structure.

```
operation XMLIN
; This operation receives and
; reconnects an XML stream.

params
    xmlstream [DTD:ABCDTD.ABC]  MYSTREAM  : IN
endparams

clearxmlload MYSTREAM, "DTD:ABCDTD.ABC"
retrieve/reconnect
...
```

xmlvalidate

Name

xmlvalidate

Validate an XML stream.

Synopsis

xmlvalidate{/file} *XMLStream*{, *ValidationData*}

where:

- /file—instructs xmlvalidate to treat *ValidationData* as a system path to a DTD file.
- *XMLStream*—is the field, variable, or parameter containing the XML stream.
- *ValidationData*—is the field, variable, or parameter containing the validation rules to be applied to *XMLStream*. *ValidationData* can refer to a Repository object, to a file, or can contain the validation rules itself. xmlvalidate applies the following rules to distinguish these situations:
  - *ValidationData* is treated as a system path if the argument /file is used.
  - If *ValidationData* uses DTD syntax, xmlvalidate treats *ValidationData* as a DTD.
  - Otherwise, *ValidationData* is treated as the name of a DTD stored in the Repository.

Where ValidationData specified a DTD stored in the Repository, use the following format:

{DTD:}*Name.Model*

where:

- DTD:—specifies that the XML stream is defined using a DTD (this is to ensure compatibility with future developments in the XML standard).
- *Name*—is the name of the DTD as specified in the business object model.
- *Model*—is the name of the DTD's business object model.

Use

All component types.

Description

Validation objects (files, variables and so on) must follow the W3C Recommendations for DTDs. Furthermore, DTD objects should *not* contain the following items:

- an XML declaration (for example, `<?xml version="1.0"?>`)
- an enclosing DOCTYPE declaration (in fact, DTDs that include a DOCTYPE declaration do not conform to the XML 1.0 Recommendation.)

When *ValidationData* is omitted, UNIFACE reads the DTD declarations embedded in the *XMLStream*. If no declarations are embedded in *XMLStream*, then the parser reports a validation error.

If *XMLStream* contains embedded element and attribute declarations and you also specify *ValidationData*, the XML parser receives multiple declarations for items in the stream. The XML parser reports a validation error in this situation.

The following values are commonly returned in `$procerror`:

- 0—successful
- -1—`<UGENERR_ERROR>`. An error occurred.
- -1406—`<UPROCERR_MEMORY>`. Memory allocation failure.
- -1504—`<UXMLERR_VALIDATE>`. An error occurred during validation of an XML stream.

Note: Additional information is provided in `$procerrorcontext` , such as error messages from the XML parser.

Validate an XML stream

You can use the Proc statement `xmlvalidate` to validate an XML stream, even if the DTD for the XML stream is not in your Repository.

The following operation `XVALIDATE` validates an XML stream using `xmlvalidate`:

```
operation XVALIDATE
params
   numeric I_STATUS          : OUT
   string  I_STATUSCONTEXT   : OUT
   string  I_DTD             : IN
   string  I_XML             : IN
endparams

; I_DTD is a file path; use the argument /file
```

```
; to indicate this to xmlvalidate.
xmlvalidate/file I_XML, I_DTD
I_STATUS = $procerror
I_STATUSCONTEXT = $procerrorcontext
end ; operation XVALIDATE
```