software AG

Tamino

Advanced Concepts

Version 8.2.2

October 2013

tamino XML Server

This document applies to Tamino Version 8.2.2.

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

Copyright © 1999-2013 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, United States of America, and/or their licensors.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at http://documentation.softwareag.com/legal/.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices and license terms, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". This document is part of the product documentation, located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

Document ID: INS-ADVCONC-822-20130226

Table of Contents

Preface v
I From Conceptual Model to Schema 1
1 Informal Description 3
2 Conceptual Modeling 5
Introducing Asset-Oriented Modeling 6
Asset or Property? 7
Normalization 13
Determining Business Objects 15
Resolving is_a Relations 17
Reverse Engineering of Relational Schemas 20
Models and Namespaces 23
3 Introduction to XML Schema 29
Datatypes 30
Namespaces and Wildcards 43
The Structure of a Schema Definition 45
Reuse Mechanisms 46
Elements vs. Attributes 48
4 From Model to Schema 51
Adding Type Information 52
Document-Centric Layout 53
Creating a Type Library 54
Implementing Business Objects 55
Segmentation and Optimization 57
Multi-Namespace Schema Composition 58
Schema Evolution 60
Open Content Model 61
Versioning 62
5 Integrity 63
Simple Constraints 64
Cross Field Constraints 65
Constraints Across Documents 65
Data Integrity 67
Unique Keys 68
6 Operations 69
7 From UML to XML 71
XML Support in UML 72
From Conceptual Model to UML 73
8 Schema-Related Web Sites 77
II From Schema to Tamino 79
9 Tamino Annotations in XML Schema 81
Annotation and Appinfo 82
Schema-Level Definitions 83
Node-Level Definitions 84

10 Namespace Support 87
Qualified Queries 88
11 Indexing 89
Declaring an Index 90
Candidates for Indexes 92
Composite Keys 93
Object Identity 93
Text Retrieval 95
12 Document Composition 99
Dynamic Joins with Tamino XQuery 4 100
13 Efficient Querying 103
Data Modeling for Efficiency 104
Efficient Indexing 105
Efficient Queries 106
14 Performance Issues 111
III Utilizing Server Extensions 113
15 What are they Good For? 115
16 Queries 117
17 Derived Elements 123
18 Maintaining Semantic Integrity 129
19 Building Up a Library 133
20 More Examples 135
concat 136
contains 138
substringBefore 139
substringAfter 140
substring 141
trim 142
normalizeSpace 144
stringLength 146
qdoc 147
IV Rapid Application Development with Tamino 149
21 Introduction to XSLT 151
Procedural Transformation 152
Rule-Based Transformation 158
Limitations of XSLT 162
Using Style Sheets with Tamino 162
22 Mapping a Schema to a Web Page 165
23 Navigation with XLink 175
Defining Navigational Objects 176
Defining Navigational Links 178
24 The Tamino JavaScript API 191
25 XSLT Summary 197
26 Rapid Prototyping with XQuery 4 199
Index 201

Preface

Developing complex XML-based information systems is a relatively young discipline. Most of today's information systems are based on relational techniques, simply because relational database management systems have been in use for a long time.

However, this is going to change: the closed Enterprise Information Model is giving way to a service-oriented Open Network Information Model. With XML, it has now become possible to implement highly complex content in an open, natural and straightforward way - content that is quickly becoming too complex for relational technology. XML has become the standard technology for a huge variety of applications, including all web service implementations and SOA (service oriented architecture) systems.

In the following chapters, we discuss how an enterprise-class Internet-enabled XML server such as Tamino can store and process such complex information. We assume that you are familiar with the basics of XML and that you know how to store and retrieve single documents with Tamino.

This document is intended for systems analysts, database administrators, schema designers, application developers and web designers.

The chapter **From Conceptual Model to Schema** discusses how to model complex contents into a collection of interrelated document types. We introduce XML Schema and show how to translate a conceptual model into schemas.

The chapter **From Schema to Tamino** explains how these schemas can be implemented with Tamino. We discuss strategies for namespaces, indexing, queries, transactions, and performance.

The chapter **Utilizing Server Extensions** shows how Tamino's Server Extensions can be used to extend the built-in functionality of Tamino. Code for example programs (date comparison, derived elements, fetching documents, triggers, and more) is provided.

The chapter **Rapid Application Development** with Tamino introduces programming with XSLT. We show how XSLT can be used to implement a presentation layer that derives HTML web pages from presentation neutral XML content, and compare it with the emerging query language XQuery. We show, too, how navigation structures can be described in a separate navigation layer with the help of XLink.

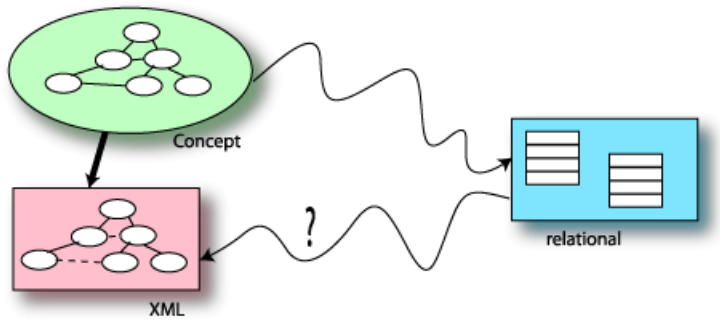
I From Conceptual Model to Schema

In this chapter, we first give a short introduction to conceptual modeling and XML Schema. Then we discuss how conceptual models can be transformed into XML schemas. A discussion of how to use the UML in XML environments ends the chapter.

Motivation

Conceptual modeling techniques are well known in enterprise software construction. Before an application is implemented, a requirements analysis is performed which results in an informal description of the business domain. The conceptual model captures the results of the requirements analysis with more formal (and mostly visual) means. The conceptual model thus describes the business domain. Constructing a conceptual model ensures that the participating analysts and engineers have understood the problem. The model can then be used as a basis for the technical architecture. The conceptual model itself should be independent from a given technical infrastructure; it should not matter whether SQL or XML is used to store the data, nor which programming language is used to implement the system. The main purpose of conceptual modeling is to improve communication between the parties involved in the development process.

Conceptual modeling is a well-established discipline in the relational world. Wouldn't it be sufficient to adopt the techniques and strategies developed there and simply map the relational designs onto XML structures? But while this approach may be necessary to reengineer existing applications (see **Reverse Engineering of Relational Schemas**), under normal circumstances it leads to poor XML representation of the conceptual model: too much structural information is lost in the process.



In the rest of this chapter, we therefore discuss how conceptual models can be mapped directly onto XML. We also introduce a modeling method that is particularly suited for the new application domains that are typical for XML. Not that XML by itself would require a new modeling method. But the good old Entity Relationship Diagram (ERD) is not really well suited for the new, open environments in which XML dominates.

Our Example: 'All that Jazz', A Knowledge Base for Jazz Music

Our example is a knowledge base for jazz music and jazz musicians. As you probably know, the relationships between jazz musicians are manifold and complex. New bands and projects are set up all the time, and there are many ways of collaboration. In this aspect jazz music very much resembles electronic business, where business relations are much more short-lived than in the old economy. In some cases we have fully virtual productions, for example in Carla Bley's famous album "Escalator over the Hill".

In the following section we detail this example. We start with an informal verbal description, formalize this description into a conceptual model, and then discuss how this model can be transformed into a document base and into XML Schema. Finally, we take a look at how UML can be employed in this process.

This information is organized under the following headings:

Informal Description
Conceptual Modeling
Introduction to XML Schema
From Model to Schema
Integrity
Operations
From UML to XML
Schema-Related Web Sites

1 Informal Description

A popular method to model an information domain is to start with an informal, verbal description of the scenario:

```
A jazz musician is a person.

A person has a name and a birth date.

A jazz musician collaborates with other jazz musicians.

A jazz musician belongs to a style
(during a certain period of time).

Instrumentalists, jazz singers, jazz composers are jazz musicians.

An instrumentalist plays one or several instruments.

An instrument has parts, consisting of other parts.

An instrument has a color and a maker.

A saxophone is an instrument.

A saxophone has a mouthpiece.
```

```
A saxophone mouthpiece has a body and a reed.

A mouthpiece body has a maker.

Mouthpiece, mouthpiece body and reed are instrument parts.

A reed has a maker and a grade.

A jam session is a form of collaboration.

A jam session is performed at a location and at a particular time.

A project is a form of collaboration
(during a certain period of time).

A band is a form of a collaboration
(during a certain period of time).

A collaboration can result in one or several albums.

An album has an album number and a title.

An album has one or several tracks.

Albums or jazz musicians are reviewed by critics.

A review has a publication date, a URL and some text.

A critic is a person.
```

The actual relationships are, as a matter of fact, much more complicated. For example we could include a full taxonomy for musical instruments and styles. But for the purpose of this introduction to conceptual modeling, the above description is adequate.

A simple grammatical analysis can identify *nouns* (Jazz musician, Person, Name, Birth Date, Band, Collaboration, Location, Album, etc.) and *verbs* (is, has, collaborate, plays, etc.) in each sentence. This step helps us to identify relevant information items of the conceptual model.

Tip: Always use simple sentences.

2 Conceptual Modeling

- Introducing Asset-Oriented Modeling 6
- Asset or Property? 7
- Normalization 13
- Determining Business Objects 15
- Resolving is_a Relations 17
- Reverse Engineering of Relational Schemas 20
- Models and Namespaces 23

We now transform this informal description into a more formal conceptual model.

In traditional conceptual modeling (such as Entity Relationship Diagrams or Object Role Modeling), nouns would end up as *entities* (or attributes) and verbs would end up as *relationships*. However, in the context of modern information systems this approach can create more problems than it solves. For example: what should we do with `Collaboration-collaborate` or with `reviews-Review`? Should we use the verb or the noun? Which represents this concept better: an entity or a relationship?

Introducing Asset-Oriented Modeling

This ambiguity is one reason for taking a different approach and removing the artificial separation between entity and relationship: in our model, both nouns and verbs become *Assets*. What is new here is that the classic relationships are treated on the same level as entities: as things. This syntactical reification (reify = to make into a thing) leads to a considerable simplification of the conceptual model, and has some other advantages, too. In particular, it results in models that are easy to transform into XML.

When modeling nouns and verbs as assets, there are two notable exceptions:

- The verb "has" indicates either that an asset is attributed with a property, as in:

  ```
  A person has a name and a birth date.
  ```

  or that an asset aggregates other assets, as in:

  ```
  A saxophone has a mouthpiece.
  ```

- The expression "is a" indicates that an asset acquires properties from another asset, as in:

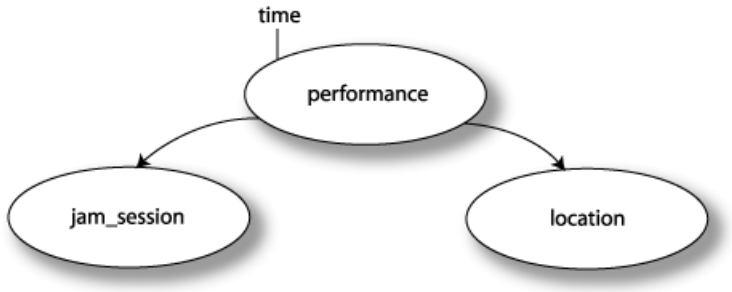
  ```
  A jazz musician is a person.
  ```

The noun on the left hand side (jazz musician) is usually a more specific term; the noun on the right hand side is usually a more general term (person).

Let us see how we can model a rather complex sentence like:

```
A jam session is performed at a location and at a particular time.
```

A possible resolution is to model `jam_session`, `performance`, and `location` as assets. The asset `performance` has a qualifying property: `time`. Usually we use the noun form (`performance`) of a verb (`performed`) to name the corresponding asset.



Asset or Property?

In the most general sense, an asset is anything we can talk about. But for the purpose of modeling we want to categorize the `things we can talk about` into assets and properties.

- In our informal description, *properties* are usually indicated by the verb "has": `A saxophone has a color`.
- Anything which plays a certain role in the context of our business is definitely an *asset*.

Tip: In many cases the distinction between a property and an asset can be made using a simple rule: A property can belong to an asset, but an asset cannot belong to a property. For example: a color cannot have a saxophone.

However, the distinction between both is not always so easy, and depends in some cases on the context. Take for example:

```
An instrument has a maker.
```

```
A saxophone has a mouthpiece.
```

In the context of our small knowledge base, `maker` and `mouthpiece` do not play any particular role. So it is acceptable to model them as properties of `instrument` or `saxophone`. However, in the context of a supply chain for musical instruments, `maker` and `mouthpiece` would certainly play a role (as manufacturer and product), so we would have to model them as assets.

An item that is only connected to a single asset (like `maker`) is always a candidate for becoming a property. In contrast, an item that also has other relationships must be modeled as an asset. Take for example:

```
A project can result in an album.
```

Here, `album` could be modeled as a property of `project`, if we had not specified:

```
An album is reviewed by critics.
```

Composite Properties

In contrast to classic Entity Relationship Diagrams, we allow *composite* properties. Above, we modeled `performance` and `location` (for example, Cotton Club, Savoy Ballroom, or Centralstation) as separate assets. But if these locations play no particular role in our business case, we can just use a composite property to represent these items.

For

```
A jam session is performed at a location and at a particular time.
```

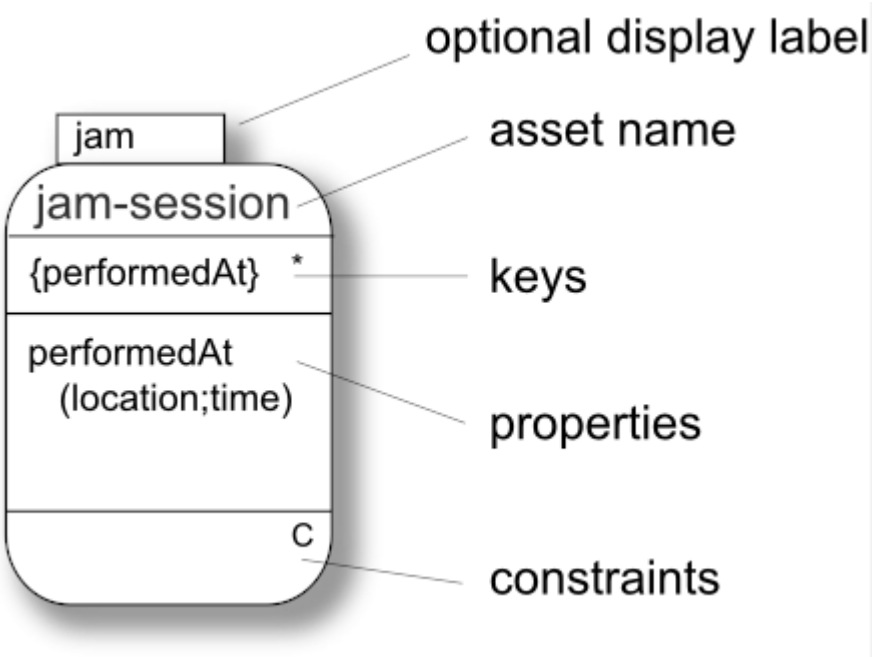
we might define a property `performed_at` that includes the sub-properties `location` and `time`.

Similarly, the property `name` of asset `person` may include the sub-properties `first`, `middle`, and `last`. A `mouthpiece` may have sub-properties `body` and `reed`. This technique of composite (or nested) properties allows us to arrive at very compact models and – not surprisingly – results in very appropriate XML representations.

A Notation for Properties

We are now ready to introduce a more formal notation for assets.

The figure below shows the graphic representation of an asset. The first line contains the asset name. This is followed by a key definition (we discuss this later), a list of properties, and a list of constraints (also discussed later). The optional display label at the top of the asset is used when the names of asset instances should differ from the asset name. In this case the display label shows the possible names of asset instances. When an asset does not have instances (i.e. when the asset is abstract) the display label is grayed out.



We introduce the following notation for properties:

Syntax	Description	Example
prop	An atomic component without further structure.	`birthDate`
(...)	A property particle, i.e. a structure consisting of several sub-properties. The parentheses contain nested expressions consisting of the following structures:	See following rows.
(sub,...,sub)	Sequence (ordered list).	`name(first, middle?,last)` Here, we require that the sub-properties of `name` are always specified in the defined order. Queries can later rely on this order.
(sub&...&sub)	Bag (unordered list).	`reed(maker&grade)` ↩ Here, we do not prescribe a particular sequence in which `maker` and `grade` must be specified. Queries cannot rely on an order relation between both.

Syntax	Description	Example
(sub|...|sub)	Choice (alternative).	`(period(from,to) | ↩ performedAt(location&time))` A property is either a `period` or it is a `performedAt` property.

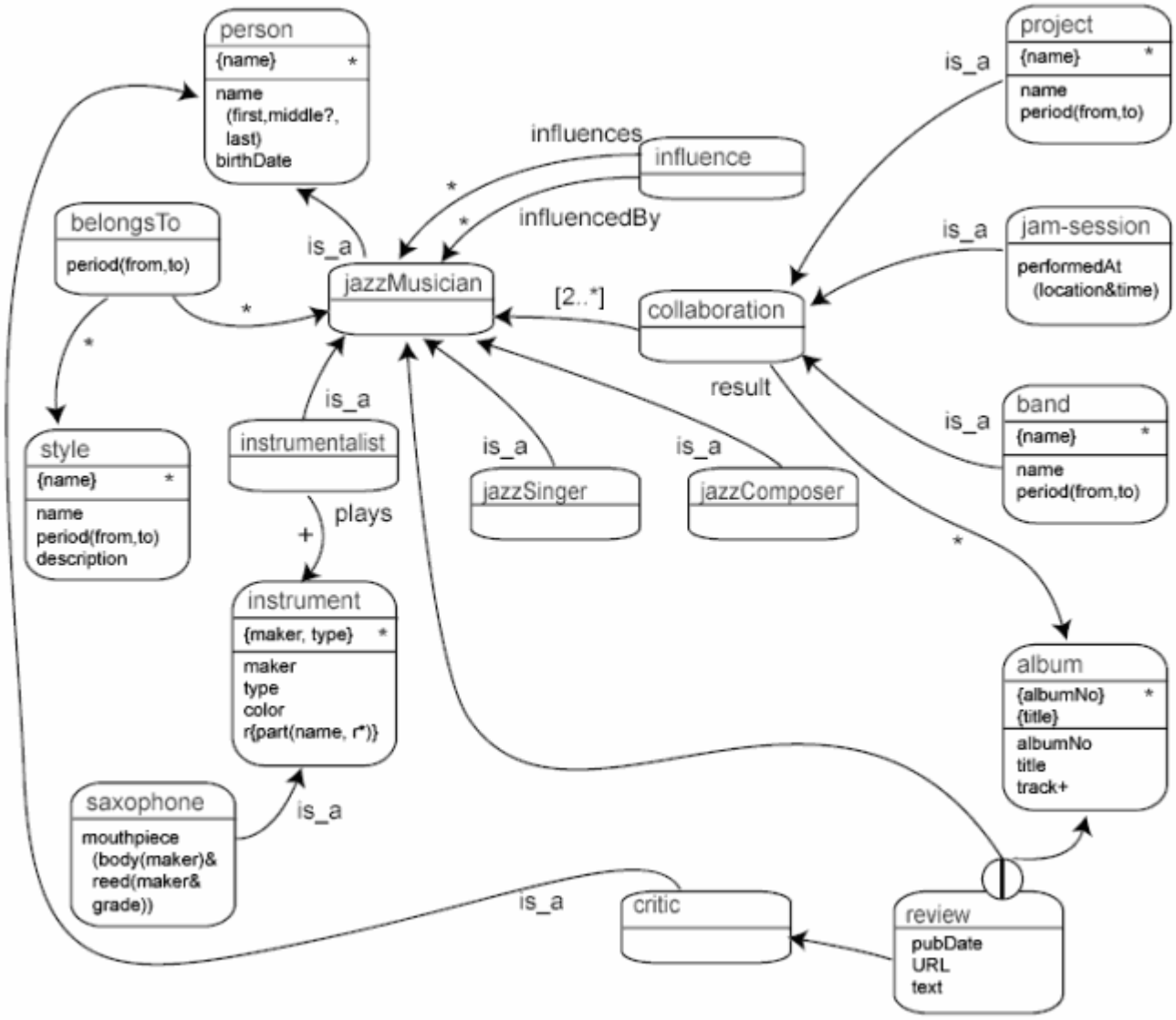
Both properties and particles can be suffixed with one of the following modifiers:

Syntax	Description	Example
(no modifier)	mandatory [1..1]	`last ↩` A last name is always required.
prop?	optional [0..1]	`middle?` Not everybody has a middle name, so we make this property optional.
prop+	repeated [1..n]	`track+` An album has one or several tracks.
prop*	optional and repeated [0..n]	`album*` An arbitrary number of albums.
prop[n..m]	a minimum of n occurrences and a maximum of m occurrences with 0 <= n <= m	`track[1..25] ↩` The number of tracks is restricted to 25 at most.

The following notation allows recursively structured properties to be defined:

Syntax	Description	Example
label{...label...}	A label defines a reference point to the expression within the curly braces. Later occurrences of the label are substituted with this expression. In particular, labels allow recursive structures to be defined.	`r{part(partNo,r*)} ↩` specifies a tree-like structure of parts. Note that the *-modifier ensures that the recursive structure is finite.

Given this notation, we now define our complete model. We have added a few more properties.



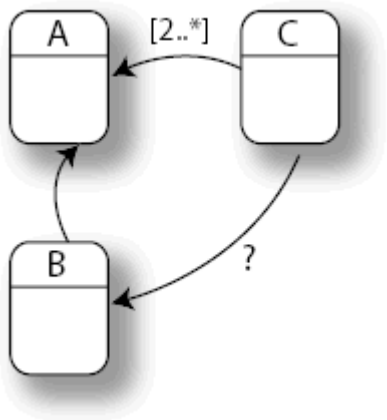
A Notation for Arcs

Assets are connected via directed arcs. You should not misinterpret these arcs as classic relationships in the sense of Entity Relationship Diagrams. (Remember that relationships are assets too.) For this reason, arcs do not have names; however, the origin of an arc may be decorated with a role name.

Similar to the notation used for properties, we use XML syntax to denote the cardinality of each arc:

+	1..n
*	0..n
?	0..1
[n..m]	n..m (0<= n <= m)

Caution: Avoid situations in which the constraints of the model can never be satisfied, for example:



Here, each instance of asset type C requires the existence of at least two instances of asset type A and at most one instance of asset type B. This is in contradiction with the implicit constraint between B and A which dictates a 1:1 relation between both.

Important: Therefore, we should always make sure that the intersection of all constraint cardinalities used within a cyclic structure is not empty. This is always the case when we only use the first three constraint types: their intersection contains always 1..1.

In addition to constraints, we may attribute the origin of each arc with a *role name*. Take for example the asset `influence`. This asset has two arcs that connect it with `jazzMusician` – one in the role of the influenced musician and one in the role of the musician who has influenced the first.

Inheritance

You will also have noticed that we have represented the `is_a` relationships not as assets but merely as arcs, the origin of the arc being decorated with the role name `is_a`. This makes it easier to identify such inheritance and classification relationships. In contrast to normal arcs, inheritance arcs may only be attributed with the “?” modifier indicating optional inheritance.

Special Cases

The `has` relationship (if it does not result in a property) results in a simple arc, too, pointing from the asset that “has” to the asset which is “had”. This is, for example, the case for the relationship between `jazzMusician` and `collaboration`. By using the noun form (collaboration) of "collaborates", the sentence

```
A jazz musician collaborates with other jazz musicians.
```

is interpreted as

```
A collaboration has jazz musicians.
```

a relationship which is modeled with a simple arc.

A further design decision is to downgrade "plays" and "results_in" in

```
An instrumentalist plays one or several
instruments.
```

and

```
A project can result in an album.
```

We replace these relationships with the simple "has" relationship, too (i.e. "plays" and "can result" do not become assets). In order to retain the semantic information, we use "plays" and "result" as role names. This is not always easy to decide. If, for example, we had the relationship

```
An instrumentalist owns one or several instruments.
```

then we would probably model "owns" as a separate asset `ownership` because this asset could become the subject of a new business relation:

```
After four weeks ownership is transferred to the pawn broker.
```

Clusters

The asset `review` shows another interesting construct: a cluster. A cluster is used to denote alternatives, and is represented by a circle containing the choice operator. In our case the cluster says that a review relates either to an album or to a jazz musician. *A cluster is a union of disjoint asset types.* Clusters are possible for normal arcs and inheritance arcs.

Normalization

After we have obtained a first draft of our model, we should normalize it. Unlike relational technology, XML allows a physical data format that very closely follows the structures of the actual business data – there is no need to break complex information items into a multitude of “flat” tables. We shall find that an XML document can represent a conceptual entity almost unmodified.

This does not mean that no normalization is required. We still must make sure that our information model does not have redundancies, and that we end up with an implementation that is easy to

maintain and consistently matches the “real world” relationships between information items. We make sure that:

- Asset types are *primitive*, i.e. their properties do not contain assets that could be modeled as independent asset types. For example, the asset type `album` must not embed data from `jazzMusician`.
- Asset types are *minimal*, i.e. they do not contain redundant properties, meaning none of their properties can be derived from other properties. For example, the asset type `person` must not contain a property `age` as this can be derived from `birthDate`.
- Asset types must be *complete*, i.e. other assets contained in the real world scenario can be derived from the defined asset types. Our example is *not* complete, as we made no provision for solo albums. Our model contains only albums that are the result of a collaboration. Also, albums usually contain information about which musician played which instrument. This is not covered by our model.
- Asset types must *not* be *redundant*, i.e. it must not be possible to derive any of the defined asset types from other asset types. In our example, we have a redundant asset. A band is a kind of project - the main difference is that it exists over a longer period of time and probably produces more albums. We could reflect this situation in our model by removing the properties `name` and `period(from,to)` from asset `band`, and by routing the `is_a` arc from `band` to `project` instead of `collaboration`.
- All asset types must have a *unique meaning*.
- Assets should have a *key*. Keys must be minimal, i.e. they must consist of the smallest set of properties that can uniquely identify an instance. In our example, not every asset has a key (for example, `belongsTo`, `influence`, `collaboration` and `review` don't have a key). We should introduce suitable keys for these assets. `jazzMusician`, `instrumentalist`, `jazzSinger`, `jazzComposer`, and `critic` do not need their own key, because they inherit one from `person`. If an asset type does not have suitable properties that can act as keys, we can easily equip them with some kind of a unique property (for example by generating a UUID for each instance).

Partitioned Normal Form

While the steps discussed above already result in a pretty robust model, there is one more thing we can do. Assets finally result in XML elements or documents, and can thus be subject to transformations (for example, via an XSLT stylesheet). To make the keys robust against such transformations, we should make sure that each asset is in *Partitioned Normal Form* (PNF).

An asset type or property is in Partitioned Normal Form (PNF) if the atomic properties of an asset constitute a key of the asset and all non-atomic properties and sub-properties are in Partitioned Normal Form themselves.

Or, in other words: All complex structures in the model (assets and complex properties) must have atomic child nodes that can act as a key.

In our example, the following asset types are not in PNF:

- `person`, because the key `name(first,middle?,last)` is a composite. A solution would be to introduce a personal ID. Here, we opt to introduce an atomic ID composed from last name, middle name and first name, such as `MingusCharles`.
- `jamSession`, because the key `performedAt(time, location)` is a composite. Here we opt for a different solution. We resolve the property `performedAt` into two independent properties: `time` and `location`. These two properties are atomic and can thus constitute a multi-field primary key that conforms to PNF.
- `saxophone`, because the composite property `mouthpiece` is not in PNF (it has no atomic property which could act as key). Here we should rather remove the property mouthpiece and use `mouthpiece_body` and `reed` directly as parts of `saxophone` (because `mouthpiece` is in fact not a single physical entity).

In particular, if we plan to store assets in relational databases (for example through Tamino's X-Node), PNF is essential. Relational technology requires fragmenting complex structures into flat relational tables. Keys that span complex structures would be lost during such a transformation to First Normal Form (1NF).

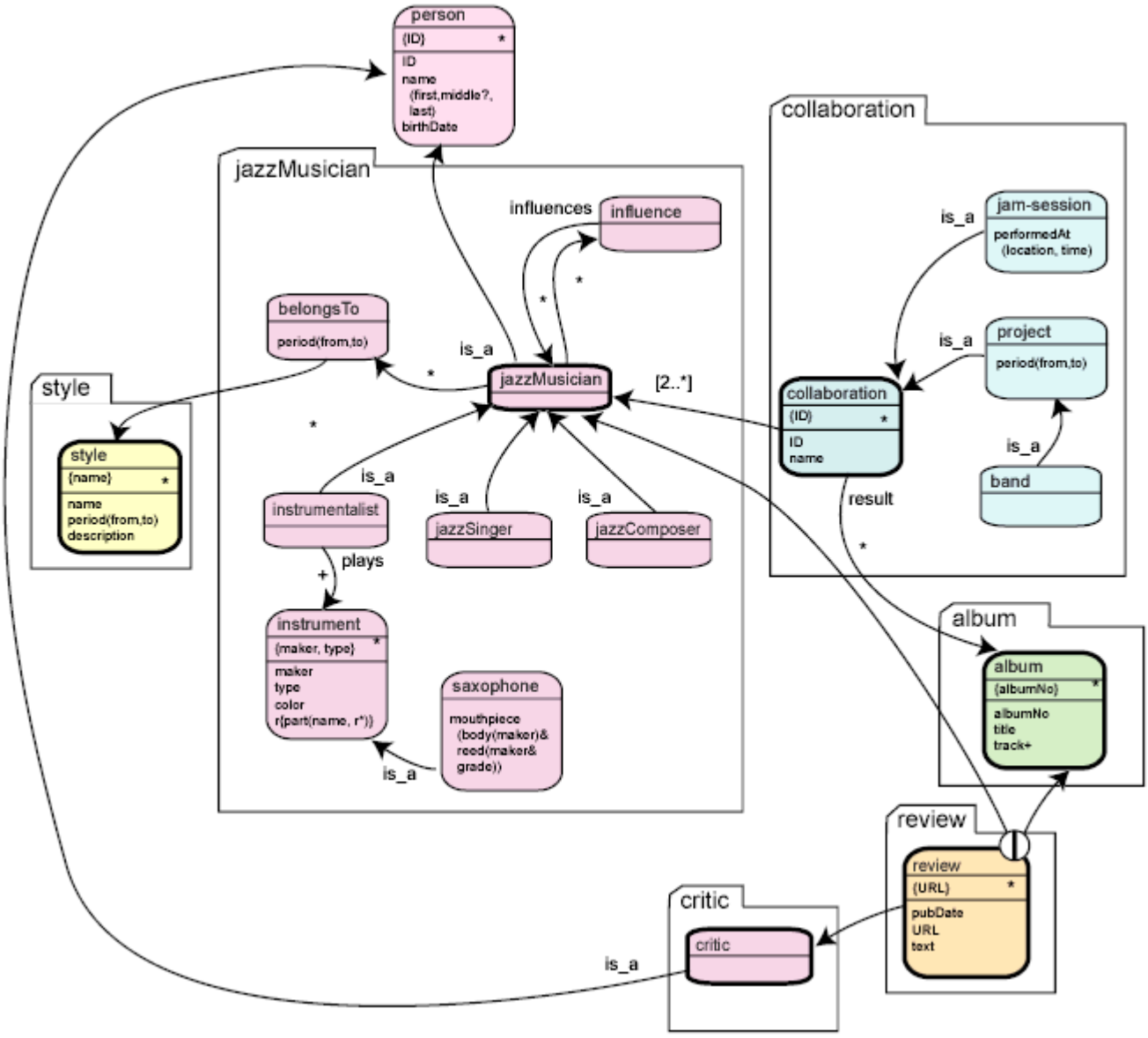
Determining Business Objects

Business objects are assets that play a prominent role in our scenario. In order to be able to identify a business object, we must not only have an idea about the structure of the information, but also what it will be used for.

In our example, all `jazzMusician` asset types, `style`, all `collaboration` asset types, `album`, `review`, and `critic` could be business object classes. Jazz musicians are clearly the most important topic in our knowledge base, but similarly important are `style` and the various collaborations. `album` could play a separate role when we connect our knowledge base with a mail order system. And `review` is probably an external resource to which we have to link via URL.

On the other hand, we made the decision not to model `instrument` as a separate business object. We are only interested here in the instrument that a given musician plays; we do not plan to set up a knowledge base about musical instruments as such. Consequently, we incorporate `instrument` and its subtypes into the `jazzMusician` business object.

We then group the remaining assets around the assets designated as business objects. Here, we have shown this by demarcating each business object with a labeled box. We use a bold outline for the identifying asset of each business object.



However, there is one constraint that we must enforce when constructing business objects from assets:

> ⚠ **Important:** Starting from the identifying asset of a business object, we must be able to reach any asset belonging to that business object by following the arcs in the indicated direction.

This constraint allows us to interpret each business object as an aggregation, and later allows us to easily implement the business objects in hierarchical XML documents.

When we check this constraint for our model, we encounter two problems: From the assets `belongsTo` and `influence`, both arrows lead to asset `jazzMusician`. This is bad, because when starting at `jazzMusician` we cannot reach `belongsTo` and `influence`.

In order to solve these problems, we simply reverse one arc for each of the assets `belongsTo` and `influence`. This results in a slightly different interpretation; we are now saying:

```
A jazzMusician has a "belonging" to a style.
```

and

```
A jazzMusician has influence.
```

By doing so, we have completed the former syntactic reification of relationships with a semantic reification – "belonging" and "influence" have become true assets of `jazzMusician`.

Caution: When we reverse an arc, any cardinality constraint of that arc becomes invalid. Therefore, we always decorate reversed arcs with an asterisk (*) to indicate that there are no cardinality constraints for that arc. By doing so, however, we may lose some structural information.

In addition, we have taken the opportunity to fix some problems with keys. `collaboration` and `review` definitely need keys, because they are identifying assets of business objects. The identifying asset of a business object *must* always have a key, because otherwise instances of business object classes could become inaccessible. We have equipped `collaboration` with a new property, namely `ID`, which we use as a key. The reason is that the property `name` may not be unique.

Resolving is_a Relations

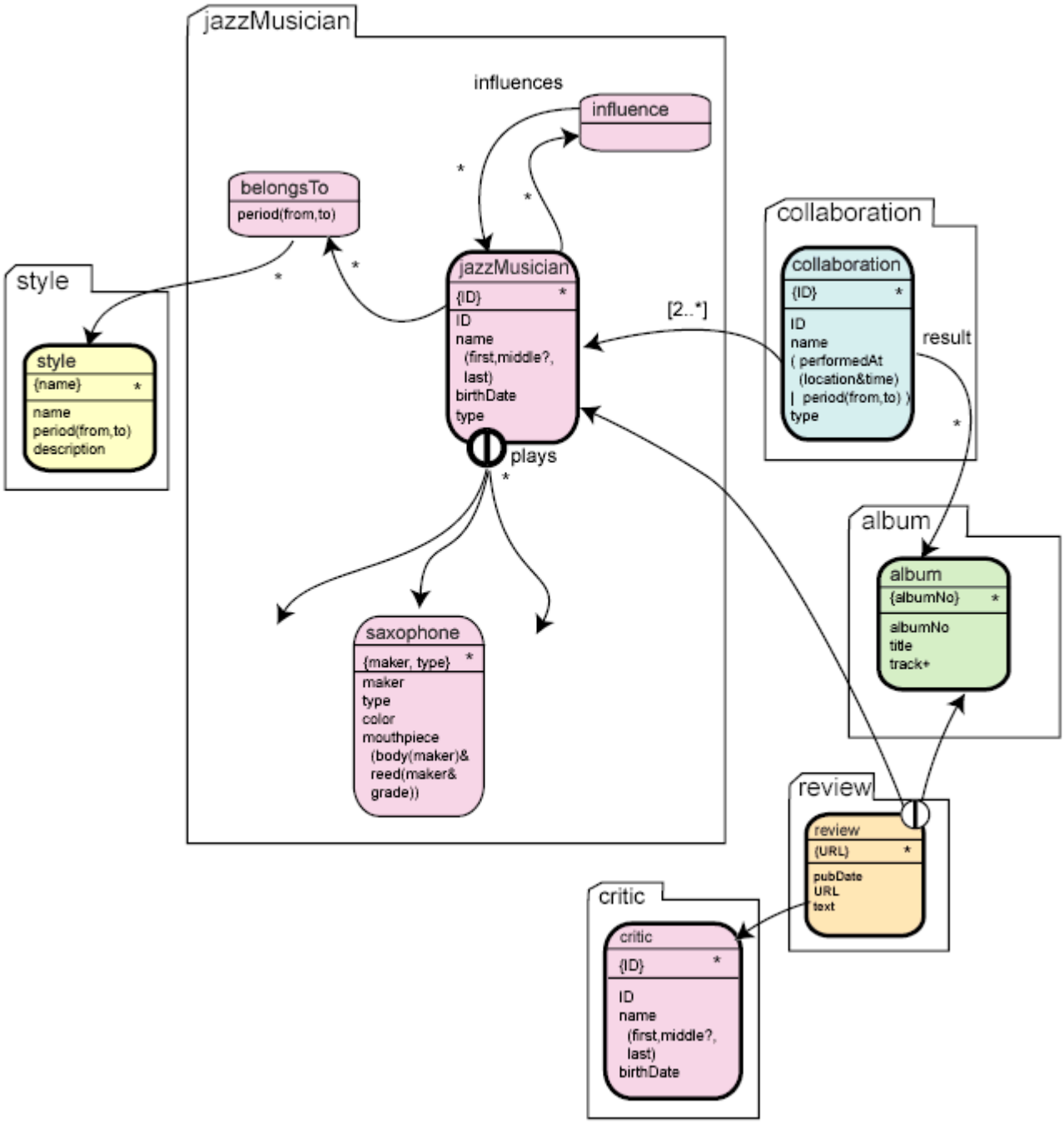
To prepare the model for implementation with XML, we resolve all `is_a` relations. Because DTDs and XML Schema do not really support inheritance, we have to find solutions for the various `is_a` relations. (DTDs do not have an inheritance mechanism at all; XML Schema cannot handle multiple inheritance.) We have the following options:

- Separate implementation of parent and child. For example, we could implement separate `person` documents which would constitute a generic person data base. `jazzMusician` and `critic` instances would have to refer to these `person` instances.
- Inclusion of parent properties in the child class. For example, we could include the properties of `person` into the asset types `jazzMusician` and `critic`.
- Inclusion of child properties in the parent class. The child type would be stored in an extra property in the parent instance. For example, we could represent `instrumentalist`, `jazzSinger`, and `jazzComposer` in a generic document type `jazzMusician` and indicate the type of musician in a special property. However, we would suffer some information loss: because jazz singers and composers do not necessarily play an instrument, we would have to use the *-cardinality for the connection to `instrument`, and not the +-cardinality. The constraint that a instrumentalist must play at least one instrument would be lost. We would have to represent this through an

extra constraint that depends on the type of musician. We shall later see how to formulate this sort of constraint.

- As an further possibility, the `is_a` relations could be implemented similarly to an aggregation that would, for example, allow a jazz musician to be a composer, a singer, an instrumentalist or any combination of these. The problem of cardinality (* or +) would not appear if the instruments are only allowed in the context of the instrumentalist. This would be possible using the `xs:all` element of XML Schema, whereby `jazzMusician` could be an element whose schema definition contains an `xs:all` element that in turn contains elements `Instrumentalist`, `Composer` and `Singer`.

After applying these operations, our model could look like this:



Here we have resolved the generic `instrument` asset into single instrument types such as `saxophone`, `guitar`, `trombone`, etc. The different instruments are just too different to be represented in one generic type. The consequence is that the asset type `instrumentalist` has a connection to all of these types. This is done with a cluster, a construct already discussed above.

Structurally, our conceptual model is now complete. In later chapters we discuss how additional constraints and operations can be defined. But before we do so, we discuss how to derive XML

schemas from the conceptual model. We give a short introduction to XML Schema in the next section (*Introduction to XML Schema*).

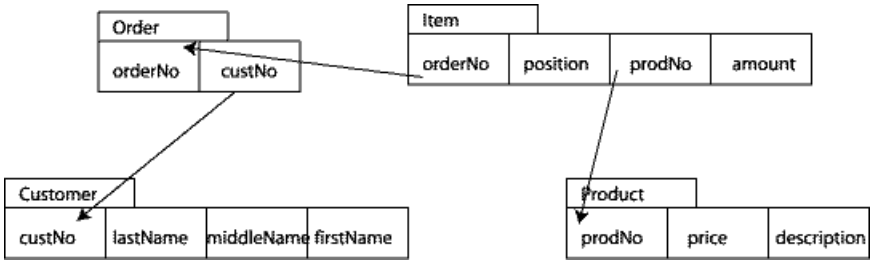
Reverse Engineering of Relational Schemas

In some cases it is necessary to reengineer existing relational schemas. This is especially the case if we plan to convert existing relational data into XML or to map relational structures onto XML structures (for example, using Tamino's X-Node). If the original conceptual model is not available, we should try to reconstruct such a model from the relational schemas. This usually results in XML data structures of higher quality than the naive approach of mapping relational data directly onto XML.

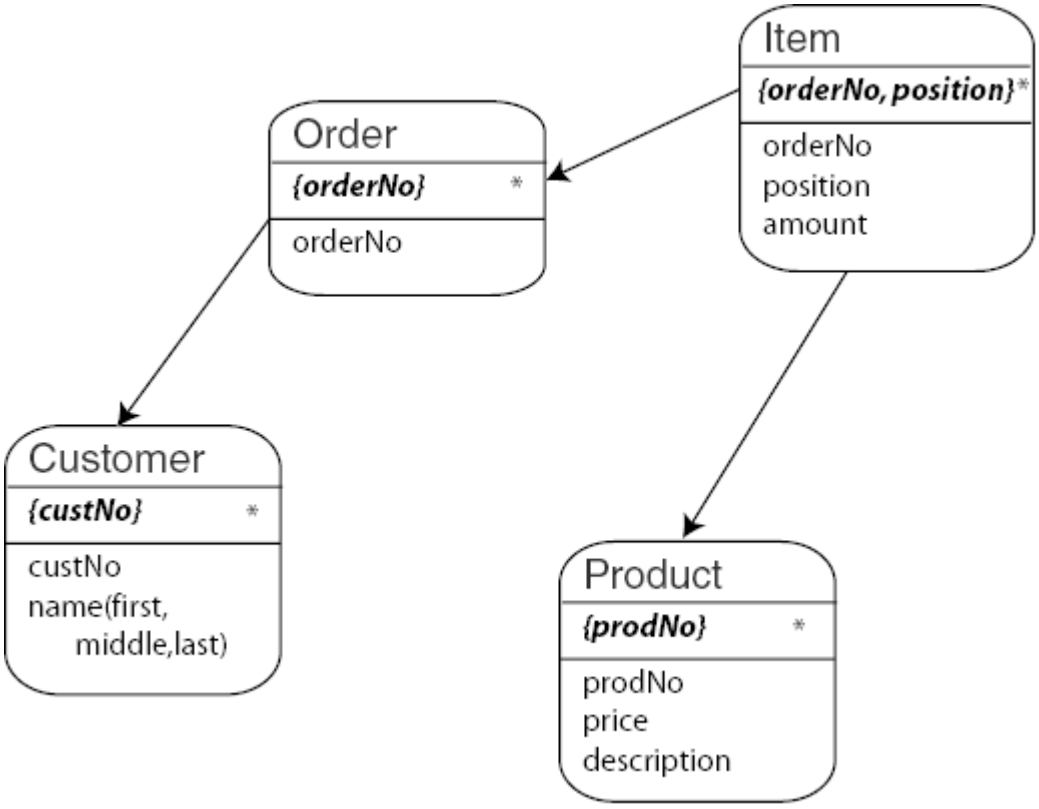
Transforming relational schemas into an Asset Oriented Model is almost trivial:

- Each table is mapped to an asset.
- Each table column (except foreign keys) is mapped to an asset property.
- Each corresponding foreign/primary key pair is represented as an arc pointing from the owner of the foreign key to the owner of the primary key.

The following is a classical example for a relational schema:

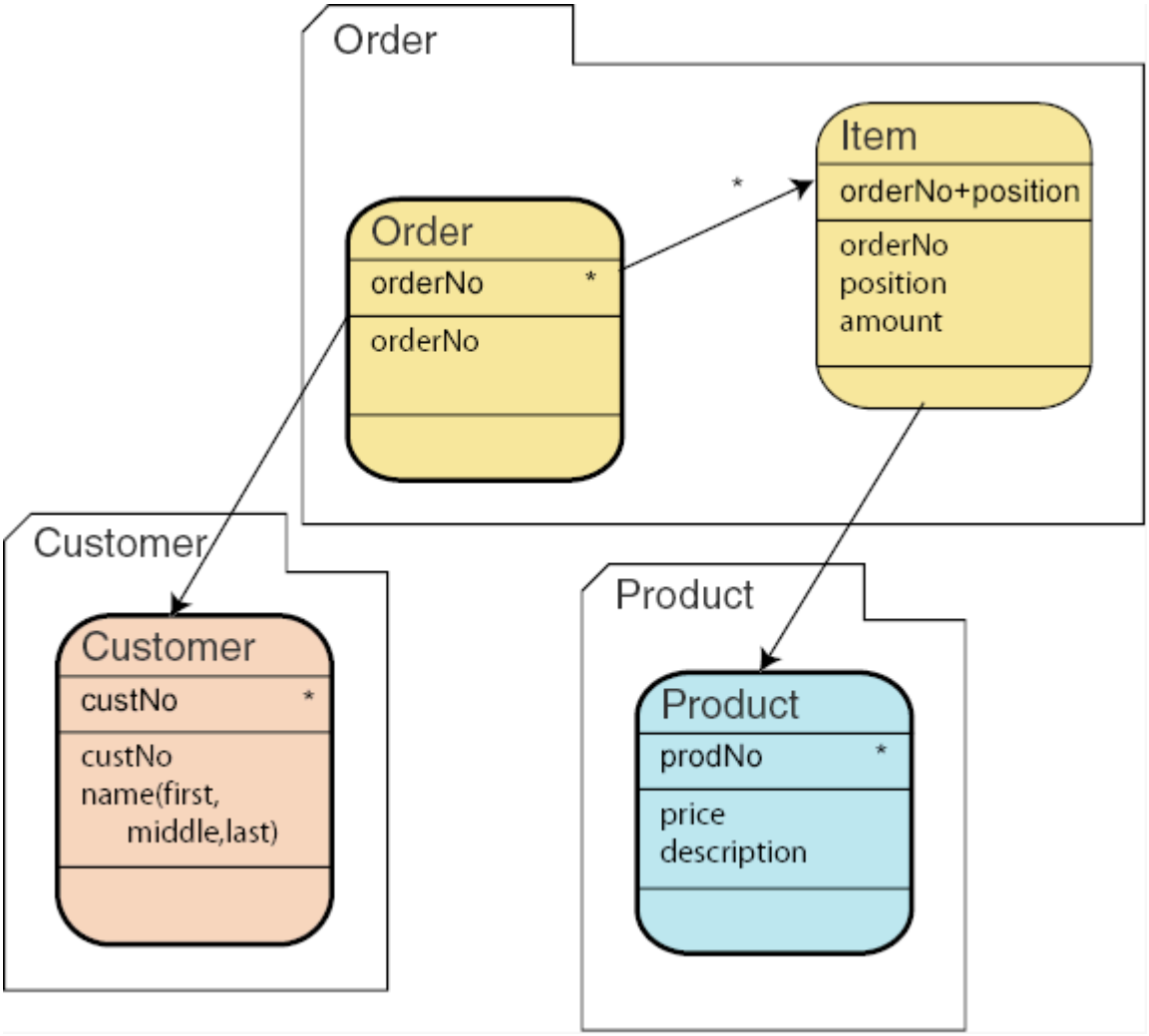


By applying the above rules, we arrive at the following asset-oriented model:



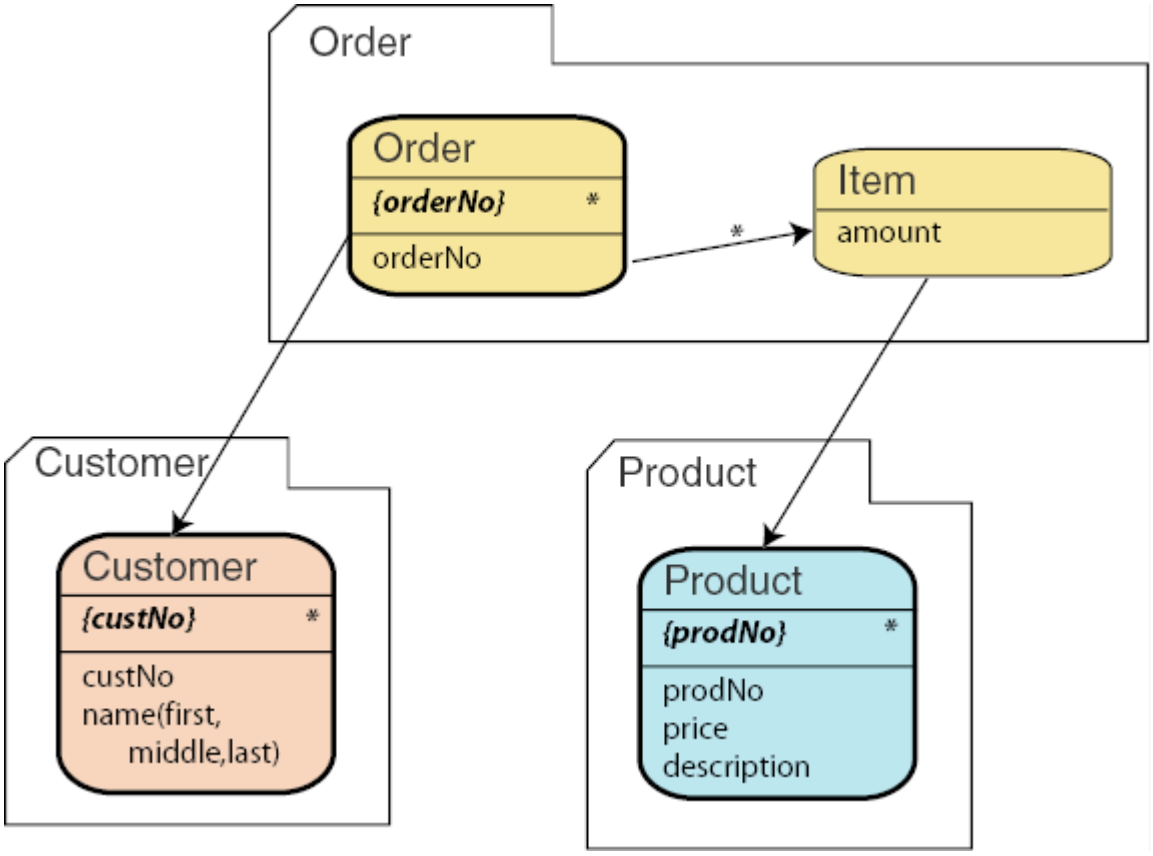
Note that we cheated a bit here. We regrouped the three columns `lastName`, `middleName`, and `firstName` into a complex property called `name`. Relational schemas flatten complex data structures such as `name` (to achieve First Normal Form) and thereby lose structural information. Regrouping of such columns, however, needs an understanding of the semantics of the model and cannot be prescribed by simple rules.

In the next step, we determine our business objects and group the assets around them, as discussed above in the section ***Determining business objects***.



We have determined three business objects or business documents: `Customer`, `Order` and `Product`, and have grouped `Item` together with `Order`. Because the asset `Order` is the identifying asset of the business object `Order` we have reversed the arc between `Order` and `Item` to indicate that `Item` belongs to `Order`. Because we reversed this arc, we decorated it with an asterisk (see the section *Determining business objects* above). This makes sense: an order can have several items.

Using this technique, we finally arrive at well-structured XML documents representing not flat tables but complex business objects. One interesting detail is that with an implementation in XML the property `position` of asset `Item` becomes redundant. Sequences of XML elements are well ordered; rows in a relational table, in contrast, are not, and therefore require an attribute such as `position` in order to establish an ordered sequence. And, of course, `orderNo` in asset `Item` also becomes redundant because it is already contained in asset `Order`.



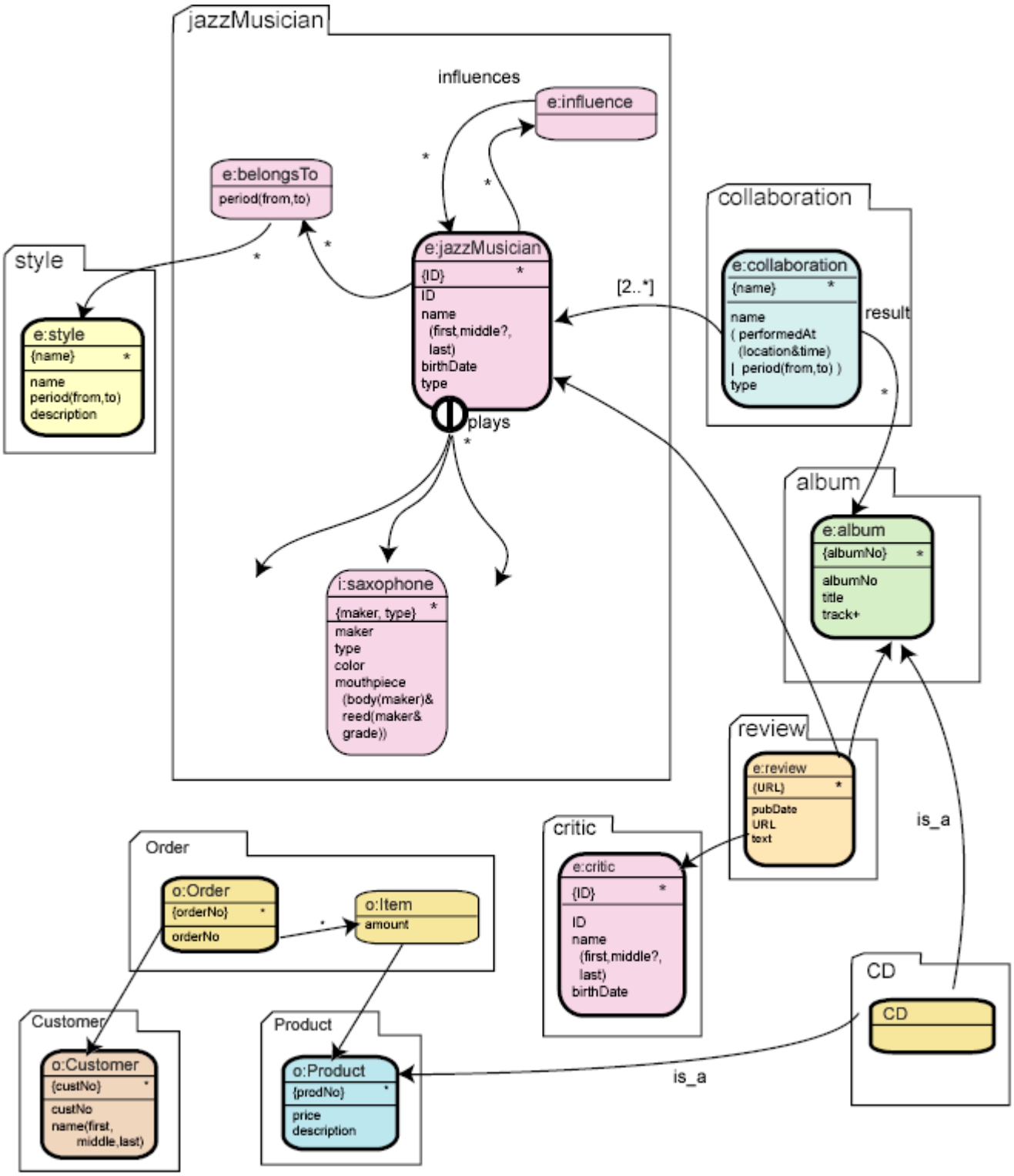
Models and Namespaces

So far, we have not discussed how models are identified. A simple model name would be not a good choice because it probably would not be unique within a global context. A better idea is to use a URI as model identification, for example, a URI based on a domain name. In our case we could choose `http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia` for the jazz model, and `http://www.softwareag.com/tamino/doc/examples/models/order/reengineered` for the order model. This technique allows us to identify models uniquely. At the same time, such an identifier defines a default namespace for the respective model. Because asset names are unique within a model, the combination of namespace and local name identifies each asset uniquely within a global context.

Let us assume that we want to separate our jazz encyclopedia model into two models: one for the core jazz encyclopedia, and another model in which we define musical instruments. The idea behind this is that we could reuse the model for musical instruments in other contexts, too, for example in a knowledge base about classical music. We would establish a model for musical instruments under a separate namespace, for example, under `http://www.softwareag.com/tamino/doc/examples/models/instruments`. Our original jazz en-

cyclopedia model could be reconstructed by merging the now instrument-less jazz encyclopedia model with the new instrument model.

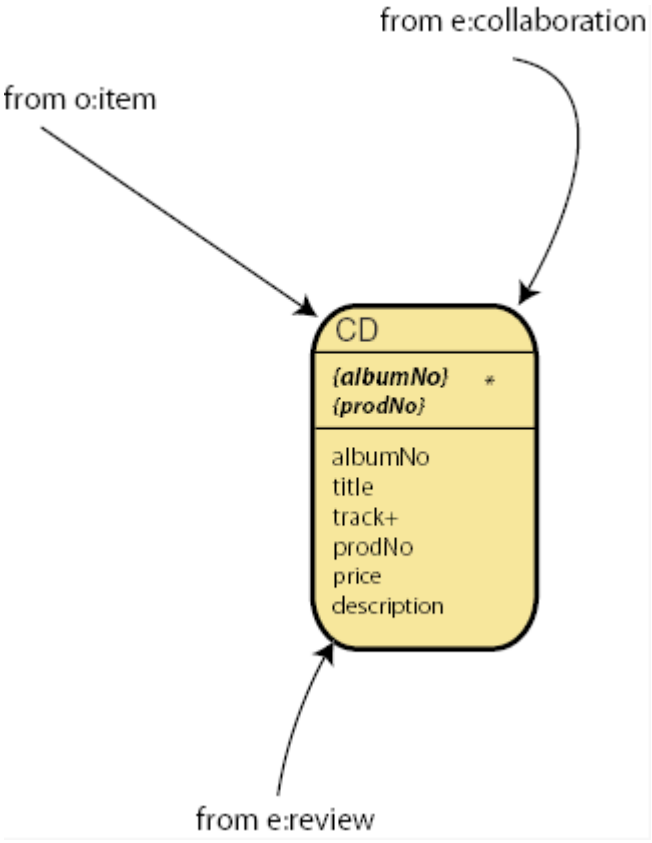
And this is where the concept of namespaces really becomes important: when we start to merge models. Let us assume that we want to create a new model for a record shop where we want to sell jazz CDs on the Web. Instead of defining everything from scratch, we import the jazz encyclopedia model (which already imports the instrument model) and the order model. We then create a new asset, namely `CD`, which inherits its properties from both the `album` asset in the jazz model and the `product` asset in the order model.


jazzMusician
influences
e:influence
e:belongsTo
period(from,to)
style
e:style
{name}
name
period(from,to)
description
e:jazzMusician
{ID}
ID
name
(first,middle?,
last)
birthDate
type
plays
collaboration
e:collaboration
{name}
name
(performedAt
(location&time)
| period(from,to))
type
result
[2..*]
i:saxophone
{maker, type}
maker
type
color
mouthpiece
(body(maker)&
reed(maker&
grade))
album
e:album
{albumNo}
albumNo
title
track+
review
e:review
{URL}
pubDate
URL
text
is_a
critic
e:critic
{ID}
ID
name
(first,middle?,
last)
birthDate
Order
o:Order
{orderNo}
orderNo
o:Item
amount
Customer
o:Customer
{custNo}
custNo
name(first,
middle,last)
Product
o:Product
{prodNo}
price
description
CD
CD
is_a

Model name:	Record Shop
Namespaces:	http://www.softwareag.com/tamino/doc/examples/models/jazz/shop
	e=http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia
	i=http://www.softwareag.com/tamino/doc/examples/models/instruments
	o=http://www.softwareag.com/tamino/doc/examples/models/order/reengineered

What happened here? The new model was defined with the default namespace `http://www.softwareag.com/tamino/doc/examples/models/jazz/shop`, which also identifies it uniquely. In addition, the model declares three namespace prefixes. The prefix "e" is assigned to the namespace of the jazz model, the prefix "i" is assigned to the namespace of the instrument model, and the prefix "o" is assigned to the namespace of the order model. All names of the assets imported from the jazz model are prefixed with "e:", all names of the assets imported from the instrument model are prefixed with "i:", and all names of the assets imported from the order model are prefixed with "o:".

What remains to do is to resolve the `is_a` relationships for the asset `CD`. This results in the following asset definition:



Here, we have inherited properties across namespaces. The properties from both `e:album` and `o:Product` are incorporated into asset `CD` and belong now to the namespace of the record shop model.

What if we have name clashes between the inherited properties? Well, this problem is not specific to namespaces but can also occur during multiple inheritance in a single namespace. The conflict is resolved by combining the conflicting properties by intersection. If there are incompatible property definitions, the intersection is empty, and the property is discarded. Of course, it is always possible to override inherited properties locally.

3 Introduction to XML Schema

- Datatypes 30
- Namespaces and Wildcards 43
- The Structure of a Schema Definition 45
- Reuse Mechanisms 46
- Elements vs. Attributes 48

W3C released the Recommendation for XML Schema in May 2001 (***http://www.w3.org/TR/xmls-chema-0***). The definition of the standard not only took into account existing schema languages such as XSchema, DDML, XML-Data, XDR and SOX, but also relied on the active participation of wide parts of the IT industry, especially database manufacturers. Most XML communities are now moving towards XML Schema.

If you are new to XML Schema, it is best to think of it as *DTD + Datatypes + Namespaces* and worry about the rest later. As you are probably already familiar with DTDs, we begin with datatypes.

Datatypes

The introduction of a full type system (***http://www.w3.org/TR/xmlschema-2/***) for elements and attributes is probably the most important aspect of XML Schema. It includes the attribute types already familiar from DTDs, but also introduces a wide range of datatypes taken from SQL and programming languages.

XML Schema differentiates between simple datatypes and complex datatypes. Complex datatypes are - as the name indicates - composed of other, simpler datatypes and are only applicable to XML elements, because only elements can contain child nodes. Simple datatypes, at the other hand, are applicable to both elements and attributes.

Simple Types

Simple datatypes are either built-in datatypes, or are derived from these datatypes. Each simple datatype is characterized by a primitive (built-in) datatype on which it is based, and by a set of constraining *facets* that are applied to the Value Space or the Lexical Space of the datatype.

Value Space and Lexical Space

XML Schema's type system makes a clear distinction between *value space* and *lexical space*. Whereas the value space consists of an abstract collection of valid values for a datatype, the lexical space contains the lexical representations of these values – i.e. the tokens that appear in the XML document.

For example, canonical lexical representations for an item of datatype `boolean` are the strings"true"and"false". But alternative string representations such as "1" and "0" are also valid lexical representations. The value space for the datatype `boolean`, at the other hand, contains the Boolean values true and false.

Primitive Datatypes

XML Schema defines a large number of built-in primitive datatypes, all of which are supported by Tamino.

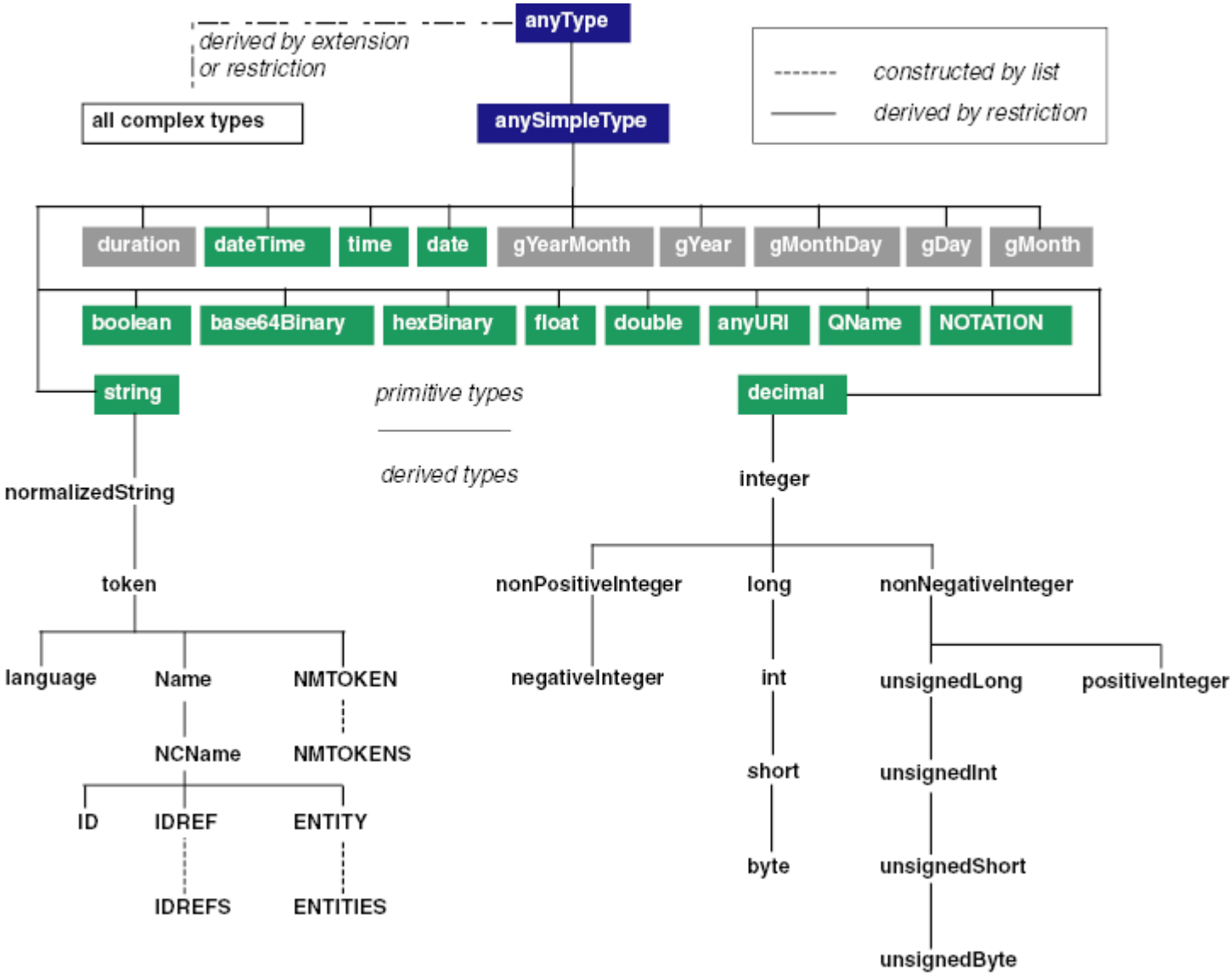
Datatype	Description	Lexical representation	SQL equivalent
string	Character string of unlimited length.	A short string	VARCHAR
boolean	Boolean value.	"true","false", "1", "0"	BIT
decimal	Decimal number. A minimum precision of 18 digits is supported by conforming processors.	"-1.23", "126.54", "0.0", "+100000.00", "210"	DECIMAL
float	A single-precision 32-bit floating point type according to IEEE 754-1985. Special values are positive and negative zero, positive and negative infinity, and not-a-number.	"-1E4", "127.433E12", "12.78e-2", "12", "0", "-0", "INF", "-INF", "NaN"	REAL
double	A double-precision 64-bit floating point type according to IEEE 754-1985.	"-1E4", "127.433E12", "12.78e-2", "12", "0", "-0", "INF", "-INF", "NaN"	DOUBLE
duration	Specifies a period of time. The value space is a six-dimensional space where the coordinates designate the Gregorian year, month, day, hour, minute, and second.	The lexical representation follows the format `PnYnMnDTnHnMnS`. An optional fractional part for seconds is allowed. Negative durations are allowed, too. *Examples:* "PT1H3M13.5S" "P1Y5M" "-PT3H"	INTERVAL
time	A specific time of day as defined in section 5.3 of ISO 8601	The lexical representation follows the format `hh:mm:ss` An optional fractional part for seconds is allowed. Additionally, a time zone can be specified: "Z" for UTC, or a signed time difference in the format `hh:mm`. *Examples:* "05:20:23.2" "13:20:00-05:00"	TIME
date	A Gregorian calendar date according to section 5.2.1 of ISO 8601.	The lexical representation follows the format `CCYY-MM-DD`. To accommodate values outside the range 1-9999, additional digits and a negative sign can be added to the left. (The year 0000 is prohibited.)	DATE

Datatype	Description	Lexical representation	SQL equivalent
		Example: "1999-05-31"	
dateTime	A specific instant of time (a combination of date and time) as defined in section 5.4 of ISO 8601.	The lexical representation follows the format `CCYY-MM-DDThh:mm:ssZ`, where the character "T" separates date from time and "Z" denotes an optional time zone. *Examples:* "1999-05-31T13:20:00-05:00" "2001-12-01T05:20:23.2"	TIMESTAMP (slightly different lexical representation)
gYearMonth	Represents a specific Gregorian month in a specific Gregorian year.	The lexical representation follows the format CCYY-MMZ. "Z" denotes an optional time zone. *Examples:* "2001-05"	DATE
gYear	Represents a Gregorian year.	The lexical representation follows the format CCYYZ. "Z" denotes an optional time zone. *Examples:* "1984"	DATE
gMonthDay	Specifies a recurring Gregorian date.	The lexical representation follows the format --MM-DDZ. "Z" denotes an optional time zone. *Examples:* "--04-01"	DATE
gMonth	Denotes a Gregorian month that recurs every year.	The lexical representation follows the format --MM--Z. "Z" denotes an optional time zone. *Examples:* "--07--"	DATE
gDay	Denotes a Gregorian day that recurs every month.	The lexical representation follows the format ---DDZ. "Z" denotes an optional time zone. *Examples:* "---13"	DATE

Datatype	Description	Lexical representation	SQL equivalent
hexBinary	Arbitrary hex-encoded binary data.	"9a7f", "FFFF3", "0100"	BINARY/BLOB
base64Binary	Base64-encoded arbitrary binary data. The entire binary stream is encoded using the Base64 Content-Transfer-Encoding defined in Section 6.8 of RFC 2045.		BINARY/BLOB
anyURI	A Uniform Resource Identifier reference (URI).		VARCHAR
QName	An XML qualified name consisting of namespace name and local part.		VARCHAR
NOTATION	Represents the NOTATION attribute type from XML attributes.	This datatype is abstract; users must derive their own datatype from it.	VARCHAR

In addition to these primitive datatypes, the Recommendation also defines built-in datatypes that are derived from these primitive datatypes by applying constraining facets. For example, the datatype `nonNegativeInteger` is derived from the datatype `integer` by constraining its value space to non-negative values, i.e. by applying the constraining facet `minInclusive value="0"`.

Another set of built-in datatypes is constructed from other built-in datatypes by list extension. The datatypes NMTOKENS, IDREFS, and ENTITIES are derived in this way from NMTOKEN, IDREF, and ENTITY. An attribute or element with such a datatype can contain several values separated by blanks.



In addition to these built-in datatypes, XML Schema allows the user to define simple datatypes by applying constraining facets to existing simple datatypes, a process which is also supported by Tamino. Each facet controls a different aspect of a datatype, for example the total number of digits or the number of fractional digits for a decimal datatype.

The following constraining facets are available:

Facet	Description
length	Defines the length of a datatype value (number of characters for strings, number of octets for binary, etc.).
minLength	Lower bound for the length of a datatype value.
maxLength	Upper bound for the length of a datatype value.
pattern	Restricts the values of a datatype by constraining the lexical space to a specific pattern. Patterns are defined via regular expressions. The syntax for the specification of patterns uses almost the same tokens and escape symbols as other languages that support patterns (such as Perl).
enumeration	Constrains the value space of a datatype to the specified enumeration values.
whitespace	This is not really a constraining facet but specifies a policy for handling whitespace in input values: `preserve` keeps all whitespace characters, `replace` replaces each whitespace character with the blank character, `collapse` reduces all sequences of whitespace characters to a single blank character.
maxInclusive	Upper bound for the value space of a datatype, includes the specified value.
maxExclusive	Upper bound for the value space of a datatype, excludes the specified value.
minInclusive	Lower bound for the value space of a datatype, includes the specified value.
minExclusive	Lower bound for the value space of a datatype, excludes the specified value.
totalDigits	Maximum total number of decimal digits in the values of datatypes derived from datatype decimal.
fractionDigits	Maximum number of decimal digits in the fractional part of values of datatypes derived from decimal.

Derived Built-In Datatypes

In addition to the primitive built-in datatypes shown above, the following derived datatypes are also available in XML Schema:

Datatype	Derived from	Description	SQL equivalent
normalizedString	string	Cannot contain carriage return (#xD), line feed (#xA) or tab (#x9) characters.	VARCHAR VARWCHAR
token	normalizedString	Cannot contain line feed (#xA) or tab (#x9) characters, cannot have leading or trailing spaces (#x20) and cannot have internal sequences of two or more spaces.	VARCHAR VARWCHAR
language	token	Language identifiers as defined by ISO 639 and ISO 3166.	VARCHAR VARWCHAR
NMTOKEN NMTOKENS Name NCName	token	Represent the corresponding attribute type from XML 1.0 (DTD).	VARCHAR VARWCHAR

Datatype	Derived from	Description	SQL equivalent
ID IDREF IDREFS ENTITY ENTITIES			
integer	decimal	The standard mathematical integer datatype of arbitrary size. Derived from datatype `decimal` by setting the facet fractionDigits to 0.	no equivalent (value range too big)
nonPositiveInteger	integer	Integer less than or equal to zero.	no equivalent
negativeInteger	nonPositiveInteger	Integer less than zero.	no equivalent
long	integer	Integer in the range from -9223372036854775808 to 9223372036854775807.	BIGINT
int	long	Integer in the range from -2147483648 to 2147483647.	INTEGER
short	int	Integer in the range from -32768 to 32767.	SMALLINT
byte	short	Integer in the range from -128 to 127.	TINYINT
nonNegativeInteger	integer	Integer greater than or equal to zero.	no equivalent
unsignedLong	nonNegativeInteger	Integer in the range from 0 to 18446744073709551615.	no equivalent
unsignedInt	unsignedLong	Integer in the range from 0 to 4294967295.	no equivalent
unsignedShort	unsignedInt	Integer in the range from 0 to 65535.	no equivalent
unsignedByte	unsignedShort	Integer in the range from 0 to 255.	TINYINT
positiveInteger	nonNegativeInteger	Integer greater than zero.	no equivalent

User-Defined Datatypes

As we have already mentioned above, it is possible for the user to restrict built-in datatypes even further. Let us look at an example. We want to declare a schema for the business object `jazzMusician`. We choose to represent the property `type` as an attribute. Since only three values are allowed, we want to declare the attribute accordingly, restricting its value range to "instrumentalist", "jazzSinger", and "jazzComposer". We can achieve this with the following definition:

```
<xs:attribute name = "type">
  <xs:simpleType>
    <xs:restriction base = "xs:NMTOKEN">
      <xs:enumeration value = "instrumentalist"/>
      <xs:enumeration value = "jazzSinger"/>
      <xs:enumeration value = "jazzComposer"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
```

Here we declare a simple datatype on the fly. This new anonymous datatype, which is only used for the attribute with the name `type`, is derived from the built-in datatype `xs:NMTOKEN` by restriction. We then use three occurrences of the `xs:enumeration` facet to define the three possible values.

Similarly, we could define an element `grade` (for the saxophone mouthpiece):

```
<xs:element name = "grade">
  <xs:simpleType>
    <xs:restriction base = "xs:decimal">
      <xs:fractionDigits value = "1"/>
      <xs:totalDigits value = "2"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Here we use a restricted form of the built-in datatype `xs:decimal`; we only allow one decimal digit before and one decimal digit after the decimal point.

Instead of using a derived simple type anonymously, we can also give it a name:

```
<xs:simpleType name="gradeType">
  <xs:restriction base = "xs:decimal">
    <xs:fractionDigits value = "1"/>
    <xs:totalDigits value = "2"/>
  </xs:restriction>
</xs:simpleType>
```

Such a type definition must be made at the schema level, as a direct child node of the `<xs:schema>` element. Later we can refer to that type definition by quoting the type name:

```
<xs:element name = "grade" type="gradeType"/>
```

Complex Datatypes

In contrast to the `simpleType` declaration there is also a `complexType` declaration. Complex datatypes are used to combine several XML elements and attributes into one datatype. Thus they are a central element in schema definition. In particular, complex datatypes are used to define elements that contain child elements and/or have attributes. As with simple datatypes, we can use complex types here as anonymous types, defined on the fly, or as explicitly named types for later reference.

Here is an example:

```
<xs:complexType name="periodType">
  <xs:sequence>
    <xs:element name = "from" type = "xs:date"/>
    <xs:element name = "to" type = "xs:date"/>
  </xs:sequence>
</xs:complexType>
```

and

```
<xs:element name = "period" type="periodType"/>
```

Here we simply define a complex element that contains a period of time: the first child element `from` contains the start date, the second child element `to` contains the end date. The elements are bound together with the `xs:sequence` connector. This constructor requires that instance documents always use the prescribed sequence of elements. For example:

```
<period>
  <from>1917-05-23</from>
  <to>1918-11-05</to>
</period>
```

is valid, whereas

```
<period>
  <to>1918-11-05</to>
  <from>1917-05-23</from>
</period>
```

is invalid.

Here is another example:

```
<xs:element name = "performedAt">
  <xs:complexType>
    <xs:all>
      <xs:element name = "location" type = "xs:normalizedString"/>
      <xs:element name = "time" type = "xs:dateTime"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Here we have used the connector `xs:all`. This connector creates a bag (i.e. an unordered list) of elements, so both of the following instances are valid:

```
<performedAt>
  <location>Dixie Park</location>
  <time>1910-03-27T17:15:00</time>
</performedAt>
```

```
<performedAt>
  <time>1910-03-27T17:15:00</time>
  <location>Dixie Park</location>
</performedAt>
```

As you can see, XML Schema makes it easy to specify unordered sequences. With the DTD one had to resort to specifying alternatives of all possible permutations of the child elements; quite a laborious process.

The third possibility to connect elements is the `xs:choice` connector, which specifies alternatives.

All of these connectors can be nested to create complex element structures. There is one exception: the `xs:all` connector cannot directly contain other connectors (but it can contain other complex elements). In the following example we define an element that is either a `period` or a `performedAt` type element:

```
<xs:element name = "collaborationContext">
  <xs:complexType>
    <xs:choice>
      <xs:sequence>
        <xs:element name = "from" type = "xs:date"/>
        <xs:element name = "to" type = "xs:date"/>
      </xs:sequence>
      <xs:all>
        <xs:element name = "location" type = "xs:normalizedString"/>
        <xs:element name = "time" type = "xs:dateTime"/>
      </xs:all>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Caution: Schema designers should always design schemas that are deterministic. A schema is deterministic if the parser can decide at each choice point which branch to take without having to look ahead in the document. For example, the particle `((a,b)|(a,c))` is not deterministic, because the parser does not have enough information when it is parsing the element `a` to decide which branch to take. As required by XML Schema, Tamino refuses to accept such a schema (with an INOXDE7909 response code). The solution is to factor the common `a` element out and redesign the particle into `(a,(b|c))`. Similarly, `((a,b)|(c&a))` is not deterministic because the `all` group `(c&a)` allows valid instances of `(a&c)`. Here we would need to redesign the particle as `((a,(b|c))|(c,a))`.

By default, an element of complex type must only contain attributes and child elements, but no other content. To allow for mixed content (i.e. text interspersed between child elements) we must

specify `mixed="true"` for the complex type definition. Thus, XML Schema allows control over the number and order of child elements within mixed content, which is not possible in a DTD.

```
<xs:element name = "track">
  <xs:complexType mixed = "true">
    <xs:sequence>
      <xs:element name = "duration" type = "xs:duration"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Here we have equipped the property `track` from the business object `album` with an additional element `duration`. This defines the duration of each track. In addition, the element `track` contains the title of each track as text content. This is possible because `xs:complexType` was specified with `mixed="true"`. Thus we would allow instances such as

```
<track>Blue Monk<duration>PT7M37S</duration></track>
```

Defining Attributes

The connectors discussed above (sequence, choice, all) are always necessary when you want to nest XML elements. Even if an element has only a single child element, the child element must be placed into a sequence. If an element has no child elements but only attributes, then we do not need these connectors. However, the type of the element is still complex.

In this case we can specify `xs:attribute` elements as children of the `xs:complexType` element. The result would be an empty element equipped with attributes (XML Schema does not have an"EMPTY" specifier as the DTD has).

Alternatively, we can specify the `xs:attribute` elements within an `xs:simpleContent` declaration, as shown in the following example:

```
<xs:element maxOccurs = "unbounded" name = "track">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base = "xs:normalizedString">
        <xs:attribute name = "duration" type = "xs:duration" use = "required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Here we have defined `duration` as an attribute of `track`.

Defining Constraints

We have also explicitly defined the cardinality constraint `maxOccurs` for the element `track`. XML Schema has two cardinality constraints: `maxOccurs` and `minOccurs`. `minOccurs` defines the minimum number of element occurrences required, `maxOccurs` the maximum. The default value for both is

1, so if nothing is specified, an element must appear once and only once. These constraints replace and extend the element modifiers as we know them from DTDs.

minOccurs	maxOccurs	DTD	Description
1	1	none	single element required
1	unbounded	+	one or more elements, at least one required
0	1	?	optional single element
0	unbounded	*	zero or more elements, optional
n	m	no equivalent	at least *n* elements, at most *m* elements.
n	unbounded	no equivalent	at least *n* elements.

Caution: A `maxOccurs` value greater than 1 is not allowed for elements that contain an `xs:all` connector, or for elements contained in an `xs:all` connector.

Complete Schema

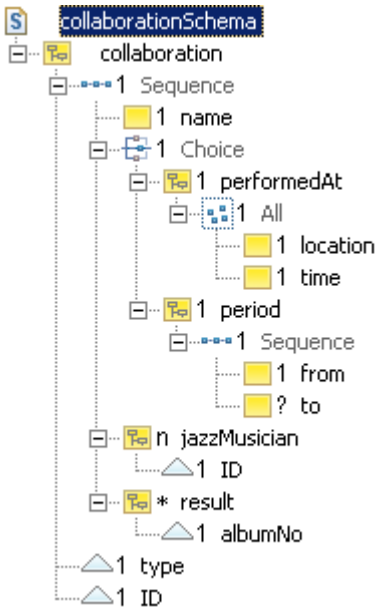
Here is a complete schema definition for the business object `collaboration`. In the XML prologue we first define a namespace prefix for XML Schema:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name = "collaboration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "name" type = "xs:NMTOKEN"/>
        <xs:choice>
          <xs:element name = "performedAt">
            <xs:complexType>
              <xs:all>
                <xs:element name = "location" type = "xs:normalizedString"/>
                <xs:element name = "time" type = "xs:dateTime"/>
              </xs:all>
            </xs:complexType>
          </xs:element>
          <xs:element name = "period">
            <xs:complexType>
              <xs:sequence>
                <xs:element name = "from" type = "xs:date"/>
                <xs:element name = "to" type = "xs:date"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
       <xs:element name = "jazzMusician" type = "xs:NMTOKEN" maxOccurs = "unbounded" ↩
minOccurs = "2"/>
        <xs:element name = "result" type = "xs:NMTOKEN" maxOccurs = "unbounded" ↩
minOccurs = "0"/>
      </xs:sequence>
```

```
      <xs:attribute name = "type" use = "required">
        <xs:simpleType>
          <xs:restriction base = "xs:NMTOKEN">
            <xs:enumeration value = "jamSession"/>
            <xs:enumeration value = "project"/>
            <xs:enumeration value = "band"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The root element is `collaboration`. This element has an attribute `type`, and a sequence of child elements consisting of the element `name` and a choice of the elements `performedAt` and `period`, followed by the elements `jazzMusician` and `result`.

XML Schema gets a bit lengthy at times. Tamino's schema editor can give you a much better overview of the structure of a document, so we shall use this representation for structural overviews.



Namespaces and Wildcards

Namespace

Namespaces were introduced into XML in order to avoid name clashes between different vocabularies. The concept of namespaces allows us, for example, to mix language elements from SVG with those of XHTML, or to process our own XML documents with XSLT. Last but not least, it allows us to define schemas with XML Schema, because by using namespace prefixes we can differentiate between XML Schema tags and our own element names.

Each namespace identifier must be globally unique - usually a URI is used for that purpose. The definition within XML documents is simple: a document node is equipped with an `xmlns` attribute to define the default namespace. Similarly, additional namespaces can be introduced by defining namespace prefixes using attributes of the form `xmlns:<prefix>="<uri>"`. The scope of such a definition is the node in which it is defined plus all child elements, unless a child element overrides it with another namespace declaration. So, if we declare namespaces in the root element of a document their scope usually is the whole document.

We say that the name of an element is *qualified* if that element is within the scope of a default namespace declaration, or if its name is specified with a namespace prefix within the scope of the namespace declaration for this prefix. An attribute is qualified if its name is equipped with a namespace prefix.

Now, let's get back to XML Schema. One major advantage of XML Schema over DTDs is that XML Schema fully supports XML namespaces. In order to do so, XML Schema introduces the concept of the *target namespace*. Each schema file may declare at most one target namespace. All elements defined in this schema file must belong to this namespace. So, a schema file may define either namespace-less elements, or elements belonging to the specified target namespace.

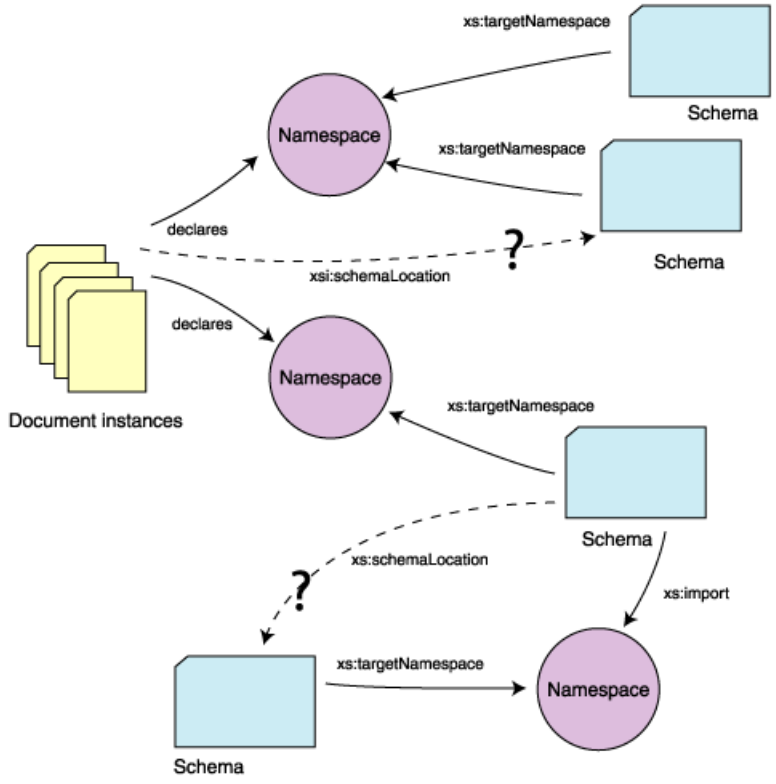
Does this mean that we cannot define multi-namespace schemas? No; the emphasis is here on schema *file*. We can always compose multi-namespace schemas by importing (see below) other schema files into a schema.

Importing Foreign Namespaces

The `import` statement is used in XML Schema to compose multi-namespace schemas. A typical example is given below:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema ↩
targetNamespace="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
           ↩
xmlns="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:i="http://www.softwareag.com/tamino/doc/examples/models/instruments">
  <xs:import ↩
namespace="http://www.softwareag.com/tamino/doc/examples/models/instruments"
                schemaLocation="instrument.xsd"/>
  ...
</xs:schema>
```

`import` statements are always given at the beginning of a schema clause. An `import` statement always specifies a namespace to import, and may optionally specify an associated schema file with the `schemaLocation` attribute. Note that this attribute only gives a hint to the XML processor about the location of the schema file associated with the imported namespace. XML processors are not required to use this attribute, but may use their own logic to find the associated schema. The same is, by the way, true for the `xsi:schemaLocation` attribute in document instances.



Namespaces are used in XML Schema to tie document instances to schemata and to tie schemata to schema components. The `schemaLocation` attribute is a mere hint for an XML processor where to find the schema definition.

Wildcards

Another mechanism to allow for foreign namespaces is the use of wildcards. A *wildcard* (i.e. an element or attribute of arbitrary content) can be declared with the XML Schema elements `<xs:any>` or `<xs:anyAttribute>`. This allows for the inclusion of elements and attributes from foreign schemas. For example, sections of XHTML, SVG, RDF and other content could be included into a document. A typical application would be the `description` property in the `style` asset of our jazz model, where we could use XHTML to mark up the content.

It is possible to constrain the namespace of the content of such a wildcard. This is done with the attribute `namespace`. This attribute can contain:

- either a list of namespace identifiers, each consisting of:
  - an explicit namespace URI;
  - the string "##targetNamespace", which denotes the target namespace of the current schema file;
  - the string "##local", which specifies the namespace of the respective document instance;
- or one of the following string values:
  - the string "##any", which allows any namespace. This is also the default value of the `namespace` attribute. This value is often used together with `processContents="skip"`;
  - the string "##other", which stands for any namespace other than the target namespace.

It is also possible to specify how the content of such an element should be processed by the parser:

- `processContents="strict"` causes the parser to check the wildcard instance for valid content;
- `processContents="lax"` causes the parser to check the content of the wildcard instance if there is an appropriate declaration, otherwise the element or attribute can be skipped;
- `processContents="skip"` stops the parser from checking the wildcard instance for valid content.

The Structure of a Schema Definition

Each schema file contains exactly one `<xs:schema>` element, which serves as the root element for the schema definition. Any global element may be used as the root element of a valid XML instance. The attribute `elementFormDefault` of the `xs:schema` element specifies whether locally defined elements in instances of the schema must be qualified with a namespace, either by using an explicit prefix or via the use of a default namespace in the instance. Similarly, the attribute `attributeFormDefault` of the `xs:schema` element specifies whether locally defined attributes in instances of the schema must be qualified with a namespace, either explicitly or implicitly as for elements. The `form` attribute specified on an element or attribute definition in the schema overrides the effect of the corresponding `elementFormDefault` or `attributeFormDefault` setting.

A schema clause can have several attributes, such as in:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
         ↩
xmlns="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
         ↩
targetNamespace="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
         elementFormDefault="qualified"
         attributeFormDefault="unqualified"> ...
</xs:schema>
```

The default namespace is set identical with the target namespace, and the prefix "xs:" is defined for the XML Schema namespace. This means that we do not have to prefix our own definitions within the schema, but we have to prefix all XML Schema tags with "xs:". We also specify that document instances must use elements in a qualified form, either by setting the default namespace of the document instance root element to "http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia" or by defining and using an appropriate namespace prefix.

Reuse Mechanisms

Global Elements and Attributes

In XML Schema it is possible to define elements and attributes both locally and globally. In contrast, a DTD can define elements only globally, and attributes only locally. This has always been a problem. For example, with a DTD it would not be possible to implement our musical instruments (within `jazzMusician`) correctly: both saxophone and trombone have a mouthpiece, but in the case of the saxophone the mouthpiece consists of a body and a reed, in the case of the trombone it is just the body. Obviously, a single global definition for `mouthpiece` is not appropriate. With XML Schema there is no problem: we simply define `mouthpiece` locally, once in the context of `trombone`, and once in the context of `saxophone`.

Sometimes, however, we want to reuse an element definition. In this case we define the element as a global element, as a direct child of the `xs:schema` clause. For example:

```
<xs:schema ...>
  <xs:element name = "jazzMusician">
    ...
  </xs:element>
  <xs:element name = "maker" type = "xs:string"/>
</xs:schema>
```

When we want to reuse this definition we simply use an *element reference* like this:

```
<xs:element ref = "maker">
```

The same technique is possible with attributes when we want to reuse attribute definitions.

Global Datatypes

By defining a simple or complex datatype as a child element of the `xs:schema` element, it can be used as a global datatype. The `name` attribute of the appropriate `xs:simpleType` or `xs:complexType` element must be assigned a value that can be referred to by other elements in the schema.

Element Groups and Attribute Groups

Other reuse constructs are named global element groups and named global attribute groups. Here is an example of a named element group:

```
<xs:group name="nameGroup">
  <xs:sequence>
    <xs:element name="first" type="xs:token"/>
    <xs:element name="middle" type="xs:token" minOccurs="0"/>
    <xs:element name="last" type="xs:token"/>
  </xs:sequence>
</xs:group>
```

We can then use this group definition within an element definition:

```
<xs:element name="name">
  <xs:complexType>
    <xs:group ref="nameGroup"/>
  </xs:complexType>
</xs:element>
```

Named element groups are also the preferred way to express recursive structures:

```
<xs:group name="partGroup">
  <xs:sequence>
    <xs:element name="partNo" type="xs:token"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="part" minOccurs = "0">
      <xs:complexType>
        <xs:group ref="partGroup"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:group>
```

In this example, the `part` element must be optional (`minOccurs="0"`) so that the recursion does not specify an infinite loop.

Elements vs. Attributes

In the XML community (and previously in the SGML community) there had always been a heated debate: when to use an element to model an information or data item and when to use an attribute. The question is one of the most frequently asked by developers that are new to XML. The question has been asked since SGML existed; however, we shall see that now, with XML Schema available, the answer has changed.

Let us look at an instance of the `collaboration` schema shown above:

```
<collaboration type="jamSession">
  <name>Antibes1960</name>
  <performedAt>
    <location>Antibes</location>
    <time>1960-07-13T20:00:00<time/>
  </performedAt>
  <jazzMusician>MingusCharles</jazzMusician>
  <jazzMusician>DolphyEric</jazzMusician>
  <jazzMusician>PowellBud</jazzMusician>
  <result>CD53013</result>
</collaboration>
```

This layout is almost completely based on the use of elements. Only the property `type` has been implemented as an attribute.

A completely attribute-based layout could look like the following:

```
<collaboration
     type="jamSession"
     name="Antibes1960"
     performedAt-location="Antibes"
     performedAt-time="1960-07-13T20:00:00"
     jazzMusician="MingusCharles DolphyEric PowellBud"
     result="CD53013" />
```

Note that we have lost some structural information. Only by adopting a (custom) naming pattern were we able to maintain the structural relationship between `performedAt`, `location` and `time`. An alternative would be to use only a single attribute such as:

```
performedAt="Antibes 1960-07-13T20:00:00"
```

However, we would lose some of the descriptive power of XML and would require a custom parser to process this attribute value.

Also, since an attribute for an element cannot appear more than once, the names of the jazz musicians have been combined here into a single attribute. This is a custom solution and would require a custom parser to process the value. Other custom solutions are possible, for example, the use of separate attributes jazzMusician-1, jazzMusician-2 etc.

Another attribute based approach makes extensive use of ID- and IDREF-attributes. The whole document is “flattened” into elements that have only attributes but no child elements:

```
<collaboration root="1">
  <root id="1"
        type="jamSession"
        name="Antibes1960"
        performedAt="2"
        jazzMusician="MingusCharles DolphyEric PowellBud"
        result="CD53013" />
  <performedAt id="2"
     location="Antibes"
     time="1960-07-13T20:00:00" />

</collaboration>
```

In this example the attribute `performedAt` would have been defined as an IDREF attribute. It identifies the element with `id="2"` which is `performedAt`.

This technique can be used to represent arbitrary structures as a list of empty elements. Basically, each XML document mimics a small relational database. While this technique offers a consistent approach to all kinds of information structures, it suffers from two drawbacks: such documents are hard to read, and they are awkward to query.

Juxtaposed to this design is to throw out attributes altogether. In our case this would require us to implement the attribute type as an element, which is easy:

```
<type>jamSession</type>
```

The truth lies somewhere between these two extremes and largely depends on context and personal taste. Is it essential that the documents should be as short as possible, or is it important to keep the processing logic simple? Is the document only to be used by machines, or is it also to be read be humans? Are the documents machine generated or are they authored by humans, and when yes, with which tools?

Especially when using DTDs for schema definition, there are some strong reasons for using attributes in some cases:

1. In DTDs, only attributes support the construction of relationships with ID/IDREF keys.
2. DTDs allow the definition of default and fixed values only for attributes.
3. A DTD does not allow type definitions (ID, IDREF, NMTOKEN, etc.) for elements.
4. In DTDs, only attributes of an element form an unordered set. This can sometimes be useful when no predetermined sequence order between information items is required.
5. Attributes are much easier to access in DOM and SAX.
6. When authoring document-centric XML in an appropriate XML editor, it is often more convenient to use attributes for annotating text. The attributes do not litter the running text, and spell checking is only applied to elements.

However, with XML Schema the reasons 1-4 no longer apply:

- Elements can now be defined as ID or IDREF.
- A wide range of datatypes is available for elements and attributes.
- Elements can now have default or fixed values.
- The `all` connector allows unordered sequences of elements.

This gives elements a certain advantage:

1. Elements can repeat. This is not possible with attributes.
2. It is possible to define choices (alternatives) between elements. This is not possible with attributes.
3. Elements are easy to extend when necessary by adding child elements or attributes.
4. Attributes of an element always form an unordered set, so it is not possible to establish a sequence order across attributes.
5. Elements can contain whitespace and delimiters; whitespace handling can be specified at the element level.
6. Elements are easier to search for in search engines.
7. When editing data-centric XML in an XML editor, storing content in attributes makes the editing process more difficult: often extra keystrokes or mouse actions are required to view the attributes.

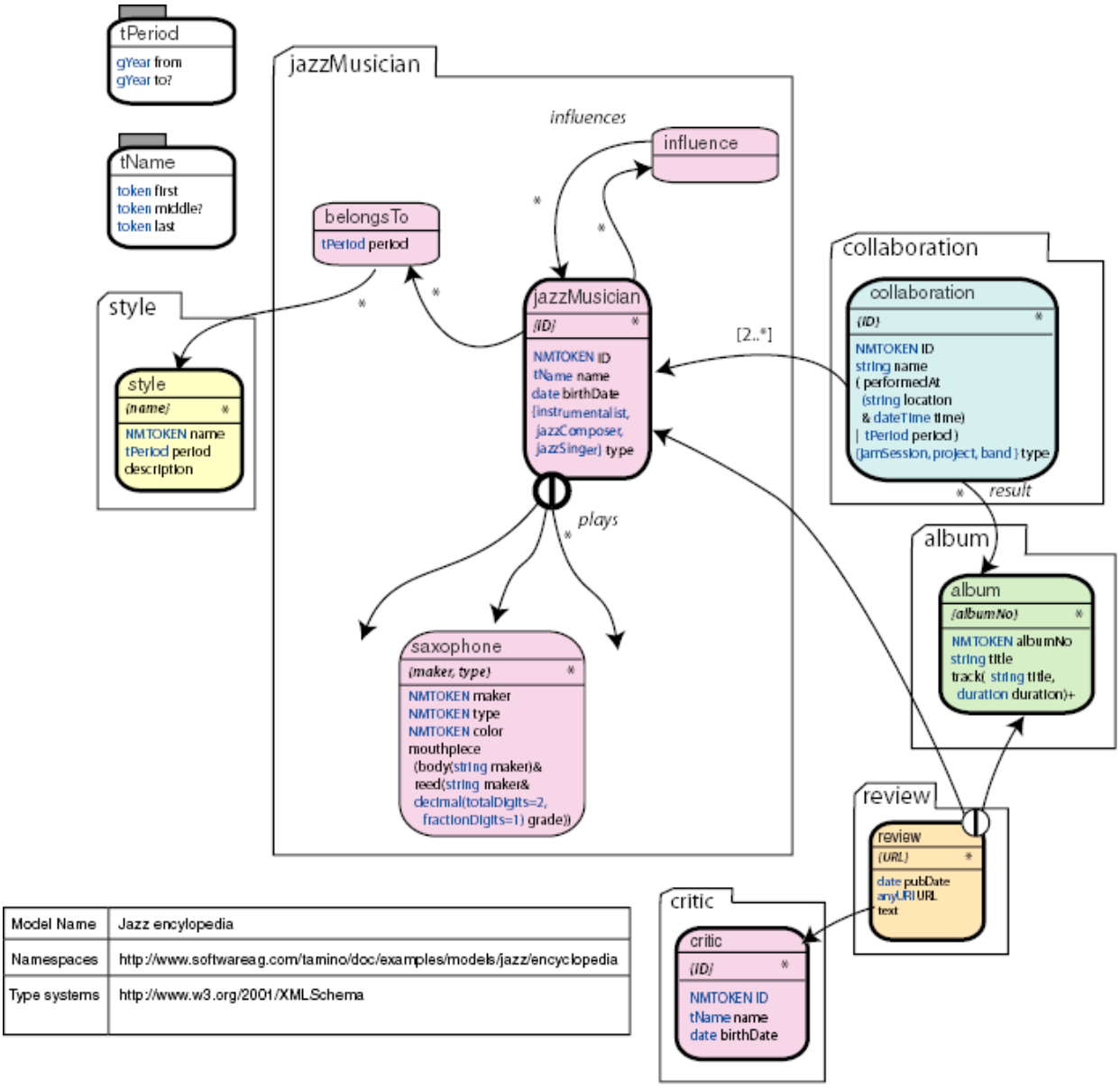
These are strong reasons for using elements. We would suggest using attributes to describe annotation only (such as language identifier, element author, element version, element ID, etc.), and using elements to represent content. However, what is content and what is metadata can depend on the context. A good definition to distinguish content from annotation is based on a suggestion by Elliot Kimber: If I removed this data, would my understanding of or my ability to comprehend the content change? If the answer is no, it's annotation, if the answer is yes, it's content.

4 From Model to Schema

- Adding Type Information 52
- Document-Centric Layout 53
- Creating a Type Library 54
- Implementing Business Objects 55
- Segmentation and Optimization 57
- Multi-Namespace Schema Composition 58
- Schema Evolution 60
- Open Content Model 61
- Versioning 62

As we have seen, with XML and XML Schema we have many options for designing XML documents. Let us return to our conceptual model.

Adding Type Information



Model Name	Jazz encylopedia
Namespaces	http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia
Type systems	http://www.w3.org/2001/XMLSchema

We are now in a position to add some type information to our model:

In the diagram, we have defined the XML Schema type system as the default type system of our model (Asset Oriented Modeling can handle multiple type systems within one model). Most of

the properties and sub-properties in this model are now prefixed with a type name (separated by a blank). All properties used as primary keys are defined with datatype `NMTOKEN`. This will save us a lot of trouble later, when we want to transport a key value in the query part of a URL. (White space character handling in URLs is awkward.)

We see, too, that the type properties in the assets `jazzMusician` and `collaboration` are defined with an enumeration as type. This would translate into the XML Schema type `xs:string` with appropriate `enumeration` facets. The property `grade` in asset `saxophone` has a type that is constrained with the facets `totalDigits` and `fractionsDigits`.

In addition, we have factored out some complex properties (`name` and `period`) as explicit types. This is done by defining the abstract assets (indicated by the grayed-out label area) `tName` and `tPeriod`. We use the names of these assets as type names in various other assets such as `jazzMusician`, `critic`, `style`, `belongsTo` and `collaboration`. Note that we have improved the definition of `tPeriod` somewhat by making the property `to` optional. This allows for open-ended periods.

Document-Centric Layout

Now we are ready to translate our conceptual model into XML Schema source code. However, the question arises, how we should best divide our model into individual schemas.

One extreme would be to create one XML document type for each asset. However, this has a disadvantage: because the existence of some asset instances can depend on the presence of other asset instances, we would require extra operations when deleting and updating assets. For example, if we wanted to delete a certain instance of `jazzMusician`, we would also have to delete the instruments he or she plays.

The other extreme would be to create a single document containing the whole model. This is even worse because such an implementation would not scale well. Such a document can become very big, and consequently various operations (loading, saving, parsing, etc.) would be very slow. Although Tamino can insert, delete, and update document subtrees, each update operation would lock the whole model and would not allow concurrent updates, even if the concurrent operation wants to update another asset.

We therefore choose the best compromise between these extremes and implement each *business object* as a single document. (In a more business-oriented scenario we would treat *business documents* such as Purchase Orders or Invoices in the same way.) This has the following advantages:

- The existence of business objects does not depend on other objects. Business objects by definition exist in their own right. Deleting a single business object, for example, does not require the deletion of other objects.
- Modifications made to a single business object do not lock the whole model. Concurrent update operations to other business objects are possible.

- This implementation fits well with current standards in application design. For example, the construction of a Java access layer for such a document would result in an implementation of the corresponding Java business object class.
- The resulting set of schemas is very intuitive. Each schema instance (i.e. each XML document) represents a business object or a business document. This is why we call this design method *document-centric*.

  Note that if a model is divided into separate object types as described here, it is possible for an application to reconstruct a view of the whole model by using appropriate XQuery join queries, or by using several X-Query commands and postprocessing the results.

Creating a Type Library

Our model contains global type definitions (the assets `tPeriod` and `tName`) that are not specific to a particular business object, and consequently in our design will not be specific to a specific schema. It makes sense to create a global type library that contains the XML Schema definition of these assets. Such a type library is created as an independent XML Schema file with the same target namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
              elementFormDefault="qualified"
              attributeFormDefault="unqualified">
  <xs:complexType name="tPeriod">
    <xs:sequence>
      <xs:element name="from" type="xs:date"/>
      <xs:element name="to" type="xs:date" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="tName">
    <xs:sequence>
      <xs:element name="first" type="xs:token"/>
      <xs:element name="middle" type="xs:token" minOccurs="0"/>
      <xs:element name="last" type="xs:token"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

This file can then be imported into the schema files that implement business objects. The XML Schema syntax to import a foreign schema file into the current schema is:

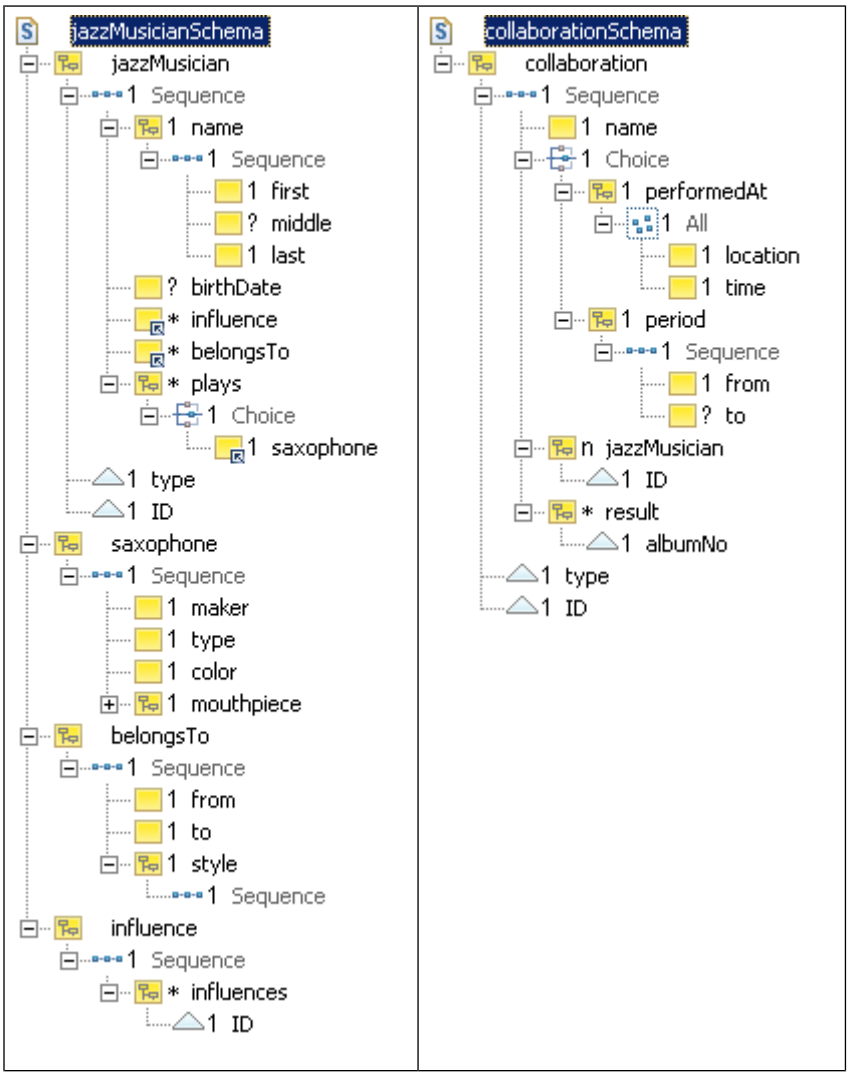
```
<xs:import namespace="..." schemaLocation = "typelib.xsd"/>
```

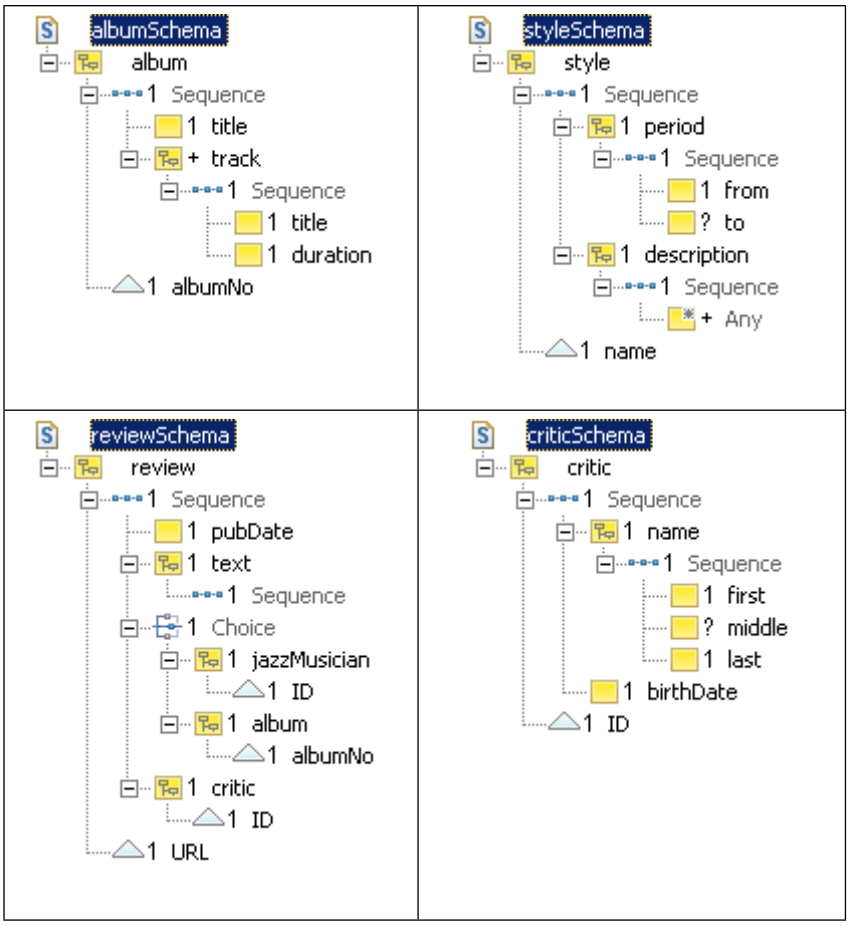
The `xs:import` clause is specified as a direct child of the `xs:schema` clause and must be specified at the very beginning of this clause. The attribute `schemaLocation` specifies the location of the imported file as a relative or absolute URL.

Implementing Business Objects

Our model now results - apart from the global type library - in the following schemas:

`album`, `collaboration`, `critic`, `jazzMusician`, `review`, `style`.





The following paragraphs discuss some implementation decisions:

- We have implemented all assets as global elements, and all properties as local elements. Aggregations of assets, such as in `jazzMusician`, are implemented via references to global elements. This allows us to identify assets and properties in schema source code easily.
- In the schema `jazzMusician`, the `instrument` cluster is implemented as a `choice` connector containing the different instruments (only `saxophone` is shown here).
- We have implemented all primary keys and qualifying properties such as `kind` and `type` as attributes. This is just for the sake of this example - you are of course free to use elements and attributes at your own discretion.
- Arcs to other business objects are implemented as a kind of foreign key. The concept of a foreign key, familiar from relational systems, is not defined in XML Schema. However, we use the term loosely here to indicate an implicit reference to an external asset. Each foreign key consists of a local element definition. The element name reflects the arc's role name (or the name of the arc's target asset when no role name is specified). This element has an attribute which matches the primary key attribute of the target asset in name and type. Again, the choice to use attributes as foreign key implementations is only for the sake of this example.
- The properties `description` of asset `style` and `text` of asset `review` are implemented as wildcards. The `any` declaration allows the element to contain XHTML markup, for example. We have

set `processContents` to "lax" for these elements and `namespace` to "http://www.w3.org/1999/xhtml", so parsers will check for valid XHTML content when an XHTML schema is available. We have also set `maxOccurs` to "unbounded" for each `any` declaration, to allow for multiple XHTML elements within a wildcard.

Segmentation and Optimization

Although this document-centric approach is the preferred way to implement a conceptual model, it is sometimes necessary to make compromises, especially when documents become too large, or when operations become inefficient.

Large documents have several drawbacks:

- Parsing a large document takes a long time. This affects almost any processing of XML documents (for example, transformation with an XSLT style sheet), because most XML processing involves parsing.
- Processing a large document with the DOM API requires a large amount of resources. The whole document is converted into object form (each document node becomes a separate object) and this whole set of objects is kept resident in memory. Recent DOM parsers feature lazy instantiation, which is less resource hungry. However, in the worst case, they require the same amount of memory as conventional DOM parsers.
- Collaborative authoring of large documents is awkward. Most database systems (and also standards for distributed authoring like WebDAV) support locking only at the document level. So when one client changes a document, the document is locked for others until the first client commits. Also, the exchange of such documents between authors can take a long time.

It therefore seems sensible to split large documents into smaller ones. In particular, this is the case when a document is subject to unrestricted growth. Take for example the document type `album` from the example above. If we opted to include the text of all reviews in the respective `album` document, we could get a nasty surprise. If a lot of people review an album, our `album` document could become very large. That is one reason why we decided to model `review` as an explicit business object.

However, segmentation can also create problems. During retrieval we need more *join* operations, and some aggregating functions become slow. For example, if we want to find out the number of albums in which a jazz musician has participated, we would first have to retrieve all collaborations of that musician, and then count the albums referenced as a result of the collaboration.

This can be improved by adding redundancy to our document base. For example, we could include an album count in each `jazzMusician` document. The downside of this is that update operations become more complicated. When we add new albums, or when we delete albums, we have to update the respective counters in the `jazzMusician` instances as well. So, tuning of schemas is always a compromise. The best way almost always depends on the frequency of updates and retriev-

als, and whether it is more important to offer fast response times for retrieval or for update, and so on. Database tuning is not an exact science, but depends very much on heuristics, experience, and skill.

Multi-Namespace Schema Composition

Let's return to the multi-namespace model defined in section **Models and Namespaces**. This model featured four namespaces:

- the default namespace `http://www.softwareag.com/tamino/doc/examples/models/jazz/shop`,
- the namespace `http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia` for the jazz knowledge base,
- the namespace `http://www.softwareag.com/tamino/doc/examples/models/instruments` for the musical instruments,
- and the namespace `http://www.softwareag.com/tamino/doc/examples/models/order/reengineered` for the order model.

How does this affect our XML schemas? The asset `CD` is defined as a separate business object, and thus results in a separate schema file with its own target namespace (`http://www.softwareag.com/tamino/doc/examples/models/jazz/shop`). We now have to implement the inherited arcs that lead to asset `CD` (from `e:collaboration`, `e:review`, and `o:item`). These arcs are implemented in the usual way within the respective schema files, in addition (and similar) to the arcs leading to `e:album` and `o:product`. Since these arcs are implemented via primary and foreign key constructs and not via reference or inclusion, all schemas stay single-namespace schemas.

Note, however, that the instruments are implemented differently. Instruments such as `i:saxophone` and `i:trombone` are part of the `jazzMusician` business object, and are consequently referred to (via an `xs:element ref=` clause) within the jazzMusician schema file. But because these instruments belong to a different model (and thus to a different namespace), they must be implemented in a schema file with target namespace `http://www.softwareag.com/tamino/doc/examples/models/instruments`. Let us assume that all instruments are defined as global elements in a schema file called *instrument.xsd*.

What we have to do then, is to import the file *instrument.xsd* into the file *jazzMusician.xsd*. And this is how it's done:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema ↩
targetNamespace="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
           ↩
xmlns:e="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
         xmlns:i="http://www.softwareag.com/tamino/doc/examples/models/instruments"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <xs:import schemaLocation="typelib.xsd"/>
  <xs:import ↩
namespace="http://www.softwareag.com/tamino/doc/examples/models/instruments"
                schemaLocation="instrument.xsd"/>
  <xs:element name="e:jazzMusician">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="e:name" type="e:tName"/>
    ...
        <xs:element name="e:plays"
                    minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:choice>
              <xs:element ref="i:saxophone"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      ...
    </xs:complexType>
  </xs:element>
  ...
</xs:schema>
```

The two `xs:import` clauses are specified at the very beginning of the `xs:schema` clause. The `namespace` attribute specifies the namespace to be imported (this must match the target namespace definition in the imported schema file), and the `schemaLocation` attribute specifies the location of the file to be imported. In addition, we must specify a namespace prefix for the imported namespace. This is done in the `xmlns:i` attribute of the `xs:schema` clause. This prefix is used when we refer to a musical instrument, for example `xs:element ref="i:saxophone"`. Note that there can be several import clauses in one schema, and even several import clauses for a given namespace.

As you can see, we have opted to use the prefix "e:" for the schema's target namespace `http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia`. This is just to preserve the namespace prefix usage in the conceptual model - continuing using this namespace as the default namespace for the schema would also be valid.

Schema Evolution

Once a schema has been defined, it is very unlikely that it will always stay in the same state. Business requirements change and bugs are detected, so the schema must be modified in order to adapt to changing circumstances. In this section we discuss how a schema can be modified safely. “Safe” in this context means that the modified schema must still cover all existing valid document instances of the original schema. The following guidelines ensure that the new schema is at least as “wide” as the original schema:

- Never make cardinality constraints narrower. You may increase `maxOccurs` and decrease `minOccurs`. However, decreasing `maxOccurs` or increasing `minOccurs` might render existing instances invalid. This logic also applies when adding or removing elements or attributes. Any non-existing element can be seen as an element with `minOccurs="0"` and `maxOccurs="0"`. If you want to add a new element, just imagine that it already exists with `minOccurs="0"` and `maxOccurs="0"`. Consequently, leave minOccurs at "0" and increase only maxOccurs to comply with the above rule. This means that all new elements must be optional. On the other hand, if you no longer need a given element, simply set `minOccurs="0"`. This makes the element optional, so both new and old instances are covered. The same logic applies to attributes. New attributes should only be added with `use="optional"`, and for attributes that are no longer needed, `use` should also be set to `optional`.
- You can always introduce new choices into a schema: you can wrap existing element definitions, element references, model group definitions (`xs:sequence`, `xs:all`) or references to global groups in an `xs:choice` clause and add more alternatives. Existing instances remain valid but the new alternatives allow for additional instances.
- Never introduce new fixed or default values or modify existing fixed values. This might render existing instances invalid.
- Never restrict the definition of existing simple type definitions. For example, you can safely change a type definition from `xs:short` to `xs:integer`, but not vice versa. The same applies for extension by list: you can safely replace `xs:NMTOKEN` with `xs:NMTOKENS`, but not vice versa. Do not introduce new facets into a type definition, and do not make the definition of existing facets narrower (e.g. reduce the number of total digits from 7 to 5).

These are general guidelines. You can also modify a schema in a way that is inconsistent with existing documents, providing you subsequently validate all affected documents, but this of course could be very time-consuming.

In Tamino XQuery 4, you can modify documents by using the `xquery update` statement to insert, delete, replace or rename nodes, but the resulting documents must comply with the existing schema; the schema itself cannot be modified by `xquery update`.

Open Content Model

Schema developers cannot always predict the requirements that may arise in the field. XML Schema therefore provides extension mechanisms that allow document authors to include elements and attributes into document instances that are not declared in the schema. These extension mechanisms are implemented in XML Schema as wildcards (`xs:any` and `xs:anyAttribute`).

Let us assume, for example, that we want to make the definition of `tName` more generic, allowing document authors to include a `title` child element. We can allow document authors to insert any number of extra child elements before, between, and after the existing child elements with the following definition:

```
<xs:complexType name="tName">
  <xs:sequence>
    <xs:any namespace="##other" processContents="lax"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="first" type="xs:token"/>
    <xs:element name="middle" type="xs:token" minOccurs="0"/>
    <xs:any namespace="##other" processContents="lax"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="last" type="xs:token"/>
    <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:anyAttribute processContents="lax"/>
</xs:complexType>
```

We have also added an `xs:anyAttribute` clause to allow for additional attributes.

Note the specification of `namespace="##other"` for the first two wildcards. This is to avoid non-determinism. Without such a specification, the wildcard could contain elements from the same namespace. When encountering a `first` or a `last` element in a document instance, the parser would not be able to decide if such an element should be accepted by the wildcard or by the following element specification without looking ahead in the input stream. For the same reason we did not introduce a wildcard in front of the element definition `middle`. `middle` is optional, so a parser would not know where to place an instance element: into the wildcard before or after the element `middle`.

Note that Tamino allows for an alternative (non-standard) open content model that does not suffer from this problem (see *From Schema to Tamino*::*Schema level Definitions*).

Versioning

There are two questions that arise when we create a new version of an existing schema:

The first question is: Should we change the target namespace of the new schema? The answer is simple: If you want to invalidate the schema against existing document instances, and against existing schemas that might include or import this schema, do so. In this case, you should retain the old schema version in order to support existing applications. Usually, this option is taken when the changes in the schema are severe. In all other cases, leave the target namespace unchanged and indicate the new schema version by other means.

This leads us to the second question: How do we indicate a version number within a schema? The good news is that XML Schema features a `version` attribute in the `xs:schema` clause. The bad news is that parsers do not evaluate this attribute, so you won't see the version number when you access a document instance through a DOM API; extra application logic is required to read out the version number. This version number is meant for human consumption, it indicates the version of the schema to the schema author. To convey version information to applications, the best method is to specify a `version` attribute for the root element of a schema. We can give this attribute a fixed value reflecting the current version. This attribute does not show up in document instances, but applications can see it through the DOM API. Of course, nobody stops us from defining such version attributes for other elements than the root element, too, so you could add different version information to different subsections of the same schema.

5 Integrity

- Simple Constraints 64
- Cross Field Constraints 65
- Constraints Across Documents 65
- Data Integrity 67
- Unique Keys 68

In classical relational databases, integrity rules and triggers are used to maintain the integrity of the information stored in the database. Integrity means that the constraints defined in the conceptual model are not violated and that the data structures defined in the conceptual model are kept intact. This is possible by applying integrity rules and triggers within the same transactional context as the operations that modify the stored information.

Especially this last condition – the transactional context – becomes impossible to satisfy when we extend our data model beyond the boundaries of traditional enterprise databases. When a model includes data from sources somewhere on the World Wide Web, it becomes impossible for database systems to guarantee the integrity of data structures that span beyond the boundaries of the transactional environment. For example, a database cannot "lock" foreign web resources during a transaction, and thus cannot stop other users from interfering with that transaction.

On the other hand, web resources may be temporarily unavailable. And, increasingly, our hardware is becoming mobile, either as traveling PDAs, or in the form of wireless LANs. In these cases, it is not always possible to satisfy integrity constraints immediately, and instead of using transactional integrity techniques we need to use *synchronization* techniques to keep the data model consistent in the long term.

In general, the resource manager (i.e. the database) is the wrong instance for the enforcement of data integrity. In many cases this task is better left to the application logic, or to appropriate middleware.

In the following sections we indicate how constraints can be defined for XML documents. The method of choice in Tamino for implementing constraints is triggers. See the description of trigger functions in the chapter *Tamino Server Extension Functions* in the documentation for server extensions for details.

Simple Constraints

Constraints are used to add more meaning to a model. During the definition of the XML schemas we have already added a considerable set of constraints to our model: datatypes. Each datatype such as string, float or integer constrains the value domain of an element or attribute. Additional constraints are enumerations or type parameters (facets) such as `totalDigits`, `maxLength`, `minExclusive`, etc.

Another type of constraint is the cardinality constraint, which can be defined in schemas using `minOccur` and `maxOccur`. For example, by decorating the element

```
<xs:element name = "jazzMusician" type = "xs:string"
            minOccurs = "2" maxOccurs = "unbounded" >
```

in `collaboration`, we set up a constraint that a collaboration must consist of at least two jazz musicians. Actually, an element with no `minOccur`/`maxOccur` decoration at all has the strictest constraints: it requires a cardinality of `1..1`. The weakest cardinality constraint is `minOccurs = "0" maxOccurs = "unbounded"` which leaves all possibilities open.

All these constraints can be checked by a validating parser. This happens, for example, when a document is inserted into or updated in Tamino.

Cross Field Constraints

What interests us in this context are constraints that affect more than one element or attribute. For example, we want to make sure that a jazz musician of type `instrumentalist` plays at least one instrument, whereas other types of jazz musicians (`jazzComposer`, `jazzSinger`) are not required to play an instrument. Here, the standard trigger functions of Tamino can be used to perform the constraint checking.

Constraints Across Documents

The `document()` function in XPath can be used to access multiple documents in a single query. This allows us to formulate constraints that span multiple documents. Let us assume that we have the following `collaboration` and `jazzMusician` documents stored in a Tamino database `http://localhost/tamino/jazz/` in collection *encyclopedia*:

```
<?xml version="1.0"?>
<collaboration type="jamSession">
  <name>post-election jam</name>
    <jazzMusician>
      http://localhost/tamino/jazz/encyclopedia/dizzy.xml
    </jazzMusician>
    <jazzMusician>
      http://localhost/tamino/jazz/encyclopedia/parker.xml
    </jazzMusician>
    <performedAt>
      <location>Blues House</location>
      <time>1965-10-21T20:00:00</time>
    </performedAt>
</collaboration>
```

```
<?xml version="1.0"?>
<jazzMusician ID="ParkerCharlie" type="instrumentalist">
  <name>
    <first>Charlie</first>
    <last>Parker</last>
  </name>
  <birthDate>1920-08-19</birthDate>
</jazzMusician>
```

```
<?xml version="1.0"?>
<jazzMusician ID="GillespieDizzy" type="instrumentalist">
  <name>
    <first>Dizzy</first>
    <last>Gillespie</last>
  </name>
  <birthDate>1917-10-21</birthDate>
</jazzMusician>
```

We want to check that the performance date of the jam session is not earlier than the birth dates of its participants. We can achieve this with the following rule:

```
<rule context = "collaboration[@type='jamSession']/jazzMusician">
  <assert test = ↩
"number(translate(document(.)/*/birthDate,'1234567890-','1234567890')) &lt;
                  ↩
number(translate(substring(../performedAt/time,1,10),'1234567890-','1234567890'))">
    No jam for unborn child <value-of select="document(.)/*/name/last"/>!
  </assert>
</rule>
```

As we can see, the rule is executed in the context `collaboration[@type='jamSession']/jazzMusician`. The filter expression restricts the application of the rule to collaborations of type `jamSession`. The content of the element `jazzMusician` is used as a URL to locate the appropriate document (`document(.)`). From this document we fetch the element `birthDate`.

The `translate()` function removes the dashes from the ISO date string before the string is translated into a number. The same process is performed with the date part of element `performedAt/time` of the current document. Then both dates are compared using the operator `<` (<). This rather clumsy process of translation and conversion into a number is necessary because XPath 1.0 does not support order relations between strings (strings can only be compared for equality) and, of course, XPath 1.0 does not support XML Schema datatypes. XPath 2.0 should improve this situation substantially.

To make the resulting report more informative, we include the name of the offending musician into the error message, too. This is done with the `value-of` clause.

Let us now assume that the collaboration document does not contain pointers (URLs) to the `jazzMusician` documents but instead identifies jazz musicians by their ID. This is what we actually

want because usually URLs do not make good keys: they specify a location but do not identify a document.

```
<jazzMusician>GillespieDizzy</jazzMusician>
<jazzMusician>ParkerCharlie</jazzMusician>
```

We assume, too, that the documents are stored in Tamino. In this case we must replace all

```
document(.)/*
```

expressions with

```
document(concat('http://localhost/tamino/jazz/encyclopedia?_XQL=jazzMusician[@ID=&quot;',
               .,'&quot;]'))//jazzMusician
```

i.e., we construct an HTTP query to Tamino, such as:

```
http://localhost/tamino/jazz/encyclopedia?_XQL=jazzMusician[@ID="ParkerCharlie"]
```

and then extract the root node (`jazzMusician`) of the result document returned.

Data Integrity

Documents should only be written into the database after we have made sure that they do not violate the constraints imposed on them, i.e. that they comply with the application's business rules.

When a document is stored, Tamino checks the structural constraints and the datatype constraints defined in the document schema. This can be influenced by the content model definition for the document type. If the content model is set to "closed", Tamino only allows nodes that are defined in the document schema. Otherwise, Tamino allows additional nodes within a document instance.

Apart from that, as outlined above, other constraints may exist that cannot be appropriately described with XML Schema. Examples are cross-field constraints and cross-document constraints.

It is the application's responsibility to check for such constraints. In particular, the validation of cross-document constraints requires extra consideration for the transaction logic. To make the validation bulletproof, the validation and the following update must be performed in a single transaction with the isolation level set to "_shared" or "_protected". When doing so, we must apply the same guidelines for accessing multiple documents in one transaction as we outlined above in order to avoid deadlocks.

Unique Keys

Tamino's unique document key mechanism prevents users from storing (in a specific doctype) multiple documents with the same key. A key may be composed of one or more values of elements or attributes contained in the document. The unique document key mechanism monitors incoming documents according to specified constraints and prohibits the storage of these documents in a single document container (doctype) if a duplicate document key is identified. This is especially useful for the administration of user IDs and other IDs that have to be unique. Uniqueness can be set in the XML Schema for the document type.

6 Operations

Apart from integrity rules which check for the violation of constraints, a second concept exists to ensure referential integrity: triggers. Referential triggers are used to invoke operations when a data record is inserted, updated, or deleted. For example, when a record representing a purchase order is deleted, it is also necessary to delete the records containing the individual order lines. This can be achieved with triggers.

For XML documents, however, the concept of triggers to guarantee referential integrity is not as essential as it is for SQL. The reason is that a complex business document such as a purchase order is not – like in the relational world – fragmented into flat records but is instead stored as a single structured document. When such a purchase order is deleted, the order lines vanish, too, because they are contained in the same document.

However, there are scenarios where we might want to execute additional operations when documents are inserted, updated, or deleted:

- As we have mentioned before, it is sometimes necessary to segment a large document into several smaller documents. In this case we must make sure that all sub-parts are deleted when the main document is deleted – the classical case of a relational trigger.
- For performance reasons our model may contain redundant data. As discussed before, `jazzMusician` documents may contain an element `numberOfAlbums` with the number of all albums to which the respective musician has contributed. If we add a new `album` document to the database, we must update all correlated `jazzMusician` documents (and similarly when we delete an album).
- Usually we convert XML documents into HTML using an XSLT stylesheet when we want to display them on the Web. However, this can be a bottleneck if we do this on the fly (i.e. every time when a document is requested). A common technique is to pre-generate HTML pages from the XML documents and satisfy incoming requests directly from the HTML pages. When an XML document changes, we have to re-generate the corresponding HTML page; otherwise, users would get outdated content. This re-generation could be initiated via a trigger.

The operations that are possible in the context of insert, update, or delete operations by far exceed the scope of traditional referential triggers. In particular, they can modify data outside the database, data that resides somewhere else.

Similar to integrity rules, the best place for the implementation of such general operations is the application or some appropriate middleware.

7 From UML to XML

- XML Support in UML 72
- From Conceptual Model to UML 73

The UML (Unified Modeling Language) is a popular object-oriented modeling method. Since it has been submitted as an ISO standard, we discuss it here also in the context of modeling for XML.

XML Support in UML

Most commercial CASE tools that support UML such as Rational Rose or TogetherSoft also support the importing and exporting of XML DTDs and/or XML Schema. In the simplest case, an existing DTD or XML Schema is simply imported into the CASE tool, resulting in a number of UML classes that represent the different nodes of the XML document. Side effects of this functionality are the possibility of converting from DTD to XML Schema and vice versa, and of generating a Java-based access layer for a given document type.

However, you should not misinterpret this technique as "conceptual" modeling: it results in a model of an implementation object. Generating XML schemas from a conceptual model is somewhat more demanding. In this chapter, we discuss how this can be achieved with relatively simple means.

What we should not expect in this context, however, is a complete solution that supports round-trip engineering. UML was developed with object-oriented implementation and design methods in mind. We should therefore experience (and tolerate) a slight impedance mismatch between UML and XML.

One way to generate code with a CASE tool is to write production rules for the tool's code generator. However, this is a proprietary approach, and we would have to demonstrate different solutions for each CASE tool on the market.

We therefore choose a method that can be applied to most CASE tools. Practically all CASE tools on the market support the exporting of metadata to XMI *(XML Metadata Interchange)*. XMI is an XML-based standard for the exchange of modeling data between different design and development tools. It can capture virtually all information within a UML model.

In the context of this tutorial we use Poseidon for UML (the Community Edition is freeware, available from ***http://www.gentleware.com/***), a commercialized version of ArgoUML, as our CASE tool. We define our jazz example in UML, then export it to XMI, and finally convert the resulting XMI into XML Schema with the help of an XSLT stylesheet.

From Conceptual Model to UML

Here are the mapping rules to cast an asset-oriented model onto UML:

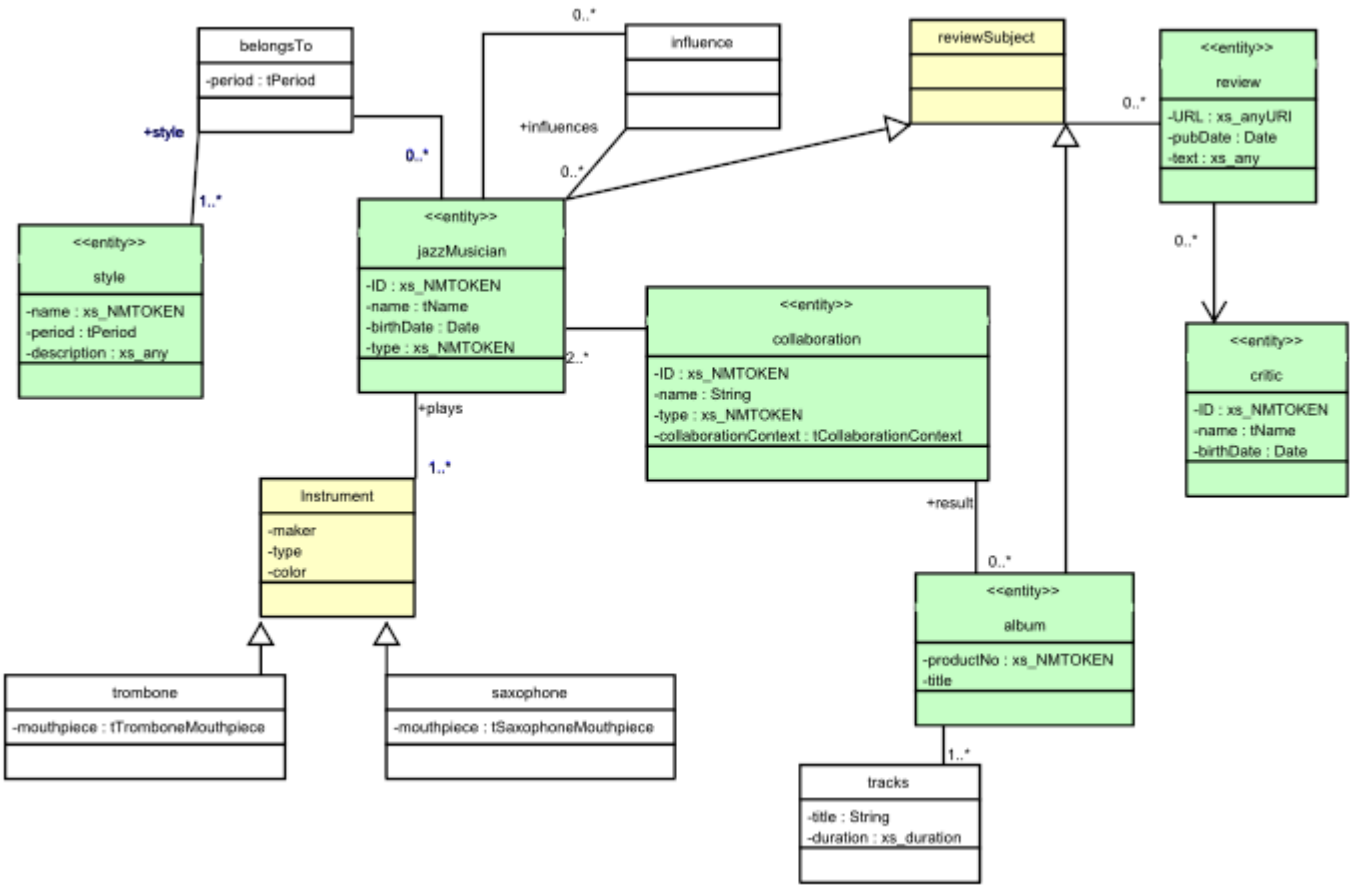
1. We decorate all identifying assets of business objects with the stereotype `entity`. This allows us to generate arcs leading to these assets differently (as these arcs lead to separate documents).
2. We use qualified names for all assets (i.e. names with namespace prefixes). Because the colon is not a valid name character in most programming languages, we replace it by an underscore.
3. Since UML is an object-oriented technology, it does not have a native concept of primary keys. It is conventional to decorate primary keys with the stereotype `primaryKey`.
4. We represent the arcs of our conceptual model as unnamed UML associations. If required, we can decorate the source end of an association with a role name and the target end with a cardinality constraint.

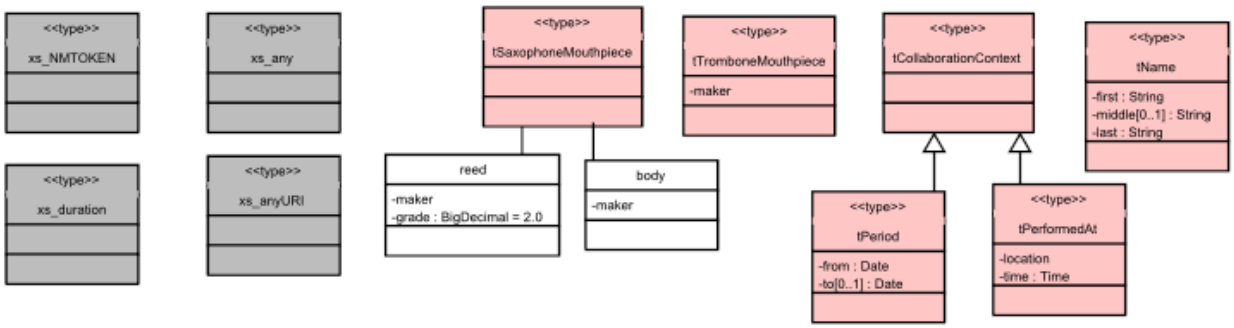
   The exception to the rule are the arcs that are decorated with an `is_a` label. These are represented as a UML generalization/specialization. Multiple inheritance is allowed in UML. Thus, the conversion process must resolve inheritance relations because XML Schema does *not* support multiple inheritance.
5. UML attribute specifications can include a type and an initial value. Other XML Schema-specific constraints, such as `minOccurs`, `maxOccurs`, `form`, `maxLength`, `length`, `totalDigits`, `fractionDigits` and `enumeration`, have no specific equivalent in UML but can be specified as tagged values (which we name appropriately `xs_minOccurs`, `xs_maxOccurs`, etc.). Similarly, a tagged value `xs_fixed=true` can be used to determine if the initial value shall be regarded as a fixed value or as a default value.
6. We can use Java-based datatypes for attributes. These are already built into the modeler and can be mapped automatically onto XML Schema datatypes during the conversion process. We can also explicitly use the built-in datatypes of XML Schema, but we have to declare them explicitly in UML. We do this by defining classes such as `xs_NMTOKEN` or `xs_ID` and decorating them with stereotype `type`. We also introduce a pseudo datatype `xs_any` to indicate wildcards.
7. UML does not support complex attribute definitions. Instead, we have to resolve complex properties. We have two options: (1) represent a complex property as an explicit aggregation, or (2) define a separate datatype for a complex property. In this example, we opt for the latter. For example, we introduce the datatypes `tPerformedAt` for `performedAt(location&time)`, `tPeriod` for `period(from,to)` and `tName` for `name(first,middle?,last)`.
8. Alternatives (choice groups) require extra care. In UML we model them as a datatype generalization. For example, the property `(performedAt(location&time)|period(from,to))` in asset `collaboration` is modeled as an element (which we call `collaborationContext`) with a type that is a generalization of the datatypes `tPerformedAt` and `tPeriod`.
9. Clusters are represented as a generalizations also. To represent, for example, the cluster containing all the instruments, we introduce a generalized class `instrument`. Because we do not want

this class to appear in the final schema, we define it as an *abstract* class. Similarly, we introduce a generalized class representing all classes that are subject to reviews, such as jazz musicians and albums.

10. By default, we assume an ordered sequence for the attributes of an UML class and would therefore generate an `xs:sequence` connector. If we want an unordered sequence (resulting in an `xs:all` connector), we indicate this by attaching the tagged value `xs_ordered=false` to the respective UML class.
11. Similarly, we attach the tagged value `xs_mixed=true` if a class shall contain mixed content.

Applying these rules, we finally arrive at the following model:





Most UML tools provide a function to serialize a model into XMI format. XMI is an XML-based industry standard for the exchange of metadata between CASE tools. Because it is XML based, XMI can be converted with the help of XSLT stylesheets into other formats such as XML Schema. An example of such a stylesheet can be found at ***http://www.aomodeling.org/***.

8 Schema-Related Web Sites

The following compilation lists a few web sites that are relevant to schema definition in XML.

- ***http://www.w3.org/***

  The World Wide Web Consortium (W3C) is the reference point for the XML specification (i.e. DTDs) and XML Schema.

- ***http://www.xml.org/*** (hosted by OASIS)

  is a repository for XML schemas.

- ***http://www.xfront.com/***

  discusses programming techniques for schema authors.

- ***http://www.xmlpatterns.com/***

  contains a collection of design patterns for XML.

- ***http://www.aomodeling.org/***

  is the home of Asset Oriented Modeling.

II From Schema to Tamino

In this chapter we concentrate on the physical aspects of XML data storage. We describe how these aspects can be described in a schema, and discuss performance issues. We describe how virtual documents can be created by means of document composition when querying Tamino, and which concepts for transactional processing are supported.

This information is organized under the following headings:

Tamino Annotations in XML Schema

Namespace Support

Indexing

Document Composition

Efficient Querying

Performance Issues

9 Tamino Annotations in XML Schema

- Annotation and Appinfo 82
- Schema-Level Definitions 83
- Node-Level Definitions 84

While XML Schema covers the *logical* aspects of document type definitions, it does not prescribe how to define the *physical* aspects. Many XML processors require extra information on how to process the instances of a given document class. Tamino, for example, needs to know the collection to which a document class belongs, the elements or attributes which are used for indexing, the type of indexing that is used, how document elements may be mapped onto fields in external databases, and so on.

Annotation and Appinfo

For these and other purposes XML Schema provides an extension mechanism. Any schema, element or attribute definition in XML Schema can be equipped with one or more *annotations*. Each annotation can consist of two child elements: `documentation` and `appinfo`. The `documentation` element contains documentation for human readers, and the `appinfo` element contains information for machines.

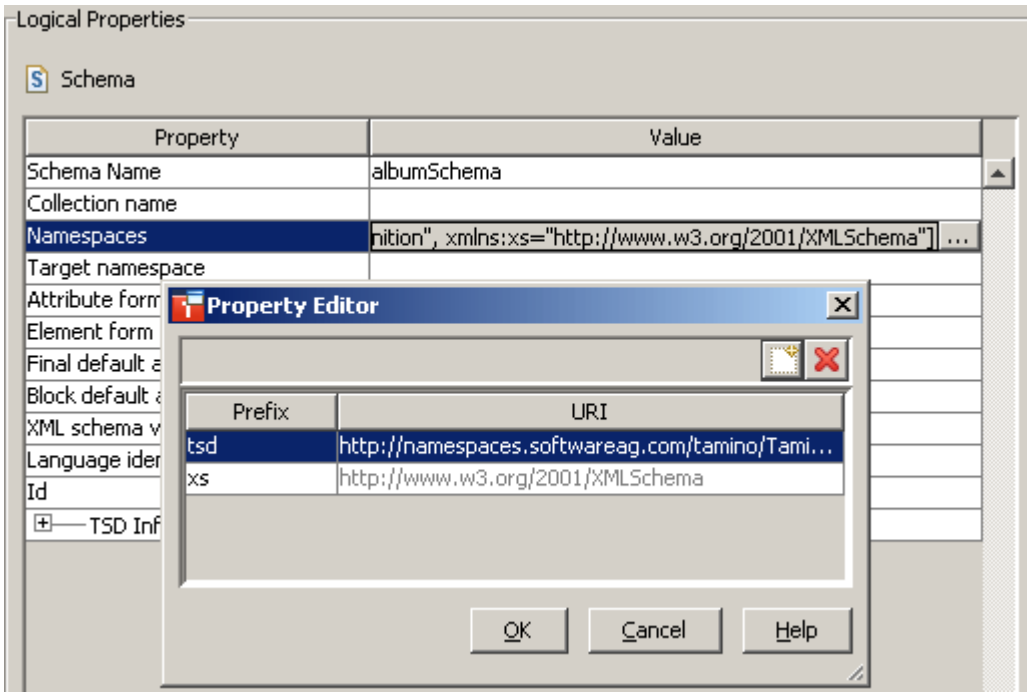
Tamino uses the `appinfo` element to store Tamino-related information within a schema file:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema
        xmlns = ↩
"http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
        xmlns:xs = "http://www.w3.org/2001/XMLSchema"
       xmlns:tsd = "http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition"
        targetNamespace = ↩
"http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
        elementFormDefault = "qualified"
>
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "albumSchema">
        <tsd:collection name = "encyclopedia"/>
        <tsd:doctype name = "album">
          <tsd:logical>
            <tsd:content>open</tsd:content>
            <tsd:accessOptions>
              <tsd:read/>
              <tsd:insert/>
              <tsd:delete/>
              <tsd:update/>
            </tsd:accessOptions>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:schema>
```

These annotations are generated by the Tamino Schema Editor.

Schema-Level Definitions

The example above shows an annotation on the schema level. This is indicated by the `<tsd:schemaInfo>` element. The attribute `name` specifies the name of the schema (`albumSchema`).



Enclosed in the `<tsd:schemaInfo>` element is a definition for the Tamino collection to which the schema belongs. Typically a collection contains document types that are somehow related, for example the document types derived from a specific conceptual model or all schemas that belong to a given application.

> Here, we define a collection `encyclopedia` that will eventually contain all the document types (`jazzMusician`, `style`, `collaboration`, `album`, `review` and `critic`) that we defined earlier (*From Conceptual Model to Schema*::*From Model to Schema*).

In addition to the `<tsd:collection>` element, the element `<tsd:schemaInfo>` contains a `<tsd:doctype>` element which defines the name of the document type as `album`. This name is not necessarily the same as the schema name, but must match the root element name of each document instance belonging to that particular document type.

The `<tsd:doctype>` element may contain an element `<tsd:logical>`. This element specifies the access options for the document instances and the content type:

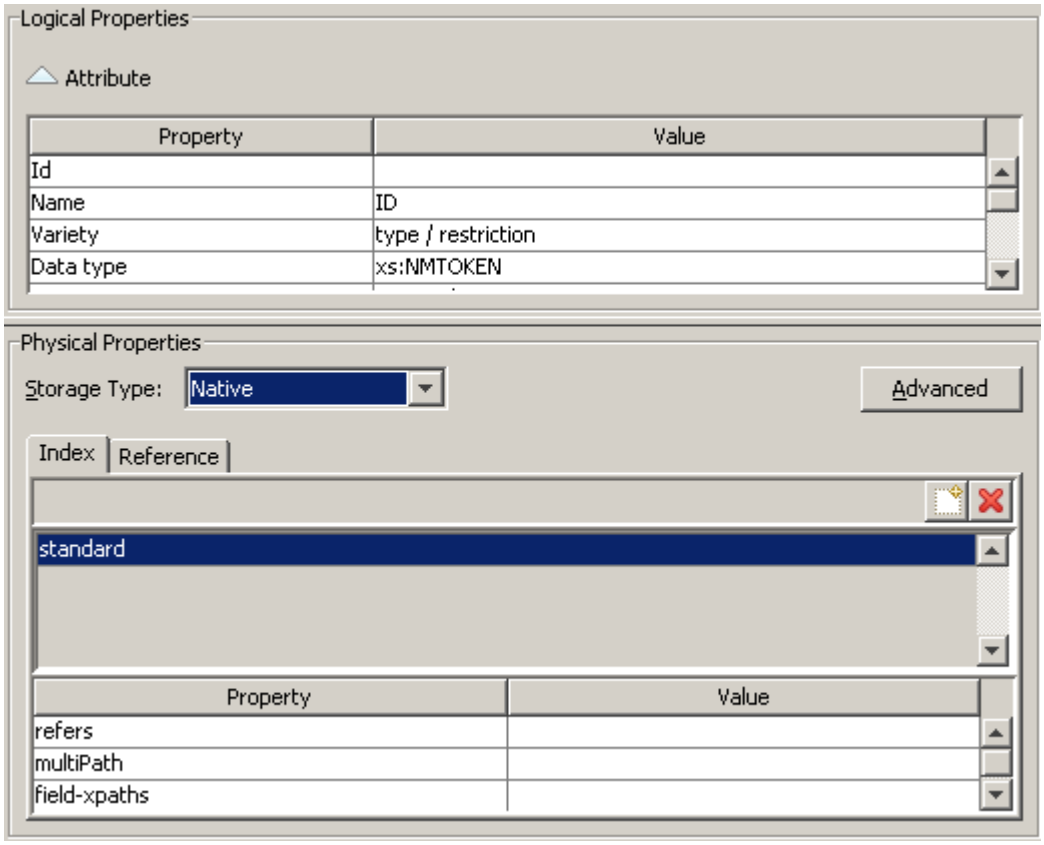
- The access options specify the database operations (read, insert, update, delete) that are allowed for this document type;
- The content type "closed" specifies that a document instance may not contain deeper structures than specified in the schema. In contrast, content type "open" allows document instances to be extended with tags that are not defined in the schema.

This is a more flexible - albeit non-standard - approach to the open content model than using `xs:any` and `xs:anyAttribute`. For a discussion of the pros and cons of the Tamino-specific open content model, see also *Tamino-specific Extensions to Logical Schema*::*Open Content vs. Closed Content Validation*.

> We could, for example, define our `style` document type with content type "open" because its `description` element may contain other markup. However, using an `xs:any` child node for the `description` element definition is probably the better choice here: thus we stay with the W3C XML Schema recommendation, and we are able to specify a namespace for the XHTML content of the `description` element.

Node-Level Definitions

The physical properties of each document node can be specified with similar annotations. Again, these annotations are generated by the Tamino Schema Editor. Here is an example for the attribute `ID` of our schema `jazzMusician`:



and the resulting code:

```
<xs:attribute name = "ID" type = "xs:NMTOKEN" use = "required" form = "unqualified">
  <xs:annotation>
    <xs:appinfo>
      <tsd:attributeInfo>
        <tsd:physical>
          <tsd:native>
            <tsd:index>
              <tsd:standard></tsd:standard>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
      </tsd:attributeInfo>
    </xs:appinfo>
  </xs:annotation>
```

Contained in the `<xs:appinfo>` element is the element `<tsd:attributeInfo>`, which describes the properties of this attribute node. In the Tamino Schema Editor we have specified that the storage type of the attribute is "Native". This means that the attribute is to be stored as native XML in Tamino. There are also other options, for example, to map a document node to a Tamino Server Extension (see *Utilizing Server Extensions*::*Derived elements*) or to fields in a foreign database such as Adabas or a third-party RDBMS.

10 Namespace Support

- Qualified Queries 88

Tamino supports XML Namespaces as defined in the W3C Recommendation *Namespaces in XML* (***http://www.w3.org/TR/REC-xml-names/***). Tamino schemas may define a target namespace, may import sub-schemas from other namespaces, and may contain wildcards with content from foreign namespaces. In section *Introduction to XML Schema*::*Namespaces and wildcards* we discuss how namespaces are used with XML Schema; the support in Tamino does not differ from this.

Qualified Queries

The question, however, is: How do we qualify element and attribute names within queries? How do we set up namespace bindings in a query (or in an update or insert operation)? The answer depends on the query language used:

- X-Query (XQL) does not allow bindings to be defined between namespace prefixes and namespace identifiers. If we need to qualify names within a query, we must use the bindings that were set up in the schema of the document type. We just prefix names with the same prefixes that were defined in the schema.

  Specific aspects of namespace handling with X-Query are discussed in the section *_XQL, _DELETE* of the Tamino document *Namespace Handling for Specific X-Machine Requests*, for example how Tamino treats X-Query queries on schemas that have a default namespace but no prefix for it.

  Given the schema in section *From Model to Schema*::*Multi-Namespace Schema Composition*, a query for jazz musicians who play the saxophone would look like this:

```
e:jazzMusician[e:plays/i:saxophone]
```

- Tamino XQuery 4 allows (and requires) namespace bindings to be defined within the queries, and thus allows prefixes to be used that differ from those defined in the schema. We define these bindings in the query prologue using the expressions `default element namespace={namespace-identifier}` or `declare namespace {namespace-prefix}={namespace-identifier}`. In XQuery 4, the above query for jazz musicians who play the saxophone would look like this:

```
declare namespace ↩
ency="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
declare namespace ↩
instrument="http://www.softwareag.com/tamino/doc/examples/models/instruments"

let $j := input()/ency:jazzMusician
   where ($j/ency:plays/instrument:saxophone)
     return $j
```

11 Indexing

- Declaring an Index 90
- Candidates for Indexes 92
- Composite Keys 93
- Object Identity 93
- Text Retrieval 95

Indexing is one of the principal features that distinguish a database from a simple file system. In a file system, searching for a specific element value usually requires scanning through the whole document set and parsing all the documents. A database system, in contrast, can organize much more efficient access paths by means of indexes.

Tamino can index attributes and simple elements as well as complex elements, and it allows two different categories of index: standard index and text index. Within these categories, various mechanisms can be used, which give rise to simple indexes, multi-path indexes, compound indexes and reference indexes.

Declaring an Index

Tamino's annotations (see *Tamino Annotations in XMLSchema*::*Node Level Definitions*) also indicate whether or not document nodes should be indexed, and which type of indexing is to be used:

Standard Indexing
Standard indexes are the classical database indexes. The complete node content is used as the index value. When a document is stored or updated, Tamino inserts standard indexes into a specific index, enabling efficient queries. Tamino can index all XML Schema datatypes.

Text Indexing
Here, Tamino analyzes the content of an indexed node word by word and stores the individual words in an index. We discuss this technique in more detail in **Text Retrieval**.

Standard and Text Indexing
This may be required in special cases but should, in general, be avoided because of the high overhead.

Indexes can be declared for attributes and for leaf elements, but also for elements that contain child elements. In the latter case the index is constructed from the concatenated text of the element and its child elements, i.e. from the result of the `text()` function applied to that element. For example, if we had:

```
<name><first>Louis</first><last>Armstrong</last></name>
```

and declared an index for `<name>`, the string "LouisArmstrong" would be used as index value.

The way in which the markup is treated in a given database can be controlled by setting the database property `markup as delimiter`. This can be done using the Tamino Manager. If the property is set to "yes", markup is treated as a delimiter for text (i.e. it is handled like white space). If it is set to "mixed", markup is not treated as a delimiter in mixed-content elements. If the property is set to the default value "no", markup is not treated as a delimiter.

Indexes are not necessarily required to perform queries. Tamino can interpret any valid query expression, even if an index is not defined. However, this can be costly. For example, the query

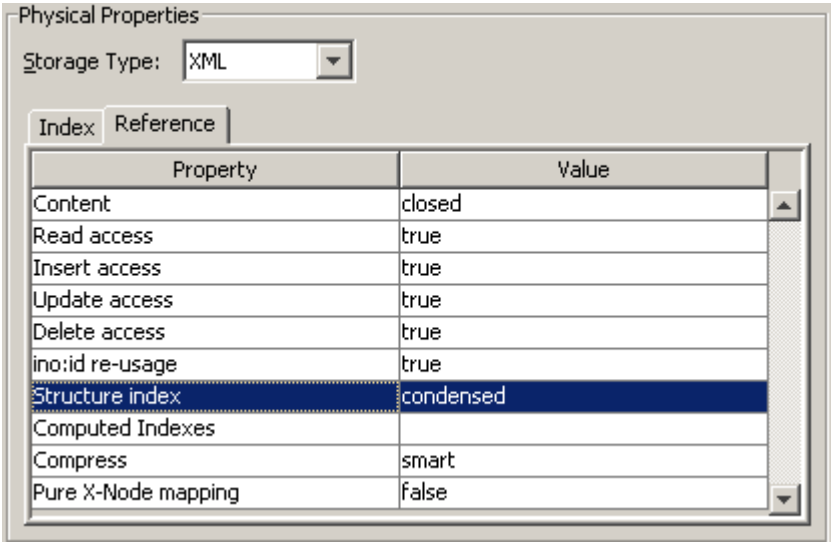
```
jazzMusician[@ID="ColtraneJohn"]
```

would require Tamino to read all `jazzMusician` instances, parse them to extract the attribute `ID`, and compare the `ID` attribute value of each document with the search string. This may be practicable if we have only a small document base with half a dozen documents, but is out of the question if our database contains thousands of `jazzMusician` instances.

In contrast, if the attribute `ID` is declared as an index, Tamino would only read and parse those `jazzMusician` documents that have "ColtraneJohn" as the value of the attribute `ID`.

So far, we have only discussed indexes for document nodes that are defined in the schema. But what about markup that is not defined in the schema but only appears in individual document instances, for example in a wildcard? Can Tamino apply indexing to these document *extensions* as well?

The answer is yes. The optional element `tsd:physical/tsd:structureIndex` in the `tsd:doctype` declaration allows Tamino to add nodes found in document instances to its internal repository, and thus to execute queries using such nodes much faster.



For example, the document instance:

```
<?xml version="1.0"?>
<jazzMusician
    xmlns="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
    type="instrumentalist"
    ID="GillespieDizzy">
  <name>
    <first>Dizzy</first>
    <last>Gillespie</last>
  </name>
  <birthDate>1917-10-21</birthDate>
```

```
  <deathDate>1993-01-06</deathDate>
</jazzMusician>
```

contains an element that was not declared in the schema: `deathDate`. Without the `structureIndex` property, a query on `deathDate` would take a long time, but setting `structureIndex` to "FULL" or "CONDENSED" speeds up the query considerably. The difference between "FULL" and "CONDENSED" is that "CONDENSED" only remembers *that* an element `deathDate` is used in at least one instance of this document type. In contrast, "FULL" remembers in *which* documents such an element exists. This requires more memory but may make the query even faster when only a few documents contain this element.

The query `jazzMusician[deathDate="1959-07-17"]`, for example, would have the following effect:

`structureIndex="NO"`
: A full scan of all `jazzMusician` instances is required.

`structureIndex="CONDENSED"`
: If no `jazzMusician` instance has a node `jazzMusician/deathDate` then the query can be answered quickly (no object returned). If at least one instance has a node `jazzMusician/deathDate` then it is still necessary to scan all `jazzMusician` instances in order to answer the query.

`structureIndex="FULL"`
: Since many jazz musicians are still alive, the query can be answered faster because Tamino knows which `jazzMusician` instances have a node `jazzMusician/deathDate`. Only those instances are scanned.

`structureIndex` should be set to "NO" when we expect elements that contain random markup (e.g. XHTML). This would be the case for the `description` element in our `style` document type.

Note that after an initial set of indexes has been defined, it is possible to add or remove indexes subsequently by using update schema operations, based on the X-Machine command `_define`.

Candidates for Indexes

Generally, we should declare the primary keys of all business objects as standard indexes. In our case, these are the attributes `@ID` for the business objects `jazzMusician`, `critic`, and `collaboration`, `@name` for business object `style`, and `@albumNo` for business object `album`. The only business object that does not require an explicit index is `review`, because we access `review` objects via URL.

In addition, foreign keys are also prime candidates for indexes. For example, `jazzMusician/influence/influences/@ID` can be a useful index when we want to find out by whom a given jazz musician was influenced. For example:

```
jazzMusician[influence/influences/@ID="ParkerCharlie"]
```

would certainly result in an impressive list of `jazzMusician` documents, including, for example, those of John Coltrane and Miles Davis. Similarly, we could index `jazzMusician/belongsTo/style/@name`, too, in order to be able to quickly find all musicians belonging to a particular style.

The general rule is to index all those nodes which are frequently used as access paths for queries, and not to index nodes that are only rarely used in queries. Indexing speeds up queries but it also slows down insert, update and delete operations and consumes disk space. The conceptual model gives us a first hint for the possible access paths. However, later fine tuning (see also **Efficient Querying**) requires us to study the actual usage pattern of the database before making a final decision about which nodes to index.

Composite Keys

Finally, we have to consider the case when a primary key is a composite, i.e. it consists of several nodes. For example, this would be the case if we use the node `name` as a primary key for business object `jazzMusician`. This node comprises the child elements `first`, `middle`, `last`. We can choose to create indexes for all of them or only for one or two. We could also use a Tamino compound index to integrate all three elements in a single index. In our case, it might be sufficient to create an index only for the element `last`, because this element can narrow down a search sufficiently. Querying for Pat Metheny with an XPath expression would then look like this:

```
jazzMusician[name/last="Metheny"
             and name/first="Pat"
             and not(name/middle)]
```

(Note that the expression `name/middle=""` would not work because it would require that the element `middle` exists.)

Object Identity

Keys derived from document elements or attributes are only one of several methods to locate a document:

- *URLs* can be used to access any resource on the Internet. However, we should note that a URL does *not* establish an object's identity. A URL specifies a location and not an object. When a document is moved to another location, its URL changes.

  When storing documents and other data in Tamino, Tamino allows a document name to be assigned to such an object. This name is stored in the internal attribute `ino:docname`. This method of object identification is mostly used for non-XML documents stored in Tamino, but it can, of

course, also be used for XML documents. For example, if we store a JPEG image in database `jazz` in collection `encyclopedia` under the name *dizzy.jpg*, we can retrieve it with the following URL: "http://localhost/tamino/jazz/encyclopedia/dizzy.jpg".

We are going to use this addressing mechanism for `review` documents. Remember that we specified the review business object with a key `URL` of type `xs:anyURI`. Instead of implementing an explicit child element `URL`, we use the `ino:docname` feature. This allows us to access `review` documents directly via URL instead of using an XPath or XQuery 4 expression.

- *Generated Identifiers* like the attribute `ino:id` can be used to identify uniquely an object within a certain context. `ino:id`, for example, is generated by Tamino when a new document is stored. It is unique for the specific document type. It is, for example, used when an existing document is replaced: in this case the `ino:id` of the existing document is specified in the new document instance. This causes Tamino not to store the document under a new `ino:id` but instead to overwrite the document with the specified `ino:id`.

  In general, the `ino:id` should only be used for programmatic access to a document (e.g. in a Java servlet) but not as a “permanent” reference to other documents; that is, the `ino:id` of a document should not be stored in other documents. Since the `ino:id` is an internally generated identifier, it may, for example, change when a document subset is moved to another database. This could render the references invalid. The same is true for internally generated identifiers of other database systems.

  Tamino allows documents to be located via URLs consisting of database location, collection name, document type name and `@ino:id`: "http://localhost/tamino/jazz/encyclopedia/jazzMusician/@33"

- *Primary Keys* are derived from object properties, i.e. from the values of document elements and attributes. In the simplest case they are derived from a single value, either a simple content type element or an attribute, provided this value is unique.

  Composite keys are unique values that are composed from several non-unique elements and attributes. We recommend avoiding composite keys derived from complex elements: as pointed out earlier (*From Conceptual Model to Schema*::*Normalization*), such keys are not robust against transformations. As shown above, they are also awkward to query. As we show later (**Efficient Queries**), composite keys are also detrimental to performance when they contain optional elements.

  Keys are unique only in a given context, for example, in the context of a certain document type in a given Tamino database (see section **Schema-Level Definitions**). Keys do not establish a *global* object identity.

  Tamino can locate documents by key via a URL consisting of database location, collection name, and query string: "http://localhost/tamino/jazz/encyclopedia? _XQL=jazzMusician[@ID="ColemanOrnette"]"

- *Globally Unique Identifiers* are not derived from the element and attribute values of a document but are generated with a suitable algorithm. They can establish a unique object identity over

the boundaries of a given database or server. Typical examples for such identifiers are the UUID (Universal Unique Identifier) and URI based identifiers.

UUIDs are used by several XML-based standards as global identifiers.

Each UUID is 128 bits long and consists usually of a 60-bit time stamp and a 48-bit network address (IEEE 802). Good generators can cater for cases when the hardware clock does not have the required resolution (by counting UUIDs with the same clock value). If a network address is not available, a random number is used instead. In this case it cannot be guaranteed that UUIDs are unique, although a conflict is very unlikely.

The advantage of UUIDs is that they are easy to generate and are quite short. The disadvantage is that they are not friendly to the human eye: `<jazzMusician ID="0076B468-EB27-42E5-AC09-9955CFF462A3">`.

Other XML-based standards such as XML Namespaces or XTM (XML Topic Maps) use *domain name based identifiers*. These identifiers consist of a registered domain name, an additional path expression that identifies the document type within the domain, and the primary key of that document type. Although they look syntactically very much like URLs, it is important to realize that their path specification does not define the actual *location* of an object but a virtual position in a semantic space. Therefore, these identifiers must also be stored as a key in the identified object:

```
<jazzMusician ID="http://www.jazzServer.org/people/jazzMusician/EllingtonDuke">.
```

The advantage of these identifiers is that they usually give the human reader some context information about the object. The disadvantage is that they can be quite long, and that the generation of these identifiers can require some bookkeeping.

Text Retrieval

Text retrieval is one of the facilities where Tamino's query languages X-Query and XQuery extend the functionality of the W3C standards XPath and XQuery. Text retrieval means that the content of a node is broken up into words or even word fragments. Each of these words or word fragments is treated as an individual key value and included in the index.

In an X-Query expression we can use the operator ~= (contains) to search for a word contained in a specific node.

A typical example for the use of text retrieval is the `description` element in the document type `style`. The query:

```
style[description~="question"]
```

finds all jazz styles that contain the word "question" in the element `description`. This query is always possible, regardless of whether or not we index `description`. However, if we specify `description` as a text index, the query is processed much more efficiently. Here is how a definition of the `description` element in the schema might look:

```
<xs:element name = "description">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:physical>
          <tsd:native>
            <tsd:index>
              <tsd:text/>
            </tsd:index>
          </tsd:native>
        </tsd:physical>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:any processContents="skip"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

A corresponding document instance may contain the following element:

```
<description>
  <XHTML>
    To be-bop or not to be-bop, there is no question
  </XHTML>
</description>
```

The element was defined as an *any* element. The property `processContents="skip"` allows the element to contain any markup which is not checked against the schema definition (provided the document content model was defined as "open"). Here we have stored a simple XHTML text. The index property was defined as "text". Consequently, any word contained in the description would be added to the index and queries for words contained in the description can be answered efficiently with the help of the index.

The `contains` operator (~=) can be used in conjunction with the proximity operators `adj` and `near`. These operators can be used to search for adjacent words: `adj` requires the words in the specified sequence; `near`, in contrast, does not care about the specified sequence. For example, both:

```
style[description~="not" adj "to"]
```

```
style[description~="not" near "to"]
```

are successful;

```
style[description~= "to" near "not"]
```

is successful, whereas:

```
style[description~= "to" adj "not"]
```

fails.

In XQuery 4 we have similar possibilities. The functions `tf:containsText`, `tf:containsAdjacentText` and `tf:containsNearText` support text retrieval operations.

The `contains` operator is case insensitive, and it can also use wildcards to search for word fragments. For example, the search strings "*-bop", "BE-*", "B*p", "*e-bo*" would find all descriptions that contain "be-bop". However, the last search string "*e-bo*" only uses the index if the database parameter `Word Fragment Index` is set to "yes" (Tamino Manager). This option is set to "no" by default, because using the word fragment index causes substantial overhead (all possible word fragments must be extracted from words and stored as index values!).

Even without using the word fragment index, text indexes require more overhead than standard indexes. We should therefore use the text index facility only for those nodes that we really plan to access with the X-Query `contains` (~=) operator or the XQuery text retrieval operators.

Words that are very likely to be used as indexes should be defined in *load lists*. This can be done by adding a load list document to our database into collection `ino:vocabulary`. For example:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ino:loadlist ino:loadlistname="myloadlist"
              xmlns:ino="http://namespaces.softwareag.com/tamino/response2">
  <ino:word>jazz</ino:word>
  <ino:word>blues</ino:word>
  <ino:word>swing</ino:word>
  <ino:word>ragtime</ino:word>
</ino:loadlist>
```

The required schema is already defined in Tamino. It is possible to define several load lists (with different names). When a database is started, Tamino concatenates all load lists stored in the database and pre-loads the words contained in them for the indexing to speed up the loading of documents.

12 Document Composition

- Dynamic Joins with Tamino XQuery 4 100

The composition of complex data objects from simpler database objects has a long tradition in relational technology. In particular, the *join* operation is heavily used there because relational technology decomposes complex information structures into “flat” two-dimensional tables consisting of atomic values. To reconstruct the complex information structures from those tables, it is necessary to “join” several tables during a query. In addition, by providing a join operation when querying data, relational databases allow users to re-arrange and combine data freely in ways that were not foreseen when the data model was designed.

With a native XML database, composition is used much more sparingly, because the database can store complex information items in their native form, so it is not necessary to “re-compose” these information items from flat tables. However, there are still cases in which we may want to combine several documents (or several document parts) into a single document, or in which we want to rely on other documents to retrieve a certain document.

Take, for example, our jazz encyclopedia. Maybe we want to find all collaborations in which a given jazz musician participates. Because we do not know the jazz musician's ID, we want to use his or her first and last name as search criteria instead. This requires us to locate a matching `jazzMusician` document first, extract the ID from that document, and then find a collaboration document that matches the ID in the attribute `jazzMusician/@ID` – a typical situation for a join.

Mathematically, a join in its most general form is the Cartesian product (cross product) of two document types, followed by some constraint to select only a part of the result set. However, this is only the mathematical theory because it is very inefficient: the Cartesian product of 1,000 jazz musicians with 3,000 collaborations would result in at least $3*10^9$ combinations. (Remember that each collaboration points to at least two jazz musicians, so we get 3000*1000*1000 combinations!) Therefore, database implementations differ vastly from this approach.

Tamino supports document composition using the full dynamic join functionality provided with the XQuery 4 query language.

Dynamic Joins with Tamino XQuery 4

XQuery 4 is a very powerful query language subsuming the functionality of both XSLT and XPath, although with a different, SQL-like syntax. XQuery 4 is based on the W3C XQuery recommendation. Language features such as FLWR-expressions (for, let, where, return) and variables allow for the simplest and the most complex join operations. In addition, XQuery supports namespaces and the full XML Schema type system.

The following example demonstrates how we can compose a joint document from `collaboration` instances, `jazzMusician` instances, and `album` instances:

```
default element namespace = ↩
"http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"

for $c in input()/collaboration
   return
     <collaboration type={$c/@type} ID={$c/@ID}>
       { $c/name }
       { $c/performedAt }
       { $c/period }
       { for $id in $c/jazzMusician/@ID
           let $j := input()/jazzMusician[@ID = $id]
           return
             <jazzMusician ID={$j/@ID}>
               {$j/name}
               {$j/birthDate}
               {$j/plays}
             </jazzMusician>
       }
    { let $a := input()/album[@albumNo = $c/result/@albumNo]
        return
          <album albumNo={$a/@albumNo}>
            {$a/title}
            {$a/track}
          </album>
    }
    </collaboration>
```

Here, we use a `for` instruction to run through all occurrences of the node `collaboration/jazzMusician/@ID`. We then use the value of this node to select `jazzMusician` document instances from collection `encyclopedia`. The actual join expression is contained in the filter expression `[@ID = $id]`. XQuery 4 allows XPath-style expressions within XQuery expressions, so you can leverage some of your skills writing XPath expressions. An alternative way to express such a join would be to replace the expression `let $j := input()/jazzMusician[@ID = $id]` with `for $j := input()/jazzMusician where $j/@ID = $id`. Since XQuery 4 allows nested queries and nested loops, and any number of variables, join expressions can be very complex.

In the second part of the query we perform a join with album documents. Since the node `collaboration/result` can only have single occurrences, we can use `let` instead of `for`.

13 Efficient Querying

- Data Modeling for Efficiency 104
- Efficient Indexing 105
- Efficient Queries 106

Before we go into the details of database tuning, we should make clear that due to ongoing development and tuning of the Tamino engine the performance hints given here are only based on a snapshot of the current situation. Future versions of Tamino may perform differently.

Optimizing Tamino for efficient querying involves three steps: Data Modeling, Index Definition, Query Definition.

Data Modeling for Efficiency

First we should get the data model right:

- Your document types should implement whole business objects and documents such as customers, suppliers, purchase orders, jazz musicians, albums, etc. You should avoid "relational" designs such as First Normal Form. Business objects represented by an ensemble of flat tables are suitable for relational databases, but not for native XML databases.

> For our example, we created one document type for each of the business objects `style`, `jazzMusician`, `collaboration` and `album`.

- Avoid "all-in-one" documents. Large documents can slow down processing considerably. For example, the current Tamino version compresses documents larger than 32 KB in order to save disk space and speed up disk access, so documents above this size need more CPU time.

> For example, with XML it would be easily possible to create a single document containing our whole jazz encyclopedia (which, eventually, could grow into a size of several hundred MB). But do not expect good performance from such a design.

- If a document contains clearly identifiable *hot spots* and *cold areas*, i.e. a small area is accessed frequently whereas another large area is accessed only rarely, consider separating these two areas into two separate documents. This increases the processing speed for the frequently accessed area.

> In our example, we have stored the album reviews in separate documents. These reviews are far less likely to be accessed than the `album` document itself.

- Sometimes it can be appropriate to re-introduce redundant data elements in order to speed up retrieval. The downside to this is that updating becomes more complicated and takes more time.

For example, if we frequently need to know how many albums a jazz musician has published, retrieval performance could be improved by including this information in each `jazzMusician` document; we would no longer need to search all collaborations of a musician and then count the albums. However, we would need to update all referenced `jazzMusician` documents each time we insert, update or delete a `collaboration` document.

Efficient Indexing

The next step is to define indexes correctly:

- Associate all primary and foreign keys used in the conceptual model with indexes of search type "standard" in Tamino. Search type "standard" results in the classical database index.

  In our example, primary keys are:

  `jazzMusician/@ID`, `style/@name`, `collaboration/@ID`, `album/@albumNo`, `critic/@ID`, and `review/@URL`. We might omit `review/@URL` from this list, as we address `review` documents via URL. Internally, `review/@ino:docname` serves as a primary key.

  The schema element `tsd:unique` can be used to ensure that keys are unique. This is particularly relevant for primary keys.

  Foreign keys are:

  ```
  jazzMusician/belongsTo/style/@name
  jazzMusician/influence/influences/@ID
  collaboration/jazzMusician/@ID
  collaboration/result/@albumNo
  album/review/@URL
  review/critic/@ID
  ```

- Nodes that are used as *sort criteria* should also be defined as indexes of search type "standard".

  In our example, a typical candidate is `jazzMusician/name/last`.

- Other nodes that are expected to be used as search criteria should also be defined as indexes. However, if a node is not very selective it does not make much sense as an index. For example, a node describing the gender of a person can only take one of two values. Such a node would make a bad index, because each index value would select half of the population; however, it could make sense as part of a compound index.

  In our example, we would definitely not declare `jazzMusician/instrument/color` as an index.

- If you expect the `contains` operator (~=) to be used frequently for a specific node, define this node as a key of search type "text". This creates a word index, which speeds up ***text retrieval***

operations on the current node and child nodes. Do not use search type "text" too liberally, because it slows down write operations to the database.

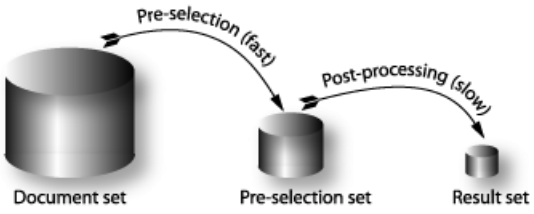
In our example, we declare `style/description` and `review/text` as text indexes. This allows us to search for words and word combinations in these elements.

- If you expect wildcard characters to be used at the beginning and end of a search word, such as `*cit*` to find "citation", "recite", "recitation", you should consider setting `word fragment index` to "true" for the database. But note that this results in a huge index and slows down write operations even more.
- If you expect queries on document nodes that are not defined in the document schema, consider setting `structureIndex` to "CONDENSED" (see ***Indexing***::***Declaring an index***). If you expect queries on optional document nodes that appear only sparingly in document instances, consider setting `structureIndex` to "FULL".

Efficient Queries

Finally, we look at the queries. In this section, the majority of the examples are based on X-Query, but equivalent processing is possible in XQuery. For examples of equivalent coding in X-Query and XQuery, see the *Performance Guide* in the Tamino documentation set.

Internal query processing can involve a pre-selection step and a post-processing step, depending on the nature of the query. If the query involves searching on one or more indexes, the pre-selection step finds the documents that satisfy the index search criteria; if the query involves search criteria that do not use indexes, the post-processing step is required.



In the pre-selection step, the indexes are used to select an intermediate result set. In the post-processing step, this set is narrowed by applying the remaining search criteria. This post-processing step involves detailed analysis of each record contained in the intermediate result set.

For example, in the query:

```
jazzMusician[belongsTo/style/@name="Bebop" and name/first="Charlie"]
```

the expression `belongsTo/style/@name="Bebop"` is processed as a pre-selection because the foreign key `belongsTo/style/@name` is defined as a standard index.

The rest of the filter expression `name/first="Charlie"` is processed during the post-processing phase because `name/first` is not defined as an index.

The same is true for the equivalent XQuery 4 expression. In

```
for $j in input()
   where $j/belongsTo/style/@name="Bebop" and $j/name/first="Charlie"
   return {$j}
```

`$j/belongsTo/style/@name="Bebop"` is executed first to construct the pre-selection set, then `$j/name/first="Charlie"` is executed.

Queries that do not have a pre-selection step (because there are no indexes among the search criteria) cause a long response time when only a few records are extracted from a large collection. You can easily determine whether a pre-selection is used with your query: put your X-Query query string in `ino:explain(...)` and Tamino will tell you whether your query involves a pre-selection and whether it involves post-processing.

In XQuery 4 we can obtain the same information by including the expression `{?explain?}` in the query prologue.

The query above, for example:

```
ino:explain(jazzMusician[belongsTo/style/@name="Bebop"
                         and name/first="Charlie"])
```

results in:

```
<xql:result>
  <ino:explanation ino:preselection="TRUE"
                   ino:postprocessing="TRUE" />
</xql:result>
```

As already explained above, this query involves both a pre-selection and a post-processing phase. Not surprisingly, both `ino:preselection` and `ino:postprocessing` have the value "TRUE".

Because Tamino automatically separates pre-selection and post-processing criteria and applies further query optimization, the sequence of search criteria in a filter expression does not matter. For example, the query

```
jazzMusician[belongsTo/style/@name="Bebop"
             and name/first="Charlie"]
```

is executed at the same speed as

```
jazzMusician[name/first="Charlie"
             and belongsTo/style/@name="Bebop"]
```

(Remember, `belongTo/style/@name` is indexed, `name/first` is not.)

Here are a few more guidelines for efficient querying:

- There is one situation in which an indexed node cannot be processed during pre-selection: the query for the non-existence of the node. If a node does not exist, its value is not contained in the index, and consequently the test for non-existence cannot rely on the index. This test is therefore processed during the post-processing phase. Depending on the size of the pre-selected document set, this can be slow.

  For example, let us assume that we had declared `jazzMusician/name/middle` as a standard index. The query for jazz musicians without a middle name:

  ```
  jazzMusician[not(name/middle)]
  ```

  would still require a scan through all `jazzMusician` documents.

- Avoid using the equality operator (=) when only a "text" index is defined, or the contains operator (~=) when only a "standard" index is defined. In both cases, Tamino correctly evaluates the query, but via post-processing! If you frequently apply both operators on the same node, consider defining it as both a standard *and* text index.

  For example, the query `style[@name~="Cool*"]` would be handled in the post-processing stage, after reading all `style` documents. This is because `style/@name` was defined as a standard index, not as a text index.

- Make use of Tamino's X-QUERY extensions to XPath. These extensions perform better than the equivalent standard XPath expressions.

  For example, use `[age between 40,65]` instead of `[age >= 40 and age <= 65]`. (For the definition of `jazzMusician/age` please see *Utilizing Server Extensions*::*Derived elements*).

- Not only is it good style to make key and search expression type-compatible (e.g. to use a string search value for an alphanumeric key or a numeric search value for a numeric key); this also ensures that you always obtain correct results. Comparing an alphanumeric constant with a numeric element, for example, causes the numeric element to be converted into a string and a string comparison to be performed. This would probably not return the expected results. The performance suffers from this conversion too.

  For example, write `[age = 55]` and not `[age = "55"]` if you have defined `age` as an element of type `integer`. Write `[@ino:id=42]` and not `[@ino:id="42"]`.

  XQuery 4, on the other hand, checks for type consistency in expressions and throws an error if you try to compare an integer with a string. (Remember that XQuery 4 supports the full XML Schema type system.) In XQuery 4 you always must specify a correctly typed literal, as in:

```
  where $j.birthdate = xs:date("1923-07-27").
```

Queries that do not use post-processing are especially useful when it is not necessary to access any documents, for example, when using the `count()` function.

14 Performance Issues

To achieve optimal transaction performance, again, proper modeling is essential:

- Avoid “relational” designs that split a business object into a multitude of “flat” documents. Updating a business object would require the updating of multiple documents, thus impairing performance. In some cases, however, it may be necessary to split a large business object into several documents. A Boeing 747, for example, can certainly be seen as a single business object, but representing its parts list in a single document would certainly lead to performance problems (see next paragraph).
- Avoid “all-in-one” documents. Very large documents are slow to update because parsing and writing them may take a long time. Also, when a single document contains several business objects, you may run into performance problems due to locking conflicts. Because locks are set at the document level, you would lock all business objects contained in the updated document, even if a certain business object is not affected by the update. This prohibits other users from concurrently accessing or updating these business objects (depending on the isolation level).

III Utilizing Server Extensions

In this chapter we introduce X-Tension, which is Tamino's mechanism for producing server extensions. We implement a few example extensions for our jazz knowledge base, and also implement some server extensions that are of general interest. Full information on server extensions is available in the document *X-Tension: Tamino Server Extensions*.

What are they Good For?

Queries

Derived Elements

Maintaining Semantic Integrity

Building Up a Library

More Examples

15 What are they Good For?

Tamino server extensions extend – not surprisingly – the functionality of the Tamino server. Server extensions are defined on the level of individual databases. They can be implemented in a variety of languages, such as C++, Java, Natural and Visual Basic.

The areas of Tamino functionality that can be extended are:

Query functions
Server extension functions can be called in the context of a Tamino query and can thus extend the query language. Note that server extension functions can only extend X-Query and XQuery 4 expressions that are interpreted by the Tamino server, not XPath expressions that are used, for example, in an XSLT stylesheet processed in a pass-thru servlet.

Mapping functions
These functions are applied when documents are stored (`onProcess`), retrieved (`onCompose`) or deleted (`onDelete`). Typically, these functions are used to implement nodes of an XML document that are *not* stored natively in Tamino. These can be nodes that are derived from the values of other nodes, or nodes that are stored outside of Tamino, for example in a file system.

Triggers
Triggers are similar to mapping functions. The difference is that a mapping function consumes a subtree, whereas a trigger does not. Typically, triggers are used to execute actions when document nodes are inserted (`onInsert`), modified (`onUpdate`) or deleted (`onDelete`). The trigger action is executed in addition to the native Tamino operation.

Server event functions
These functions are executed at the end of a server request, commit or rollback and at the end of a session (connection end). Typically, they are used for housekeeping operations.

Shadow functions
These functions are used to create index values for non-XML documents as a shadow of the original.

Init functions

These functions allow initialization operations to be executed on the server extension object prior to any query, trigger or mapping function execution.

Server extensions that are written in Java can be implemented using the Tamino X-Tension Builder tool. Alternatively they can be implemented with third-party tools and then imported into Tamino with the help of the X-Tension Object Analyzer. Typically, several server extension functions are combined into a single server extension module such as a Java package.

In the following sections we discuss the development of Java server extensions with some examples. We implement two query functions that are of interest in connection with XML Schema datatypes, and we also implement a map-out function that is of interest in connection with the jazz example shown in *From Conceptual Model to Schema*::*From Model to Schema*. The section ***More examples*** shows how to extend X-Query with the help of query functions.

16 Queries

In this chapter we implement a server extension function that compares two `date` or `dateTime` values. As we have seen in *From Conceptual Model to Schema*::*Constraints across documents*, the comparison of values of these datatypes using XPath's native facilities is awkward; anything that facilitates this is beneficial. Note that this is only a deficiency of XPath 1.0; in contrast, both XPath 2.0 and XQuery support XML Schema datatypes, including date and time formats. The module concept of XQuery offers a powerful mechanism for managing user-defined functions.

In addition, we implement a function that returns a `dateTime` string of the current time of day. (The complete source code including an *Install.xml* file and Javadoc HTML file is contained in the directory *sxsjxsd* in the documentation set.)

We implement these functions in Java, and because we are lazy we implement them by using Java's sophisticated calendar support. After all, object-oriented programming is about re-use.

We start the Tamino X-Tension Builder and create a new server extension which we call `xsd`. We call the package `tamino.SXS.xsd` and the class simply `xsd`. The tool will generate a code framework for a Tamino server extension.

We can now add our new query functions as new methods into this framework. To do so, we use the menu function **Add Function**:

- The first function we add is called `dtComp()`. We define it as a query function. The tool asks us for the type of the result, which we set to `integer`. Then we add two operands, `op1` and `op2`, of type `charstr`.
- The second function is called `current()`. Again, we define it as a query function and set the result type to `charstr`. There are no operands.

The X-Tension Builder generates the required method code frames into the class code. We can now implement our custom logic (**emphasized**):

```
// xsd.java: Implementation of Server Extension xsd
//
// Tamino Server Extension xsd
//
// $javadoc:on

package tamino.SXS.xsd;
import com.softwareag.ino.sxs.*;
import java.text.SimpleDateFormat;
import java.util.Date;

// Javadoc comments.
// TODO: Add more detailed description if necessary
/**
 * Tamino Server Extension xsd
 * @author My name
 * @version 1.0
 */

public class xsd extends ASXJBase {
        // Version information for current Server Extension xsd.
        // TODO: Change SXS Version here if necessary:
        static final SXSVersion sxsVersion = new SXSVersion (1, 0);

        // Description of current Server Extension:
        // TODO: Change this string if necessary
        static final String sxsAbout = "Tamino Server Extension xsd";

        // TODO: Enter further class variables here.

        /**
         * The default constructor
         * (No other constructor allowed.)
         */
        public xsd () {
                // TODO: enter SXS initialization here.
        }

        // Description of Server Extension Function dtcomp
        // TODO: Change description if necessary
        static final String dtcompAbout = "Query Function dtcomp";

       /**
         * comparison of XSD date/time
         * @param op1 parameter (XML Schema date, dateTime, time)
         * @param op2 parameter (XML Schema date, dateTime, time)
         * @return comparison result:
         *         -1 (op1 < op2), 0 (op1 = op2), 1 (op1 > op2)
         */
        public int dtcomp (String op1, String op2)
                        throws java.text.ParseException
        {
```

```
        // To delete dtcomp, remove it here and from Install.xml.
      Date d1 = toDate(op1);
      Date d2 = toDate(op2);
      return d1.compareTo(d2);

    }

    // Description of Server Extension Function current
    // TODO: Change description if necessary
    static final String currentAbout = "Query Function current";

    public String current ()
    {
        // To delete current, remove it here and from Install.xml.
       SimpleDateFormat df2 =
       new SimpleDateFormat ( "yyyy-MM-dd'T'HH:mm:ss" );
       return df2.format(new Date());

    }

  /**
     * Convert lexical XML Schema date/time representation
     * to Java Date format
     * @param s input string
     * @return Date value
     */
  private Date toDate(String s)
                throws java.text.ParseException
  {
     // determine formatting string
     String f = (s.indexOf("T",0) >= 0 ?
                 "yyyy-MM-dd'T'HH:mm:ss" :
                 (s.indexOf(":", 1) >= 0 ?
                           "HH:mm:ss" : "yyyy-MM-dd"));
     // check for explicit time zone
     int p = Math.max (s.indexOf("+", 8),s.indexOf("-", 8));
     if (p >= 0) {
       f += "z";
     // indicate time zone in formatting string
       s = s.substring(0,p-1)+"GMT"+s.substring(p);
           // keep SimpleDateFormat happy
     }
     else if (s.charAt(s.length()-1) == 'Z') {
                        // check for UTC time zone
       f += "z";
       s = s.substring(0,s.length()-1) + "UTC";
     }
     // create SimpleDateFormat object
     SimpleDateFormat df = new SimpleDateFormat ( f );
     // and use it as a parser
     return df.parse(s);
  }
```

```
}
```

Tip: A good way of developing such a server function is first to create and debug it in your preferred Java development environment, and then copy the code into the X-Tension Builder.

Along with the Java code, the X-Tension Builder has created a control file, called *Install.xml*. Remember that Tamino server extensions can be written in a variety of programming languages. This file describes the specific extension module in a generic, language-neutral format. It also specifies the datatypes of input and output parameters in terms of server extensions. The following table shows how Java datatypes relate to server extension specific types:

server extension type	Java input and result parameters	Java output and in/out parameters
ino:XML-OBJ (i.e. node list)	String	StringBuffer
xs:string	String	StringBuffer
xs:boolean	boolean	BooleanRef
xs:int	int	IntRef
xs:double	double	DoubleRef
xs:float	float	

And this is how an *Install.xml* file looks:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ino:Administration
    xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <ino:Object
        Name="xsd"
        Id="tamino/SXS/xsd/xsd"
        Infrastructure="Java">
        <ino:About
            Author="My name"
            CreationDate="2008-08-14"
        />
        <ino:Function
            Name="dtcomp"
            Usage="Query">
            <ino:Parameter
                Name="op1"
                TSDType="xs:string"
            />
            <ino:Parameter
                Name="op2"
                TSDType="xs:string"
            />
```

```
            <ino:Parameter
                Name="result"
                TSDType="xs:int"
            />
            <ino:About>Query Function dtcomp</ino:About>
        </ino:Function>
        <ino:Function
            Name="current"
            Usage="Query">
            <ino:Parameter
                Name="result"
                TSDType="xs:string"
            />
            <ino:About>Query Function current</ino:About>
        </ino:Function>
    </ino:Object>
</ino:Administration>
```

As you can see, this file describes all the functions implemented in that particular server extension module, together with their respective parameters. This file must be modified manually if we subsequently decide to change the name of any function, the name and number of parameters, or their type.

After compiling, we pack this class into an *.sxp* file using the **Pack** function of the X-Tension Builder. With the help of the Tamino Manager we can now install the new server extension into our `jazz` database. Thereafter, we can use the new functions within queries, for example:

```
jazzMusician[xsd.dtComp(birthDate,"1920-01-01") >= 0]
```

which returns all jazz musicians born in 1920 or later;

```
jazzMusician[xsd.dtComp(birthDate,xsd.current()) > 0]
```

which should result in an empty set of documents.

In XQuery 4 we can similarly write:

```
for $j in input()/jazzMusician[xsd.dtComp(birthDate,"1920-01-01") >= 0]
  return $j
```

and:

```
for $j in input()/jazzMusician[xsd.dtComp(birthDate,xsd.current()) > 0]
  return $j
```

The first case can be expressed solely with the means of XQuery 4 because XQuery 4 supports XML Schema datatypes. We could write:

```
declare namespace xs="http://www.w3.org/2001/XMLSchema"
for $j in input()/jazzMusician[birthDate >= xs:date("1920-01-01")]
  return $j
```

The second case, however, still requires a server extension to fetch the current date.

Note that search criteria that involve server extension functions are always processed in the post-processing phase of a query (see *From Schema to Tamino*::*Efficient Queries*). So, even if `birthDate` is defined as an index, this search criterion still results in a scan through all documents.

Tip: An alternative way to develop Tamino Server Extensions is to use the X-Tension Object Analyzer. This tool can be invoked from the Tamino Manager. Using this method, we would first develop the necessary Java classes using an IDE of our choice. Then, we can load the executable (*.class* or *.jar* file) into the X-Tension Object Analyzer. Now we can edit the details of the imported executable, such as the server extension name, description, author, and help files. The **Pack Object** function creates a server extension package file that can be installed in a Tamino database. This is a map-out function with the `onCompose` property defined.

17 Derived Elements

Server extensions are also useful in the area of derived fields. A derived field is an information element that is not stored within a document, but is computed when the document is retrieved. This is usually done to avoid redundant data within a document, and in cases when the value of a document node depends on the environment.

In XQuery, derived values can be generated by using constructors in the `return` clause. The remainder of this section describes how to deal with derived elements using X-Query.

A typical example is a person's age. We can store a person's date of birth in a document, but we cannot store his or her age because it changes over time. However, with server extensions we can define an `age` element in a document that describes a person, for example in `jazzMusician`. When such a document is retrieved, we must compute the value of this element by subtracting the date of birth from the current date.

Because we rely on values from other document nodes – here `birthDate` – during the computation of a derived field, we must make use of a callback function. There are three types of callback functions: XML, ODBC and System. It is the XML callback functions that we need, because we want to query Tamino for the `birthDate` element.

Here is the complete code for the server extension. The callback code is highlighted.

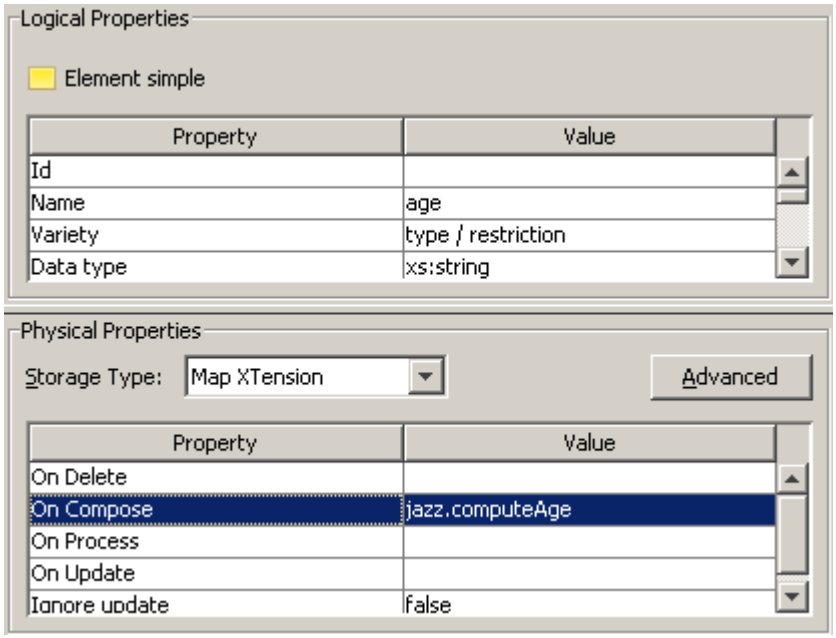
```
//      encyclopedia_jazzMusician.java:
//         Implementation of server extension encyclopedia_jazzMusician:

package tamino.SXS.jazz.encyclopedia.jazzMusician;
import com.softwareag.ino.sxs.*;
import java.util.Date;
import java.text.SimpleDateFormat;
// Javadoc comments.
/**
 * All extensions for jazz database
 * @author Berthold Daum
 * @version 1.0
```

```
 */
public class encyclopedia_jazzMusician extends ASXJBase {
  /**
   * The default constructor
   * (No other constructor allowed.)
   */
        public encyclopedia_jazzMusician () {
        }
        /**
         * compute age from birth date (map-out function)
         * @param object_id parameter
         * @param element_id parameter
         * @param document parameter
         */
  public void computeAge (int object_id,
                          int element_id,
                          StringBuffer document)
              throws java.lang.Exception
  {
    // create query string for callback
    String xmlQuery =
           "jazzMusician[@ino:id='"+object_id+"']/birthDate";
    // create new string buffer for call back results
    StringBuffer response = new StringBuffer(1024);
    // callback: XQL query
    int ret = SxsXMLXql ("encyclopedia", xmlQuery, response);
    // Here we dive deep into the rather complex error
    // handling for server extensions callbacks.
    if (ret != 0)
      {
        // identify the callback error
        int msgNo = SxsGetMsgNo ();
        switch (msgNo) {
        case INO_ERROR:
          StringBuffer msgBuf = new StringBuffer ( );
          // Get the Tamino Server message number
          int inoMsgNo = SxsGetInoMsgNo ( );
          // Get the Tamino Server messageline
          ret = SxsXMLGetMessage (msgBuf);
           throw (new Exception("INO_ERROR: "
                                    +inoMsgNo+" "+msgBuf));
        default:
        // Get the SXS message corresponding to callback error
          String msg = SxsGetMsgText ();
          throw (new Exception("SXS Error: "+msgNo+" "+msg));
        }
      }  // error handling done, now the real logic
      // Remember, we had obtained the birthDate in "response"
      // Convert into string
      String  docstr = response.toString();
      // We have to isolate the value of the birthDate
      //  element from the Tamino response string.
```

```
      // Could be properly done with SAX,
      // but to be brief we do it by hand.
      int t = docstr.indexOf("<birthDate", 0);
      // we have to scan for tag begin and end separately
      // because the tag includes attributes (ino:id).
      int b = docstr.indexOf('>',t+10)+1;
      // Find end of element
      int e = docstr.indexOf("</birthDate>", b);
      // Convert content into Date format
      SimpleDateFormat df =
              new SimpleDateFormat ( "yyyy-MM-dd" );
      Date d1 = df.parse(docstr.substring(b,e-1));
      // Now extract years and months
      // These methods are deprecated, but we use them anyway.
      int y1 = d1.getYear();
      int m1 = d1.getMonth();
      Date d2 = new Date();
      int y2 = d2.getYear();
      int m2 = d2.getMonth();
      // Compute age
      int age = y2-y1;
      // Adjust age, if we are still
      //   before this year's birthday
      if ((m1 > m2) ||
           ((m1 == m2) && (d1.getDate() > d2.getDate()) ))
                age--;
      // Now pack into appropriate tags
      docstr = "<age>"+age+"</age>";
      // Write element to result buffer
      document.replace (0, docstr.length(), docstr);
  // To delete computeAge, remove it here and from Install.xml.
  }
}
```

After the usual process – compilation, packing, and installation – we can use this server extension. First, we must define an element `<age>` in the corresponding `jazzMusician` schema:



The resulting schema fragment looks like this:

```
<xs:element name = "age" type = "xs:short">
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:physXNode>
          <tsd:mapXTension>
            <tsd:onCompose>
              encyclopedia_jazzMusician.computeAge
            </tsd:onCompose>
          </tsd:mapXTension>
        </tsd:physXNode>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

After making this schema known to the database, we can store `jazzMusician` document instances. These instances must include dummy `<age>` elements:

```
<age>0</age>
```

If we did not supply such elements (if we had defined `minOccurs="0"` in the element declaration), the `computeAge` server extension would not be called when the document is retrieved, and no age would be computed and displayed.

Tip: If a server extension derives a value for an attribute rather than for an element, and a default value for the attribute is specified in the schema, it is not necessary to provide a

dummy value for the attribute. This is because Tamino stores document instances with default attribute values if explicit attribute values are not provided.

Tip: You can allow a default value to be generated for an element or an attribute by using a server extension Query function. Refer to the section *tsd:default* in the document *Tamino XML Schema Reference Guide* for details.

When we now query `jazzMusician` documents, the current age of each jazz musician is displayed. For example, the following query issued in October 2004:

```
jazzMusician[@ID="ColtraneJohn"]
```

results in:

```
<jazzMusician ino:id="1" type="instrumentalist"
              ID="ColtraneJohn">
  <name>
    <first>John</first>
    <last>Coltrane</last>
  </name>
  <birthDate>1926-09-23</birthDate>
  <age>78</age>
</jazzMusician>
```

The same query issued in October 2002 results in an age of 76.

The derived field can also be used in queries:

```
jazzMusician[age > 70]
```

However, because we cannot index derived fields, search criteria involving derived fields are always processed in the post-processing phase of a query (see *From Schema to Tamino*::*Efficient Queries*).

18 Maintaining Semantic Integrity

Triggers are the mechanism that is best suited for maintaining semantic integrity in a database (if we allow for a closed-world assumption). Triggers are well known from SQL. Most relational database management systems implement trigger mechanisms.

Let us assume that we have stored several `album` documents within our encyclopedia collection. In addition, we have stored several `review` documents that relate to some of those album documents. When we delete an `album` document, we want to make sure that the `review` elements relating to the deleted album document are also removed from the database.

We can achieve this by defining an `onDelete` trigger to the `album` node of the `album` document type.

This trigger is called immediately before the `album` node is deleted.

```
<xs:element name = "album" >
  <xs:annotation>
    <xs:appinfo>
      <tsd:elementInfo>
        <tsd:logical>
          <tsd:trigger>
            <tsd:onDelete type = action>
              encyclopedia_album.deleteRelatedReviews
            </tsd:onDelete>
          </tsd:trigger>
        </tsd:logical>
      </tsd:elementInfo>
    </xs:appinfo>
  </xs:annotation>
  ...
</xs:element>
```

Sample trigger code is shown in the following example. The deleted `album` document node is passed to the method `deleteRelatedReviews`, which implements the trigger. This allows us to

extract the `album/@albumNo` attribute value, and then to delete any review instances that specify this value in their `review/album/@albumNo` node. These cascading `delete` operations are performed via server extension callbacks.

```
package tamino.SXS.jazz.encyclopedia.album;
import com.softwareag.ino.sxs.ASXJBase;

public class encyclopedia_album extends ASXJBase {

  /**
   * The default constructor
   * (No other constructor allowed.)
   */
  public encyclopedia_album() {
  }
  /**
   * Delete all reviews relating to this album instance (used as trigger function)
   * @param object_id parameter
   * @param element_id parameter
   * @param document parameter
   */
  public void deleteRelatedReviews(
    StringBuffer collection,
    StringBuffer doctype,
    StringBuffer inoId,
    String document)
    throws java.lang.Exception {
    // We rely on the fact that albumNo is an attribute of the root element
    // (the official way would be to employ a parser)
    int aPos = document.indexOf("albumNo");
    if (aPos < 0)
      throw new Exception("No albumNo attribute");
    int qaPos = document.indexOf('"', aPos + 8);
    int qePos = document.indexOf('"', qaPos + 1);
    String albumNo = document.substring(qaPos, qePos + 1);

    // create Delete query string for the callback
    String xmlQuery = "review[album/@albumNo=" + albumNo + "]";
    // create new string buffer for call back results
    StringBuffer response = new StringBuffer();
    // callback: delete all documents matching this query
    int ret = SxsXMLDelete("encyclopedia", xmlQuery, response);
    if (ret != 0)
      processError();
  }

  // Here we dive deep into the rather complex error
  // handling for server extensions callbacks.
  private void processError() throws Exception {

    // identify the callback error
```

```
    int msgNo = SxsGetMsgNo();
    switch (msgNo) {
      case INO_ERROR :
        StringBuffer msgBuf = new StringBuffer();
        // Get the Tamino Server message number
        int inoMsgNo = SxsGetInoMsgNo();
        // Get the Tamino Server messageline
        int ret = SxsXMLGetMessage(msgBuf);
        throw (new Exception("INO_ERROR: " + inoMsgNo + " " + msgBuf));
      default :
        // Get the SXS message corresponding to callback error
        String msg = SxsGetMsgText();
        throw (new Exception("SXS Error: " + msgNo + " " + msg));
    }
  }
}
```

Note that triggers can be cascaded. We could, for example, equip `review` documents with an `onDelete` trigger that removes any `critic` instances that are only referred to by the deleted `review` instance. The deletion of an `album` document could thus cause the deletion of a `critic` document. This is conceptually debatable: both reviews and critics are first class business objects, so their existence should not depend on other business objects.

19 Building Up a Library

Building up a library of server extensions helps to avoid the duplicate development of server extension functions and facilitates software reuse. The most systematic approach to the construction of such a library is to use a multi-level strategy.

On the server level

On this level, we define the server extension functions that are common to all databases on the server. These are functions that, for example, extend basic functionality or implement common corporate functionality.

Typically, these functions are grouped into packages that contain functions with related functionality. For example, we create one package for the support of XML Schema datatypes; we create another package to add more functionality to X-Query and XQuery 4; and we create another package for triggers.

In Tamino, server extensions are always installed into particular databases. Therefore, we install these common server level extensions into each database on the server.

We should use short names for these server extensions because on this level most server extensions will extend the query functions and it is awkward to type long name prefixes in queries. (See below for the possibility to rename server extension functions.) In the section ***Queries*** we called the server extension for the XML Schema datatypes `xsd` and the corresponding package *tamino.SXS.xsd*.

A typical example are the functions `dtComp()` and `current()` in the package `tamino.SXS.xsd`. These functions are of general interest because they extend the functionality of X-Query to the new date and time datatypes introduced with XML Schema. Additional functions are shown in the section ***More examples***.

On the database level

On the level of a single database, we define the server extension functions that are common to all collections within the database. Again, we should give relatively short names to the server extension and use the following naming pattern for the package names: *tamino.SXS.{database}.{extension}*.

On the collection level

Here we define the server extension functions that are common to all document types within a particular collection. We should aim to implement these extensions in a single package. We can then use the collection name as the name of the server extension. The package name should follow the pattern: *tamino.SXS.{database}.{collection}*

On the document type level

Server extensions on the document type level contain the server extension functions that are specific to that particular document type. Again, we should aim to implement these extensions in a single package. We use the following naming pattern for the server extension: `{collection}_{doctype}`

and name the package according to the following pattern: *tamino.SXS.{database}.{collection}.{doctype}* .

Tip: Individual server extensions functions can be renamed with the help of the Tamino Manager. If, for example, `xsd.dtComp` should be inconvenient for queries, we can assign a different external name to this function, for example `compare-date`.

20 More Examples

- concat 136
- contains 138
- substringBefore 139
- substringAfter 140
- substring 141
- trim 142
- normalizeSpace 144
- stringLength 146
- qdoc 147

In this section we provide a few more examples of server functions. We have implemented:

- some of the XPath functions that did not find their way into X-Query;
- particular string functions which might prove useful in both X-Query and XQuery 4;
- a function similar to the XSLT `document()` function.

We call this server extension module `xqx` and name the package accordingly `tamino.sxs.xqx`. The complete source code, including the file *Install.xml* and the Javadoc HTML file, is contained in the directory *sxsjxqx* in the documentation set.

concat

Concatenates strings.

Syntax

```
public String concat (String op1, String op2)
```

```
public String concat3 (String op1, String op2, String op3)
```

```
public String concat4 (String op1, String op2, String op3, String op4)
```

Description

The original XPath `concat()` string function allows any number of arguments (at least 2). We have implemented the functions `concat`, `concat3`, and `concat4` to cover cases with two, three, and four arguments. The implementation could hardly be simpler:

```
public String concat (String op1, String op2) {
  return op1+op2;
}
```

```
public String concat3 (String op1, String op2, String op3) {
  StringBuffer sb = new StringBuffer(op1);
  sb.append(op2);
  sb.append(op3);
  return sb.toString();
}
```

```
public String concat4 (String op1, String op2,
                       String op3, String op4) {
  StringBuffer sb = new StringBuffer(op1);
  sb.append(op2);
  sb.append(op3);
  sb.append(op4);
  return sb.toString();
}
```

Examples

```
xqx.concat(name/last, name/first)
```

constructs an ID from last name and first name.

```
xqx.concat3(birthDate,"T","04:30:00")
```

appends a time to a date.

contains

True if op1 contains op2.

Syntax

```
public boolean contains (String op1, String op2)
```

Description

The string function `contains()` is a bit more demanding:

```
public boolean contains (String op1, String op2) {
  return (op1.indexOf(op2) >= 0);
}
```

Examples

```
xqx.contains(title,"Moon in June")
```

is true for all titles containing "Moon in June". In contrast to the X-Query contains-operator (~=), this function is case sensitive.

substringBefore

Returns substring before op2.

Syntax

```
public String substringBefore (String op1, String op2)
```

Description

The original XPath name is `substring-before()`.

```
public String substringBefore (String op1, String op2) {
  int i = op1.indexOf(op2);
  if (i > 0) return op1.substring(0,i);
  return "";
}
```

Examples

```
xqx.substringBefore(performedAt/time,"T")
```

returns the date part of `dateTime` element `performedAt/time`.

substringAfter

Returns substring after op2.

Syntax

```
public String substringAfter (String op1, String op2)
```

Description

The original XPath name is `substring-after()`.

```
public String substringAfter (String op1, String op2) {
  int i = op1.indexOf(op2);
  if (i >= 0) {
    int b = op2.length()+i;
    if (b < op1.length()) return op1.substring(b);
  }
  return "";
}
```

Examples

```
xqx.substringAfter(performedAt/time,"T")
```

returns the time part of `dateTime` element `performedAt/time`.

substring

Returns substring from position p with length l.

Syntax

```
public String substring (String s, int p, int l)
```

Description

Two `substring` functions are provided in XPath:

- `substring(s, p, l)` returns a substring of `s` starting at position `p` with length `l`.
- `substring(s, p)` returns a substring of `s` starting at position `p` up to the end of the string.

Because Tamino server extensions do not support method overloading, we implement only the three-argument form. The length `-1` indicates the two-argument version (substring up to end of string).

```
public String substring (String s, int p, int l) {
  p--;          // Java counts from 0, Xpath from 1
  if (l < 0) return s.substring(p);
  return s.substring(p,p+l);
}
```

Examples

```
xqx.substring(1999-05-01,6,2)
```

returns the month (`"05"`).

```
xqx.substring(1999-05-01,6,-1)
```

returns the month and the day (`"05-01"`).

trim

Removes white space from beginning and end.

Syntax

```
public String trim (String s)
```

Description

Here we implement a function that is not available in XPath but works similarly to the XPath string function `normalize-space()`. However, `trim()` only removes leading and trailing white space from a string, leaving white space in the interior of the string intact. Again, the implementation is trivial:

```
public String trim (String s) {
  return s.trim();
}
```

Examples

If a document instance of document type `style` contains an element:

```
<name>
    swing
</name>
```

then the simple path expression:

```
style/name
```

returns "swing" with leading and trailing carriage-return characters and blanks. In many cases this is unwanted, for example if we want to use the string as a search string or concatenate it with other strings.

```
xqx.trim(style/name)
```

removes these unwanted white space characters.

normalizeSpace

Normalizes white space.

Syntax

```
public String normalizeSpace (String s)
```

Description

The full implementation of `normalize-space()` requires a bit more coding:

```
public String normalizeSpace (String s) {
  StringBuffer sb = new StringBuffer(s.length());
  boolean whiteSpace = true;
  for (int i=0; i < s.length(); i++) {
    char c = s.charAt(i);
    if ((c==' ')||(c==13)||(c==10)||(c==9)) {
      if (!whiteSpace) {
        whiteSpace = true;
        sb.append(' ');
      }
    } else {
      sb.append(c);
      whiteSpace = false;
    }
  }
  if (whiteSpace) {
    sb.deleteCharAt(sb.length()-1);
  }
  return sb.toString();
}
```

Examples

If a document instance of document type `album` contains an element:

```
<title>
    Open Up
    (Whatcha gonna do for the rest of your     life?)
</title>
```

then the simple path expression:

```
album/title
```

returns the title with all carriage-return, linefeed, tab, and blank characters.

```
xqx.normalizeSpace(album/title)
```

normalizes this string to:

```
    Open Up (Whatcha gonna do for the rest of your life?)
```

stringLength

Returns length of string.

Syntax

```
public int stringLength (String s)
```

Description

The original XPath name is `string-length()`. The implementation is trivial:

```
public int stringLength (String s) {
  return s.length();
}
```

Examples

```
xqx.stringLength(name/last)
```

returns the length of the element `name/last` including white space characters.

```
jazzMusician[xqx.stringLength(./name/last)=9]
```

returns all jazz musicians whose last name is 9 characters long.

qdoc

Fetches document specified by path.

Syntax

```
public String qdoc (String qpath)
```

Description

Finally, here is a function that is similar to the `document()` function in XPath, but constrains itself to documents stored in the same database. This allows us to implement the retrieval of the document via a Tamino callback function, instead of retrieving the document via an HTTP request.

The function takes one argument, namely an X-Query expression that identifies the document (or document part). This expression must include the collection name. Before returning a result, it strips off all of Tamino's packaging for query results and returns the data as a vanilla XML node (or node list).

```
public String qdoc (String qpath)
                   throws java.lang.Exception {
```

```
  // split operand into collection and relative path
  int p = qpath.indexOf("/", 0);
  if (p<0)
   throw(new Exception("Proper syntax is collection/path"));
  StringBuffer response = new StringBuffer(1024);
  // do query to Tamino via SXS callback
  int ret = SxsXMLXql (qpath.substring(0,p),
                       qpath.substring(p+1), response);
  if (ret != 0)
    // Error handling as shown in section Derived elements
    // ....
  }
  // Create document string
  String  docstr = response.toString();
  // We have to isolate the returned node(s)
  int t = docstr.indexOf("<xql:result>", 0)+12;
  if (t >= 12) {
    int e = docstr.indexOf("</xql:result>", t);
    return docstr.substring(t,e);
  }
  return "";
}
```

Examples

Since `qdoc()` – in contrast to `document()` – does not require the full specification of a URL and removes the Tamino response wrapping, queries using `qdoc()` are significantly simpler. The following query finds all collaborations in which jazz musicians whose last name is "Parker" participated:

```
collaboration[jazzMusician/@ID=xqx.qdoc(
             "encyclopedia/jazzMusician[name/last='Parker']"
             )/@ID]
```

The next query finds all jazz musician documents taking part in such collaborations. We see that `qdoc()` can be nested. Note that we have to use the `"` notation for the innermost level of string demarcations.

```
jazzMusician[@ID = xqx.qdoc(
  "encyclopedia/collaboration[jazzMusician/@ID=xqx.qdoc(
   'encyclopedia/jazzMusician[name/last=&quot;Parker&quot;]'
  )/@ID]")/jazzMusician/@ID]
```

Similarly, the following query returns all `album` documents that are the result of such a collaboration:

```
album[@albumNo = xqx.qdoc(
  "encyclopedia/collaboration[jazzMusician/@ID=xqx.qdoc(
   'encyclopedia/jazzMusician[name/last=&quot;Parker&quot;]'
  )/@ID]")/result/@albumNo]
```

IV Rapid Application Development with Tamino

In this chapter, we show how Tamino's pass-thru servlet can be used to generate customized HTML pages from XML data stored in Tamino. We give a short introduction to XSLT and discuss its advantages and disadvantages. Using XLink, we show how navigational structures can be modeled and transformed into HTML hyperlinks. Finally, we show how XQuery 4 can be used for prototyping purposes, and discuss the pros and cons of the various approaches to application prototyping.

Introduction to XSLT

Mapping a schema to a web page

Navigation with XLink

The Tamino JavaScript API

XSLT summary

Rapid Prototyping with XQuery 4

21 Introduction to XSLT

- Procedural Transformation 152
- Rule-Based Transformation 158
- Limitations of XSLT 162
- Using Style Sheets with Tamino 162

The “official” method for transforming XML into other formats (often presentation formats) is XSLT (eXtensible Stylesheet Language: Transformations). Historically, XSLT had been a part of the XSL (eXtensible Stylesheet Language) specification, but XSL was split into three parts: XPath, XSLT, and XSL Formatting Objects (XSL-FO). XSL-FO was designed as the presentation format for XML. However, it plays currently only a minor role since most of its functionality is covered by HTML+CSS. XSL-FO is usually an intermediate step when generating PDF from an XML document.

XSLT is now a recommendation in its own right. It enables style-sheet controlled transformations from one XML document format into another document format, which can be either XML or non-XML. XSLT can, for example, be used to transform presentation-neutral XML data into presentation formats such as HTML, XHTML, XForms, WML, SMIL, SVG, etc. In the chapter *From Conceptual Model to Schema*::*Integrity* we already discussed other applications for XSLT, such as constraint checking and generating XML Schema from XMI.

Although XSLT is quite powerful, it has some deficiencies that have led to the development of various extensions. Also, programmers who are familiar with imperative languages such as Java or C++ sometimes find it hard to think in XSLT's rule-based structures. For the transformation into HTML, however, most of the XSLT coding can be avoided by the use of XSLT generators, which allow visual construction of the resulting web page (or visual mapping of XML elements to HTML elements of an existing web page) and generate most of the required XSLT code. Examples of such generators are Altova's XML Spy, eXcelon's Stylus and Whitehill's XSL Composer.

Such tools are useful to develop stylesheets that map XML documents onto individual HTML pages. However, when we want to create generic transformations (for example, where the final layout depends on the document type and/or on the content), or when we need stylesheets to produce output other than HTML, we have to dig into XSLT programming. In the following sections we give a short introduction.

Procedural Transformation

The basic construct in XSLT are templates. Each XSLT stylesheet must consist of at least one template. A template can be explicitly invoked by name, or it can be implicitly applied via pattern matching according to the `match` expression defined in the head of the template. This allows two programming styles in XSLT which can be mixed freely, namely rule-based programming and procedural programming.

Rule-based programming

This is a more declarative approach. Rules (i.e. templates with a `match` expression) specify which elements of the input document they apply to, and how they transform these elements. Rules are applied recursively. The programmer describes the transformation in terms of logic and is not concerned with the sequence of execution.

Procedural programming

This programming style is easier to understand for programmers with experience in imperative languages such as Java or C. The programmer describes to the XSLT processor exactly what to do and in which sequence. The XSLT style sheet looks very much like the target document, with interspersed XSLT instructions to fill in the blanks.

To support procedural programming, XSLT provides the following operations:

Control structures.

XSLT instructions such as `xsl:for-each`, `xsl:if`, and `xsl:choose` provide procedural control structures for loops, conditional execution and case structures. The result of an `xsl:for-each` instruction can be sorted with an `xsl:sort` instruction and numbered with the `xsl:number` instruction.

The instruction `xsl:call-template` is used to invoke a template by name (recursive calls are possible). Parameters can be passed to the invoked template but it is not possible to return results to the caller.

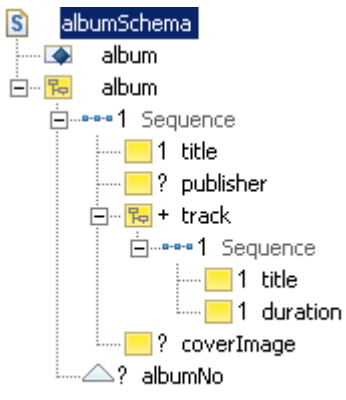
The instruction `xsl:apply-templates` can be used to start rule-based processing (see ***Rule-Based Transformation***).

Accessing content.

The `xsl:value-of` instruction writes the content of a node or node list to the output stream as text. The `xsl:copy-of` instruction writes the content of a node or node list to the output stream in its original form.

Here is an XSLT example that transforms `album` instances into an HTML page. We have extended the `album` schema from the chapter *From Conceptual Model to Schema*::*From Model to Schema* to include some more information:

Schema	Instance

	`<?xml version="1.0" encoding = "UTF-8"?>`
	`<?xml-stylesheet type="text/xsl" href="album.xsl"?>`
	`<album xmlns="http://www.softwareag.com/tamino/doc/`
	`examples/models/jazz/encyclopedia"`
	`albumNo="BGJ-47">`
	`<title>Blues House Jam</title>`
	`<track>`
	`<title>Post Election Jam I</title>`
	`<duration>PT19M35S</duration>`
	`</track>`
	`<track>`
	`<title>Post Election Jam II</title>`
	`<duration>PT20M35S</duration>`
	`</track>`
	`<coverImage>`

Schema	Instance
	post-election-jam.jpg </coverImage> </album>

The stylesheet programming is strictly procedural and deterministic. It is the stylesheet that defines the layout of the resulting HTML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- Make sure we generate HTML output -->
<xsl:output method="html" indent="yes"/>
<!-- Just a single rule for the root node -->
<xsl:template match="/">
<!-- Generate HTML document root -->
<html><head/><body>
  <!-- Select album node -->
  <xsl:for-each select="album">
    <!-- The usual nested tables -->
    <table><tr><td>
      <table width="100%">
        <tr bgcolor="silver">
          <td>
            <!-- Title element as headline -->
            <h2><xsl:value-of select="title"/></h2><br/>
            <!-- Test if we have a publisher element -->
            <xsl:if test="publisher">
              <!-- if yes generate publisher entry -->
              Publisher:
                <xsl:value-of select="publisher"/><br/>
              </xsl:if>
              <!-- Generate album number entry -->
              AlbumNo:
              <xsl:value-of select="@albumNo"/>
          </td>
          <!-- Test if we have a cover image -->
          <xsl:if test="coverImage">
            <!-- if yes generate image reference -->
            <td>
              <img src="{coverImage}" alt="{title}"/>
            </td>
          </xsl:if>
        </tr>
      </table>
    </td></tr>
    <tr><td>
      <!-- now do the tracks -->
      <br/><h4>Tracks</h4>
```

```
      <table width="100%" >
        <!-- We may have multiple tracks, therefore loop -->
        <xsl:for-each select="track">
          <tr bgcolor="silver">
            <td>
              <!-- Print track number -->
              <xsl:number value="position()" format="1-"/>
              <!-- Print title of track element -->
              <xsl:value-of select="title"/>
            </td>
            <!-- Print duration -->
            <td align="Right">
              <!-- Convert duration to mm:ss format -->
              <xsl:value-of select=
                   "substring-before(substring-after(duration,'T'),'M')"/>:
               <xsl:value-of select=
                   "substring-before(substring-after(duration,'M'),'S')"/>
            </td>
          </tr>
        </xsl:for-each>
      </table>
    </td></tr></table>
  </xsl:for-each><br/>
</body></html>
</xsl:template>
</xsl:stylesheet>
```

To implement the stylesheet logic we have used the XSLT instructions discussed above. Optional elements are included in an `<xsl:if>` block to suppress the decoration (such as "Publisher:") if there is no `publisher` element.

Applying this stylesheet to the above XML document instance results in the following HTML file:

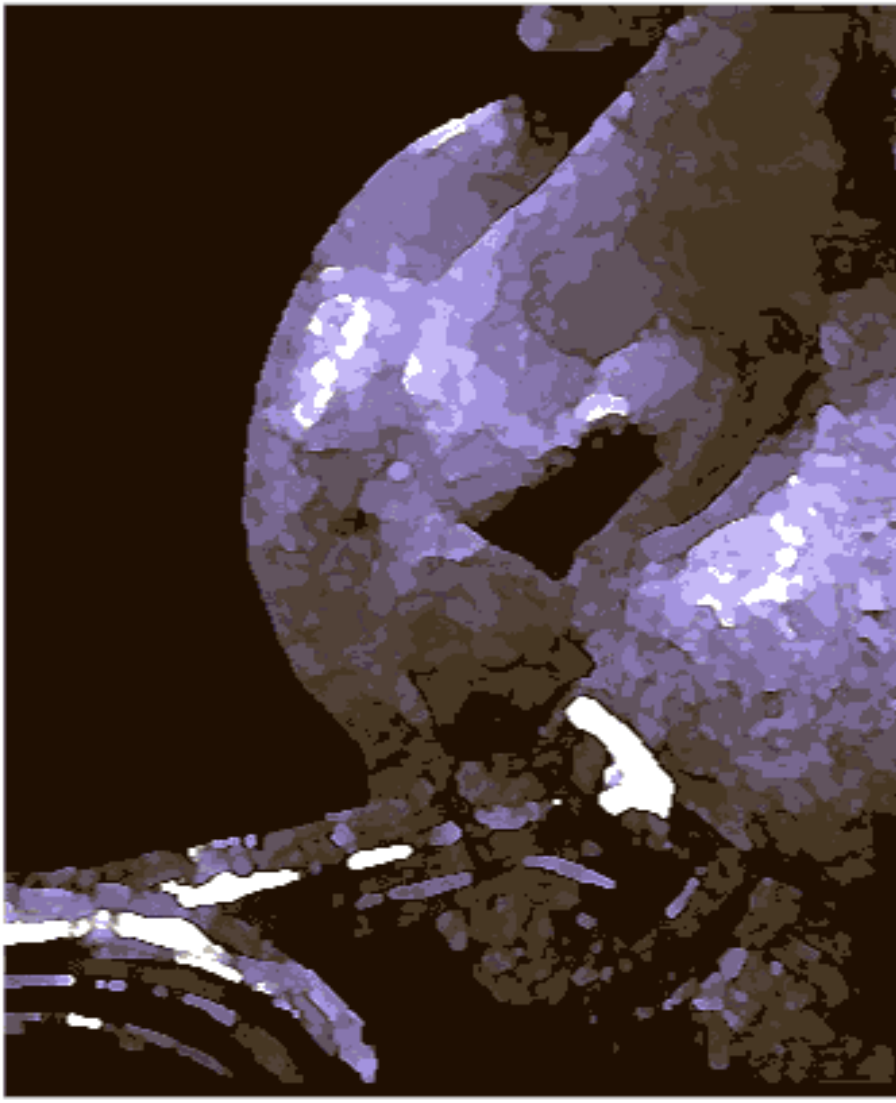
```
<html>
   <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=utf-8">
   </head>
   <body>
      <table>
         <tr>
            <td>
               <table width="100%">
                  <tr bgcolor="silver">
                     <td>
                        <h2>Blues House Jam</h2><br>
                        ProductNo:
                        BGJ-47
                     </td>
                     <td><img src="post-election-jam.jpg"
                              alt="Blues House Jam">
```

```
                    </td>
                 </tr>
              </table>
           </td>
        </tr>
        <tr>
           <td><br><h4>Tracks</h4>
              <table width="100%">
                 <tr bgcolor="silver">
                    <td>1-Post Election Jam I</td>
                    <td align="Right">19:35</td>
                 </tr>
                 <tr bgcolor="silver">
                    <td>2-Post Election Jam II</td>
                    <td align="Right">20:35</td>
                 </tr>
              </table>
           </td>
        </tr>
     </table><br></body>
</html>
```

The final representation in a web browser looks like this:

Blues House Jam

ProductNo: BGJ-47



Tracks

1-Post Election Jam I
2-Post Election Jam II

Rule-Based Transformation

With rule-based transformation, the main XSLT control elements are templates (`<xsl:template>`). A template consists of a head and a body. The head of each template specifies the context in which the template should be activated. This is done by specifying an attribute `match` with an XPath expression to select the relevant context nodes.

The template body describes what to do. This can be procedural XSLT instructions (see above). In addition, we may apply recursion with the instruction `xsl:apply-templates`, which applies all templates defined in the stylesheet to all nodes in the selected context.

The `select` attribute of `xsl:apply-templates` defines the context in which the templates are to be executed. `select="."` stands for the current context: the processor will try to match templates with the child elements of the current node.

In addition, `xsl:apply-templates` has an optional `mode` attribute. This introduces an additional selection mechanism for templates: only those templates that have a matching mode attribute in their head are applied.

The result of an `xsl:apply-templates` instruction can be sorted with an `xsl:sort` instruction. In addition, the results can be numbered with the `xsl:number` instruction.

If the heads of more than one template match a certain context, the template with the best match is selected for execution:

- Templates in the current style sheet are selected over templates from imported style sheets.
- The more specific a matching expression in the template head is, the better is the match.
- In addition, it is possible to specify an explicit priority for a template.

Here is an example rule-based stylesheet that produces the same output as the previous procedural style sheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
<!-- Make sure we generate HTML output -->
<xsl:output method="html" indent="yes"/>
<!-- The root node does the basic setup -->
<xsl:template match="/">
<!-- Generate HTML document root -->
<html><head/><body>
  <!-- Process all children of the root node -->
  <xsl:apply-templates select="album"/>
  <!-- Second pass for tracks -->
  <h4>Tracks</h4>
```

```
  <!-- Mode parameter allows to select templates -->
  <xsl:apply-templates select="album/track" mode="tracks"/>
</body></html>
</xsl:template>

<!-- Template for title -->
<xsl:template match="title">
  <h2><xsl:value-of select="."/></h2><br/>
</xsl:template>

<!-- Template for publisher -->
<xsl:template match="publisher">
  Publisher:
  <xsl:value-of select="."/><br/>
</xsl:template>

<!-- Template for albumNo -->
<xsl:template match="@albumNo">
  ProductNo:
  <xsl:value-of select="."/><br/>
</xsl:template>

<!-- Template for coverImage -->
<xsl:template match="coverImage">
    <img src="{.}" alt="{../title}"/>
</xsl:template>

<!-- Template for special tracks processing -->
<xsl:template match="track" mode="tracks">
  <!-- Print character content of track element -->
  <xsl:number format="1-"/>
  <xsl:value-of select="title"/>
  <!-- Convert duration to mm:ss format -->
  (<xsl:value-of select=
        "substring-before(substring-after(duration,'T'),'M')"/>:
   <xsl:value-of select=
        "substring-before(substring-after(duration,'M'),'S')"/>)<br/>
</xsl:template>

<!-- Dummy template to exclude tracks from first pass -->
<xsl:template match="track">
</xsl:template>
```

```
</xsl:stylesheet>
```

This stylesheet contains a separate rule for each element in the source document. The consequence is that the layout of the resulting HTML page is not determined by the stylesheet but by the XML source. The sequence of elements in the XML source triggers the execution of rules in the stylesheet. Rule-based stylesheets are therefore best used when the output document must closely match the structure of the source document.

In our example, there is one exception: To create an extra paragraph with tracks (and title it with "Tracks") we used a two-pass approach. In the first pass we convert everything except `track` elements; in the second pass we convert only `track` elements. The appropriate templates are selected via `mode` attributes.

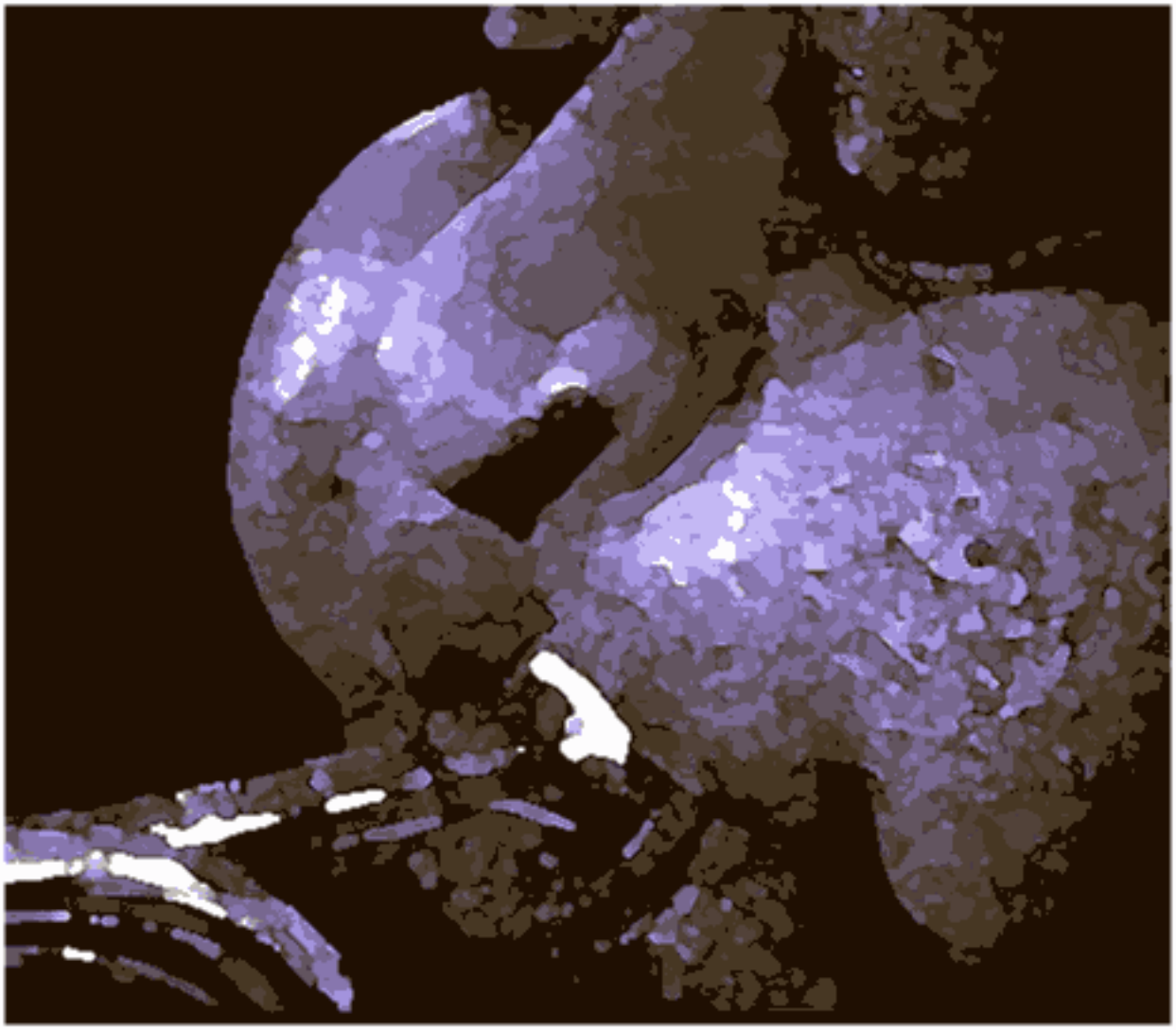
Here is the resulting HTML:

```
<html>
   <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=utf-8">
   </head>
   <body>
      ProductNo:
      BGJ-47<br>
      <h2>Blues House Jam</h2><br>
      <img src="post-election-jam.jpg" alt="Blues House Jam">
      <h4>Tracks</h4>
      1-Post Election Jam I (19:35)<br>
      2-Post Election Jam II (20:35)<br>
   </body>
</html>
```

And the result as it appears in the browser:

ProductNo: BGJ-47

Blues House Jam



Tracks

1-Post Election Jam I (19:35)
2-Post Election Jam II (20:35)

Limitations of XSLT

XSLT supports variables and parameters. However, XSLT variables are “read-only” variables: the value is assigned when the variable is defined and cannot be overwritten afterwards. Templates can specify formal parameters, too, so that it is possible to pass parameter values to templates. However, there is no way to return values to the caller. Basically, a template is stateless. XSLT is a functional language.

For programmers with a background in procedural programming this can make certain tasks difficult. Of course it is possible to mimic stateful behavior by making extensive use of recursive calls, but the stylesheets become hard to understand and execution requires a lot of memory.

In addition, XSLT does not have a complete set of built-in mathematical operators. For example, there are no trigonometric or logarithmic functions. This can be a disadvantage if, for example, we want to generate business graphics in SVG format. It is not impossible (one programmer succeeded in solving differential equations with XSLT!), but it is difficult.

Last but not least, the result of an XSLT style sheet transformation is always written to a single output stream. We cannot split output into several files (this issue is addressed in XSLT 1.1).

These limitations necessitate an extension mechanism, which XSLT fortunately provides. Several XSLT processors provide extensions, most notably Michael Kay's Saxon and the Apache Group's Xalan.

However, although the extension mechanism is standardized, the extensions themselves are not, so you have to choose a specific processor and stay with it. The good news is that there are community efforts to create a standard set of extensions: have a look at ***http://www.exslt.org/***.

Using Style Sheets with Tamino

There are several ways to apply stylesheets to an XML document. The common way is to supply a pointer to a stylesheet within a processing instruction of an XML document, for example:

```
<?xml-stylesheet type="text/xsl" href="album.xsl"?>
```

This processing instruction causes the XML processor to apply the stylesheet *album.xsl* to the content of the XML document.

In many cases, the XML client is a web browser. This is fine as long as we have control over which web browsers are used (for example, in an intranet) and can guarantee that all clients understand XSLT 1.0. But on the Internet we can be quite sure that not all clients (for example PDAs) can handle XSLT, so the conversion from XML to HTML must be done on the server.

Tamino's serialization method, in combination with the XSLT server extension, offers exactly this functionality. Using serialization, a server extension call can be included in a query. The XSLT server extension, as described in the chapter *Example: XSLT Server Extension* of the server extension documentation, makes XSLT transformations of XML documents retrieved from Tamino, using stylesheets that are stored in Tamino.

For storing stylesheets, we first define a small schema for the stylesheet document type:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs  = "http://www.w3.org/2001/XMLSchema"
  xmlns:tsd = "http://namespaces.softwareag.com/tamino/TaminoSchemaDefinition"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
  targetNamespace = "http://www.w3.org/1999/XSL/Transform" >
  <xs:annotation>
    <xs:appinfo>
      <tsd:schemaInfo name = "stylesheet">
        <tsd:collection name = "encyclopedia"/>
        <tsd:doctype name = "xsl:stylesheet">
          <tsd:logical>
            <tsd:content>closed</tsd:content>
          </tsd:logical>
        </tsd:doctype>
      </tsd:schemaInfo>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name = "stylesheet"/>
</xs:schema>
```

Note that we have defined a single untyped element with the name `stylesheet`. Accordingly, we have used the same name for the document type.

After we have defined this schema to Tamino, we can add stylesheets to our `encyclopedia` collection. To be able to identify these stylesheets later, we use the option to store a document instance under a particular document name (`@ino:docname`). This allows us to retrieve that document by its name via URL (see *From Schema to Tamino*::*Object Identity*).

The documentation for the *SerializationSpec* expression in the XQuery Reference Guide provides further information about the use of serialization.

22 Mapping a Schema to a Web Page

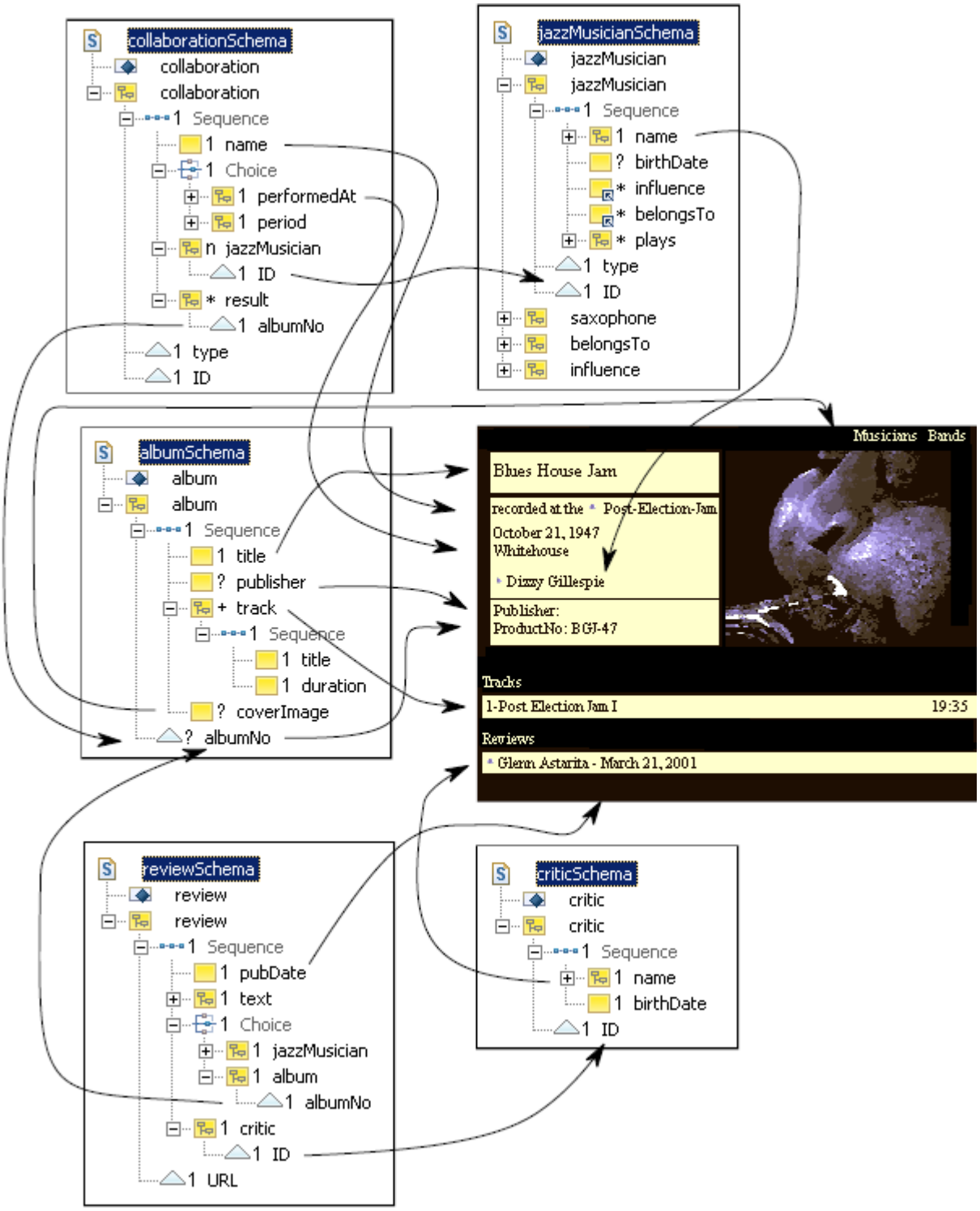
In many cases, prototypes of the planned web pages already exist and we want to "put the data into them". In these cases, the use of rule-based XSLT is hardly appropriate, and it is better to use XSLT in a procedural style.

The first step is to map the nodes from the XML schemas to the web page elements. During this step we can also make sure that all vital information is present in the web page.

Frequently, a web page does not exactly match an XML document type:

- The web page may not contain all the data of an XML document type;
- The web page may combine data from several XML document types, as shown in the following example.

Let us assume that a prototype as shown on the right already exists, and that we want to map the web page elements to XML nodes from the schemas shown on the left.



As far as the data from the `album` schema is concerned, the mapping from XML to HTML is straightforward. We simply put the whole HTML structure into an XSLT root template, and replace the example data elements with `xsl:value-of` instructions specifying the corresponding node in the `album` instance – a simple fill-in-the-blanks technique. In cases where an element has `minOccurs` and `maxOccurs` settings other than "1", we enclose it (and its decoration) in an `xsl:for-each` block.

Other HTML elements that depend on the existence of XML nodes are enclosed into an `xsl:if` block.

Original HTML Source Code

Here is the HTML source code for our prototype as generated by the HTML editor:

```
<html>
  <head>
    <meta http-equiv="Content-Type"
         content="text/html; charset=utf-8">
    <meta http-equiv="Content-Language" content="en-us">
    <meta name="GENERATOR" content="Microsoft FrontPage 4.0">
    <meta name="ProgId" content="FrontPage.Editor.Document">
    <title></title>
   </head>
```

```
   <body bgcolor="#000000" link="#FFFFCC" vlink="#C0C0C0" >
     <table>
       <tr font color="#FFFFCC">
         <td colspan="2" align="right">
           <a href="???">Musicians</a>
           <a href="???">Bands</a>
         </td>
       </tr>
       <tr>
         <td>
           <table>
             <tr bgcolor="silver">
               <td bgcolor="#FFFFCC">
                 <h2>Blues House Jam</h2>
               </td>
               <td rowspan="4">
                 <img src="post-election-jam.jpg"
                      alt="Blues House Jam">
               </td>
             </tr>
             <tr>
               <td bgcolor="#FFFFCC" valign="top">
                 <p>recorded at the <a href="???">
                 <img border="0" src="arrow.gif">
                 </a>Post-Election-Jam</p>
                 <p>October 21, 1947<br>
                 Whitehouse</p>
                 <p><a href="???"><img border="0" src="arrow.gif"></a>
                 Dizzy Gillespie<br>
               </td>
             </tr>
             <tr>
               <td bgcolor="#FFFFCC">
                 Publisher:<br>
```

```
                ProductNo: BGJ-47
              </td>
            </tr>
          </table>
        </td>
      </tr>
      <tr>
        <td><h3><br>
          <font color="#C0C0C0">Tracks</font></h3>
          <table width="100%">
            <tr>
              <td bgcolor="#FFFFCC">1-Post Election Jam I</td>
              <td align="Right" bgcolor="#FFFFCC">19:35</td>
            </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td><h3><br>
        <font color="#C0C0C0">Reviews</font></h3>
        <table width="100%">
          <tr>
            <td bgcolor="#FFFFCC">
              <a href="???"><img border="0" src="arrow.gif">
              </a>Glenn Astarita -
            March 21, 2001</td>
          </tr>
        </table>
      </td>
    </tr>
  </table><br>
 </body>
</html>
```

Adapting to XHTML

However, most HTML editors produce HTML which, in contrast to XHTML, is not well-formed XML. This requires a few fixes to the source code. Empty elements such as `<br>` and `<img ... >` must be properly closed with a slash (`<br/>` and `<img ... />`). Also, an entity reference such as ` ` must be replaced by a valid XML entity (` `) or by `<xsl:text> </xsl:text>`.

Retrieving XML Data from Tamino

The stylesheet that we create from the HTML prototype must also contain logic to join the `album` data with `collaboration` and `jazzMusician` data. We do this with the help of the XPath function `document()`, as explained in the chapter *From Conceptual Model to Schema*::*Constraints Across Documents*. We read the corresponding `collaboration` and `jazzMusician` documents into XSLT variables. The following code extracts the information that we want to display on the web page, such as the name of a collaboration, the date and location of a performance (when the collaboration is

a jam session), etc. From `jazzMusician` documents we extract the first name, middle name, and last name.

Generating Navigation

This data is also used to generate link URLs to the corresponding `collaboration` and `album` documents. When the user clicks on these links, the user leaves the `album` web page and moves to a `collaboration` or `jazzMusician` web page. These web pages are also dynamically generated: the link consists of a URL to Tamino with a query part identifying the respective document in the database, plus an appropriate stylesheet to be processed by Tamino's pass-thru servlet.

The Resulting Stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
<xsl:output method="html" indent="yes"/>
<!-- define a constant for the tamino query string -->
<xsl:variable name="query">
     ↩
http://localhost/servlets/com.softwareag.tamino.api.servlet.TaminoFilter/tamino/jazz/encyclopedia?_XQL=
</xsl:variable>
<!-- define constant for pass-thru string -->
<xsl:variable name="sheet">&_xslsrc=xsl:stylesheet/</xsl:variable>
<!-- define constant for encyclopedia collection -->
<xsl:variable
    name="ency">http://localhost/tamino/jazz/encyclopedia</xsl:variable>
<!-- Just a single rule for the root node -->
<xsl:template match="/">
<!-- Generate HTML document root -->
<html>
  <head/>
   <body bgcolor="#000000" link="#FFFFCC" vlink="#C0C0C0" >
     <!-- Top level loop. Remember that we get raw Tamino output. -->
     <xsl:for-each select="//album">
     <!-- Perform a join for the collaboration -->
     <xsl:variable name="c_query" select=
     "concat($query,'collaboration[result/@albumNo="',@albumNo,'"]')"/>
     <xsl:variable name="collab"
                   select="document($c_query)//collaboration"/>
     <table>
       <tr>
         <td colspan="2" align="right">
           <font color="#FFFFCC">
             <a href=
   "{concat($query,'jazzMusician',$sheet,'jazzMusician-index.xsl')}"
               Musicians</a>
             <a href=
"{concat($query,'collaboration[@type="band"]',$sheet,'band-index.xsl')}"
                Bands</a>
```

```
          </font>
        </td>
      </tr>
      <tr>
        <td>
          <table>
            <tr bgcolor="silver">
              <td bgcolor="#FFFFCC">
                <h2><xsl:value-of select="title"/></h2>
              </td>
              <xsl:if test="coverImage">
                <td rowspan="4">
                  <img src="{concat($ency,'/images/',coverImage)}"
                       alt="{title}"/>
                </td>
              </xsl:if>
            </tr>
            <tr>
              <td bgcolor="#FFFFCC" valign="top">
                <xsl:choose>
                  <!-- special treatment for jam sessions -->
                  <xsl:when test="$collab/@type='jamSession'">
                    <p>recorded at the
                       <xsl:value-of select="$collab/name"/></p>
                    <p>
                    <!-- call the template for date formatting -->
                    <xsl:call-template name="format-date">
                      <xsl:with-param name="date"
                                 select="$collab/performedAt/time"/>
                    </xsl:call-template>
                    <br/>
                    <xsl:value-of select=
                                  "$collab/performedAt/location"/>
                    </p>
                  </xsl:when>
                  <!-- otherwise print the band/project name -->
                  <xsl:otherwise>
                  <h4>
                    <!-- link includes specification of style sheet -->
                    <a href=
                       "{concat($c_query,$sheet,'collaboration.xsl')}">
                      <img border="0"
                           src="{concat($ency,'/images/arrow.gif')}"/>
                    </a>
                    <xsl:value-of select="$collab/name"/>
                  </h4>
                  </xsl:otherwise>
                </xsl:choose>
                <p>
                <!-- Loop over all collaborateurs -->
                <xsl:for-each select="$collab/jazzMusician">
                  <!-- Perform the join for the jazzMusicians -->
```

```
                <xsl:variable name="m_query" select=
      "concat($query,'jazzMusician[@ID="',@ID,'"]')"/>
                <xsl:variable name="musician" select=
                              "document($m_query)//jazzMusician"/>
                <!-- Create link to jazz musician web page -->
                <a href=
                    "{concat($m_query,$sheet,'jazzMusician.xsl')}">
                  <img border="0"
                       src="{concat($ency,'/images/arrow.gif')}"/>
                </a>
                <xsl:text> </xsl:text>
                <xsl:value-of select="$musician/name/first"/>
                <xsl:text> </xsl:text>
                <xsl:value-of select="$musician/name/middle"/>
                <xsl:value-of select="$musician/name/last"/>
                <br/>
              </xsl:for-each>
              </p>
            </td>
          </tr>
          <tr>
            <td bgcolor="#FFFFCC">
              <xsl:if test="publisher">
                Publisher: <xsl:value-of
                                select="publisher"/><br/>
              </xsl:if>
              ProductNo: <xsl:value-of select="@albumNo"/>
            </td>
          </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td><h3><br/>
        <font color="#C0C0C0">Tracks</font></h3>
        <table width="100%">
          <xsl:for-each select="track">
            <tr>
              <td bgcolor="#FFFFCC">
                <!-- Print track number -->
                <xsl:number value="position()" format="1-"/>
                <!-- Print character content of track element -->
                <xsl:value-of select="title"/>
              </td>
              <td align="Right" bgcolor="#FFFFCC">
                <xsl:value-of select=
                     "substring-before(substring-after(duration,'T'),'M')"/>:
                <xsl:value-of select=
                     "substring-before(substring-after(duration,'M'),'S')"/>
              </td>
            </tr>
          </xsl:for-each>
```

```
        </table>
      </td>
    </tr>
    <!-- Perform the join for the reviews -->
    <xsl:variable name="r_query" select=
      "concat($query,'review[album/@albumNo="',@albumNo,'"]')"/>
    <xsl:variable name="reviews" select="document($r_query)//review"/>
    <xsl:if test="$reviews">
    <tr>
      <td><h3><br/>
        <font color="#C0C0C0">Reviews</font></h3>
        <table width="100%">
          <xsl:for-each select="$reviews">
            <tr>
              <td bgcolor="#FFFFCC">
                <!-- Create link to review web page -->
                <a href="{@URL}">
                  <img border="0"
                     src="{concat($ency,'/images/arrow.gif')}"/>
                </a>
      <!-- Perform the join for the critic -->
      <xsl:variable name="cr_query" select=
         "concat($query,'critic[@ID="',critic/@ID,'"]')"/>
     <xsl:variable name="critic" select="document($cr_query)//critic"/>
  <xsl:value-of select="$critic/name/first"/>
               <xsl:value-of select="$critic/name/last"/> -
               <xsl:call-template name="format-date">
                  <xsl:with-param name="date" select="pubDate"/>
               </xsl:call-template>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </td>
    </tr>
    </xsl:if>
   </table><br/>
   </xsl:for-each>
  </body>
</html>
</xsl:template>
<!-- Date formatting -->
<xsl:template name="format-date">
  <xsl:param name="date"/>
  <!-- Get month and convert into name -->
  <xsl:variable name="month" select="substring($date,6,2)"/>
  <xsl:choose>
    <xsl:when test="$month=1">January</xsl:when>
    <xsl:when test="$month=2">February</xsl:when>
    <xsl:when test="$month=3">March</xsl:when>
    <xsl:when test="$month=4">April</xsl:when>
    <xsl:when test="$month=5">May</xsl:when>
```

```
    <xsl:when test="$month=6">June</xsl:when>
    <xsl:when test="$month=7">July</xsl:when>
    <xsl:when test="$month=8">August</xsl:when>
    <xsl:when test="$month=9">September</xsl:when>
    <xsl:when test="$month=10">October</xsl:when>
    <xsl:when test="$month=11">November</xsl:when>
    <xsl:otherwise>December</xsl:otherwise>
  </xsl:choose>
  <xsl:text> </xsl:text>
  <!-- Get day -->
  <xsl:value-of select="substring($date,9,2)"/>,
  <!-- Get year -->
  <xsl:value-of select="substring($date,1,4)"/>
</xsl:template>
</xsl:stylesheet>
```

In this example we see also some advanced formatting. Because XSLT was designed before XML Schema, it is not aware of XML Schema's built-in datatypes. Consequently, there are no easy-to-use formatting routines for these datatypes. This means that if we want to display, for example, a date in any format other than the standard ISO format, we have to write the necessary formatting routine ourselves. This is done in the template `format-date`.

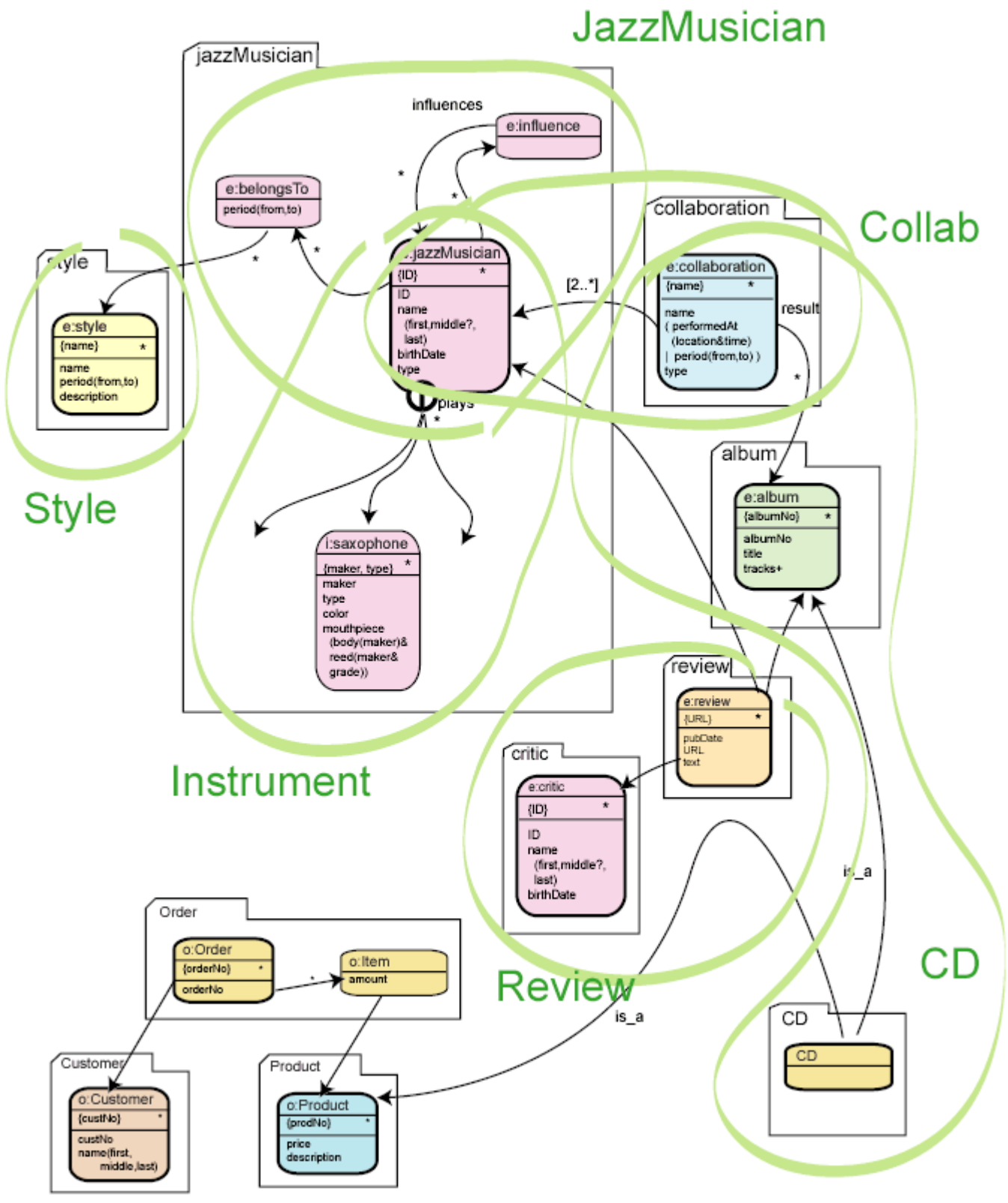
23 Navigation with XLink

- Defining Navigational Objects 176
- Defining Navigational Links 178

The approach shown above, i.e. starting with an existing HTML page and constructing a style sheet from that, works fine for small applications. For a large application, however, we would easily lose the overview over all the stylesheets, transformations, hyperlinks and other connections between the various resources.

Defining Navigational Objects

In such cases, it is necessary to adopt a more systematic approach both for the presentation design and for the navigation model. A good basis for such an approach is the conceptual data model. In the chapter *From Conceptual Model to Schema*::*From Model to Schema* we had already discussed how XML schemas can be derived from a conceptual model. Here we see how the conceptual model can guide us through web page and navigation design. We use the same multi-namespace model that we developed in the section *From Conceptual Model to Schema*:: *Models and Namespaces*



The first step is to partition the conceptual model into navigational objects (such as web pages). In many cases the relationship between business objects and navigational objects will not be a 1:1

relationship. There will be cases where a single business object needs to be partitioned into several related navigational objects, simply because of its size. In such a case we should identify one navigational object as the main navigational object for this type of business object. It should be possible to reach the other partitions from this main object via hyperlinks.

In our jazz example we have partitioned the business object `jazzMusician` into two navigational objects: `jazzMusician`, which is the main navigational object for this business object type, and `instrument`, which informs the end user about the instruments played by a particular musician.

In other cases, information from one business object must be augmented with information from other business objects. For example, in the case of our `album` business object we might want to display `collaboration` information such as participating artists, time and location of a live performance, etc. along with the album data, as we have already shown in the section ***Mapping a schema to a web page***, i.e. we aggregate data from several business objects into one navigation object.

In addition to the navigational objects that relate to conceptual business objects, we might also offer additional access structures:

- *Indices* that collate information from the instances of a given business object type. For example, we might have a web page that contains an alphabetical index of all jazz musicians, and another that contains an index of all styles. The individual index entries would lead to the main navigational objects of the respective business objects.
- *Tables of contents* and *site maps* that provide structured overviews over the web or subsets of it.
- A *synopsis* that provides an informal overview of the web.
- *Guided tours* that lead the user through the most important areas.
- *Landmarks* that act as entry points for sub-areas or for special functionality of a web. These landmarks appear in a consistent form on each web page.
- *Portals* that act as entry points for the whole web.

Each of these navigational objects can be represented by a combination of a Tamino query expression and a stylesheet as shown above in the section ***Using Style Sheets with Tamino***.

Defining Navigational Links

Navigational objects are connected via navigational links. Depending on its type, a navigational link may be traversed by the end user either in one direction (unidirectional) or in both directions (bidirectional). Navigational links are based on the links between conceptual objects but, again, there is no 1:1 relationship between a conceptual link and a navigational link. In many cases we want to complement the existing conceptual links with additional access paths that act as shortcuts. A typical example is shown in the diagram above, where we have introduced a shortcut from `jazzMusician` to `album`. Shortcuts can be derived from conceptual links by combination. In the

diagram above, the shortcut is derived from a combination of two conceptual links: `collaboration->jazzMusician` and `collaboration->album`.

In terms of modeling, a navigational model is a *view* of a conceptual model. A view may only reveal certain aspects of a conceptual model, but it may also introduce new, derived items. In fact, a single conceptual model may have multiple views, for example, to cater for different groups of users. Views are also subject to more frequent changes than conceptual models: the analysis of user behavior often suggests changes to the navigational structures of a web.

In the discipline of implementing hypertext systems (and webs are hypertext systems), this situation has led to the practice of implementing navigational links as first class objects in a separate layer. By removing navigational structures from the presentation layer, the maintenance of both the presentation layer and the navigational layer becomes easier. Changing the design of a web, for example, does not affect the navigation structures, while re-routing navigation paths does not require updating the presentation logic of all web pages.

How would we store such a first class link object in Tamino?

One possibility would be to store a separate document for each link. However, this would cause a high frequency of read accesses to Tamino and would be detrimental to performance. Therefore we combine several links into a *linkbase*. Linkbases are not our own invention but are a concept that was formulated in the context of XLink. XLink provides also most of the syntactic means that are required to describe independent navigational structures.

Introducing XLink

Conceptually, XLink builds on a network of nodes and arcs. Navigational objects act as nodes, and the arcs specify the pairs of navigational objects between which transitions are possible and the direction in which transitions are allowed. In our scenario, however, we are not really interested in describing navigational transitions between individual navigational objects but rather between sets of these objects. Therefore, we have to extend XLink. In the rest of this section we describe how we do this. At the same time we introduce the basic concepts of XLink.

To define a linkbase we create a linkbase document. Within each linkbase we define one or several extended links. Typically, we would define a separate extended link for each user type:

```
<linkbase xmlns:xlink="http://www.w3.org/1999/xlink">
  <encyclopediaLinks xlink:type="extended" user="jazzfan">
  ...
  </encyclopediaLinks>
  ...
</linkbase>
```

XLink can be used to define within each *extended link* an arbitrary number of nodes and arcs. To describe the nodes (in our case, sets of navigational objects) we use XLink *locators*. Usually, an XLink locator describes a foreign web resource. In our case, however, we want to describe a virtual

navigational object that is generated at runtime from a Tamino query and an XSL stylesheet. Therefore, we extend the definition of the locator element with an `xsl:stylesheet` attribute:

```
<musician xlink:type="locator"
  xlink:href=
   "http://localhost/tamino/jazz/encyclopedia?_XQL=e:jazzMusician"
  xsl:stylesheet="xsl:stylesheet/jazzMusician.xsl"
  xlink:label="mus"/>
```

In this example, the locator points to a set of `jazzMusician` instances stored in Tamino. The set of virtual navigational objects results from the application of the specified stylesheet to the query result. The final attribute (`xlink:label`) identifies this locator for the following *arc* definition:

```
<collaborationToJazzMusician xlink:type="arc"
  xlink:from="col"
  xlink:to="mus"
  xqlFilter="[e:jazzMusician/@ID='$keyvalue']"/>
```

This defines a possible transition between two locators which are identified by their `xlink:label` attributes, as specified in `xlink:from` and `xlink:to`. The attribute `xlink:title` can optionally be used to decorate the resulting hyperlink.

Here, we have extended the standard XLink mechanism with an `xqlFilter` attribute. This attribute describes how specific instances of the navigational target object are selected. In the filter expression we use `$keyvalue` as a placeholder for a key value. We see later how this expression can be evaluated within a stylesheet.

The Navigational Model

In the following example, we define a locator for each navigational object that represents a document type such as `jazzMusician`, `style`, `collaboration`, or `album`. In addition, we define a navigational object `instruments` that relies on `jazzMusician` but focuses on the instruments played by this musician. Also, we define locators that represent indexes such as `bandIndex` and `musicianIndex`:

```
<?xml version="1.0"?>
<linkbase
      xmlns:xlink="http://www.w3.org/1999/xlink"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns="http://www.softwareag.com/tamino/doc/examples/models/jazz/shop"
     xmlns:e="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
      xmlns:i="http://www.softwareag.com/tamino/doc/examples/models/instruments"
>
<encyclopediaLinks xlink:type="extended" user="jazzfan">
  <!-- Locators -->
  <CDs xlink:type="locator"
    xlink:href="http://localhost/tamino/jazz/shop?_XQL=CD"
    xsl:stylesheet="xsl:stylesheet/cd.xsl"
    xlink:label="cd"/>
  <collaborations xlink:type="locator"
```

```
    xlink:href=
      "http://localhost/tamino/jazz/encyclopedia?_XQL=e:collaboration"
    xsl:stylesheet="xsl:stylesheet/collaboration.xsl"
    xlink:label="col"/>
  <bandIndex xlink:type="locator"
    xlink:href=
     "http://localhost/tamino/jazz/encyclopedia?_XQL=e:collaboration
                                              [@type='band']%20sortall%20(./name)"
    xsl:stylesheet="xsl:stylesheet/band-index.xsl"
    xlink:label="bnd-ix"/>
  <musicians xlink:type="locator"
   xlink:href=
     "http://localhost/tamino/jazz/encyclopedia?_XQL=e:jazzMusician"
    xsl:stylesheet="xsl:stylesheet/jazzMusician.xsl"
    xlink:label="mus"/>
  <musicianIndex xlink:type="locator"
    xlink:href=
"http://localhost/tamino/jazz/encyclopedia?_XQL=e:jazzMusician%20sortall20(./@ID)"
    xsl:stylesheet="xsl:stylesheet/jazzMusician-index.xsl"
    xlink:label="mus-ix"/>
  <instruments xlink:type="locator"
    xlink:href=
      "http://localhost/tamino/jazz/encyclopedia?_XQL=e:jazzMusician"
    xsl:stylesheet="xsl:stylesheet/instrument.xsl"
    xlink:label="instr"/>
  <styles
    xlink:type="locator"
    xlink:href="http://localhost/tamino/jazz/encyclopedia?_XQL=e:style"
    xsl:stylesheet="xsl:stylesheet/style.xsl"
    xlink:label="sty"/>
  <reviews
    xlink:type="locator"
    xlink:href="http://localhost/tamino/jazz/encyclopedia?_XQL=e:review"
    xsl:stylesheet="xsl:stylesheet/review.xsl"
    xlink:label="rev"/>
<!-- Arcs -->
  <cdToCollaboration xlink:type="arc"
    xlink:from="cd"
    xlink:to="col"
    xqlFilter="[@albumNo='$keyvalue']"/>
  <collaborationToJazzMusician xlink:type="arc"
    xlink:from="col"
    xlink:to="mus"
    xqlFilter="[e:jazzMusician/@ID='$keyvalue']"/>
  <jazzMusicianToInstrument xlink:type="arc"
    xlink:from="mus"
    xlink:to="instr"
    xqlFilter="[@ID='$keyvalue']"/>
  <jazzMusicianToCDs xlink:type="arc"
    xlink:from="mus"
    xlink:to="cd"
    xqlFilter=
```

```
"[@albumNo=xqx.qdoc('encyclopedia/e:collaboration
   [e:jazzMusician/@ID=&quot;$keyvalue&quot;]')/e:result/@albumNo]"/>
  <cdToReview xlink:type="arc"
    xlink:from="cd"
    xlink:to="rev"
    xqlFilter="[@albumNo='$keyvalue']"/>
  <jazzMusicianToReview xlink:type="arc"
    xlink:from="mus"
    xlink:to="rev"
    xqlFilter="[@ID='$keyvalue']"/>
  <ToBandIndex xlink:type="arc"
    xlink:to="bnd-ix"
    xlink:title="Bands"/>
  <ToMusicianIndex xlink:type="arc"
    xlink:to="mus-ix"
    xlink:title="Musicians"/>
</encyclopediaLinks>
</linkbase>
```

Note that we have used the X-Query operator `sortall` for the index locators. Because these expressions appear in the query part of a URL, we must express the blank character with the code `%20`.

Among the arcs, we have also defined two that lead to the index locators. We want to include hyperlinks to these indices on every web page that we generate. Because there is no specific `from` node, we just omit the `xlink:from` attribute for these arcs.

The arc `jazzMusicianToAlbums` describes a derived link. We use the function `xqx.qdoc` (which we developed in *Utilizing Server Extensions*:: *qdoc*) to establish a link from `jazzMusician` via `collaboration` to `album`.

Using the Link Base

To interpret such a link base within an XSL stylesheet, we must first load the linkbase from Tamino. Let us assume that the linkbase has been stored in Tamino under document type `linkbase` in our collection `encyclopedia`. Then we can load the relevant extended link with:

```
<xsl:variable name="linkbase" select=
"document('http://localhost/tamino/jazz/encyclopedia?_XQL=linkbase//encyclopediaLinks
                              [@user=&quot;jazzfan&quot;]')//encyclopediaLinks"/>
```

To generate individual hyperlinks we use the following template. Because the generation is a bit lengthy, we pack it into a separate template that can be invoked with `<call-template name="gen-link"/>`.

```
<!-- Link generation -->
<xsl:template name="gen-link"
      xmlns:xql="http://metalab.unc.edu/xql/"
      xmlns:xlink="http://www.w3.org/1999/xlink">
<!-- selected arc -->
  <xsl:param name="arc"/>
<!-- key value for xql filter expression
     if omitted we do not generate a filter expression -->
  <xsl:param name="keyvalue"/>
<!-- Title: if not defined use title of arc as default -->
  <xsl:param name="title" select="$arc/@xlink:title"/>
<!-- generate link only if arc is present -->
  <xsl:if test="$arc">
<!-- select target locator -->
    <xsl:variable name="to" select=
    "$arc/../*[@xlink:type='locator' and @xlink:label=$arc/@xlink:to]"/>
    <a>
<!-- generate href -->
      <xsl:attribute name="href" >
<!-- this is the target document -->
        <xsl:value-of select="$to/@xlink:href"/>
<!-- this is the filter -->
<!-- generate only if $keyvalue is supplied and a filter expression is present -->
        <xsl:if test="$keyvalue and $arc/@xqlFilter">
          <xsl:value-of select=
                "substring-before($arc/@xqlFilter,'$keyvalue')"/>
<!-- replace placeholder with actual value -->
          <xsl:value-of select="$keyvalue"/>
          <xsl:value-of select=
                "substring-after($arc/@xqlFilter,'$keyvalue')"/>
        </xsl:if>
<!-- this is the stylesheet -->
        <xsl:if test="$to/@xsl:stylesheet">
          <xsl:text>&amp;_xslsrc=</xsl:text>
          <xsl:value-of select="$to/@xsl:stylesheet"/>
        </xsl:if>
      </xsl:attribute>
<!-- decorate with title -->
      <xsl:choose>
        <xsl:when test="$title">
          <xsl:value-of select="$title"/>
        </xsl:when>
        <xsl:otherwise>
<!-- default decoration -->
          <img border="0" src="{concat($ency,'/images/arrow.gif')}"/>
        </xsl:otherwise>
      </xsl:choose>
    </a>
  </xsl:if>
</xsl:template>
```

This template takes three parameters:

- `arc` identifies the arc to be generated. If this arc does not exist in the linkbase, no link is generated.
- `keyvalue` is the key value that identifies the particular instance of the navigation object. If this parameter is not specified, no filter expression is generated.
- `title` is a locally defined decoration for the link. If this parameter is not specified, the title defined in the arc is used instead. If no title is defined there either, some default decoration is generated.

A typical invocation of this template looks like this:

```
<xsl:call-template name="gen-link">
  <xsl:with-param name="arc"
                  select="$linkbase/jazzMusicianToAlbums"/>
  <xsl:with-param name="keyvalue" select="@ID"/>
  <xsl:with-param name="title">Albums</xsl:with-param>
</xsl:call-template>
```

Using this template call, the final stylesheet for our `album` web page looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
     xmlns:e="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
      xmlns:i="http://www.softwareag.com/tamino/doc/examples/models/instruments"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
<xsl:output method="html" indent="yes"/>
<!-- define a constant for the tamino query string -->
<xsl:variable name="query">
  ↩
http://localhost/servlets/com.softwareag.tamino.api.servlet.TaminoFilter/tamino/jazz/encyclopedia?_XQL=
</xsl:variable>
<!-- define constant for pass-thru string -->
<xsl:variable name="sheet">&_xslsrc=xsl:stylesheet/</xsl:variable>
<!-- define constant for encyclopedia collection -->
<xsl:variable
    name="ency">http://localhost/tamino/jazz/encyclopedia</xsl:variable>
<!-- Just a single rule for the root node -->
<xsl:template match="/">
<!-- Generate HTML document root -->
<html>
  <head/>
   <body bgcolor="#000000" link="#FFFFCC" vlink="#C0C0C0" >
     <!-- Top level loop. Remember that we get raw Tamino output. -->
     <xsl:for-each select="//e:album">
     <!-- Perform a join for the collaboration -->
     <xsl:variable name="c_query" select=
     "concat($query,'e:collaboration[e:result/@albumNo="',@albumNo,'"]')"/>
     <xsl:variable name="collab"
                   select="document($c_query)//e:collaboration"/>
     <table>
       <tr>
```

```
        <td colspan="2" align="right">
          <font color="#FFFFCC">

            <xsl:call-template name="gen-link">
              <xsl:with-param name="arc"
                   select="$linkbase/ToMusicianIndex"/>
            </xsl:call-template>

            <xsl:text> </xsl:text>

            <xsl:call-template name="gen-link">
              <xsl:with-param name="arc"
                   select="$linkbase/ToBandIndex"/>
            </xsl:call-template>

           </font>
        </td>
      </tr>
      <tr>
        <td>
          <table>
            <tr bgcolor="silver">
              <td bgcolor="#FFFFCC">
                <h2><xsl:value-of select="e:title"/></h2>
              </td>
              <xsl:if test="coverImage">
                <td rowspan="4">
                  <img src="{concat($ency,'/images/',coverImage)}"
                       alt="{title}"/>
                </td>
              </xsl:if>
            </tr>
            <tr>
              <td bgcolor="#FFFFCC" valign="top">
                <xsl:choose>
                  <!-- special treatment for jam sessions -->
                  <xsl:when test="$collab/@type='jamSession'">
                    <p>recorded at the
                       <xsl:value-of select="$collab/e:name"/></p>
                    <p>
                    <!-- call the template for date formatting -->
                    <xsl:call-template name="format-date">
                      <xsl:with-param name="date"
                                 select="$collab/e:performedAt/e:time"/>
                    </xsl:call-template>
                    <br/>
                    <xsl:value-of select=
                                  "$collab/e:performedAt/e:location"/>
                    </p>
                  </xsl:when>
                  <!-- otherwise print the band/project name -->
                  <xsl:otherwise>
```

```
                <h4>

                   <xsl:call-template name="gen-link">
                    <xsl:with-param name="arc"
                         select="$linkbase/albumToCollaboration"/>
                    <xsl:with-param name="keyvalue"
                                    select="@albumNo"/>
                  </xsl:call-template>

                   <xsl:value-of select="$collab/e:name"/>
                </h4>
                </xsl:otherwise>
              </xsl:choose>
              <p>
              <!-- Loop over all collaborateurs -->
              <xsl:for-each select="$collab/e:jazzMusician">
                <!-- Create link to jazz musician web page -->

                <xsl:call-template name="gen-link">
                  <xsl:with-param name="arc"
                   select="$linkbase/collaborationToJazzMusician"/>
                  <xsl:with-param name="keyvalue" select="@ID"/>
                </xsl:call-template>

                <!-- Perform the join for the jazzMusicians -->
                <xsl:variable name="m_query" select=
    "concat($query,'e:jazzMusician[@ID="',@ID,'"]')"/>
                <xsl:variable name="musician" select=
                              "document($m_query)//e:jazzMusician"/>
                <!-- Create link to jazz musician web page -->
                <a href=
                    "{concat($m_query,$sheet,'jazzMusician.xsl')}">
                  <img border="0"
                       src="{concat($ency,'/images/arrow.gif')}"/>
                </a>
                <xsl:text> </xsl:text>
                <xsl:value-of select="$musician/e:name/e:first"/>
                <xsl:text> </xsl:text>
                <xsl:value-of select="$musician/e:name/e:middle"/>
                <xsl:value-of select="$musician/e:name/e:last"/>
                <br/>
              </xsl:for-each>
              </p>
            </td>
          </tr>
          <tr>
            <td bgcolor="#FFFFCC">
              <xsl:if test="publisher">
                Publisher: <xsl:value-of
                                select="e:publisher"/><br/>
              </xsl:if>
              ProductNo: <xsl:value-of select="@albumNo"/>
```

```
            </td>
          </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td><h3><br/>
        <font color="#COCOCO">Tracks</font></h3>
        <table width="100%">
          <xsl:for-each select="e:track">
            <tr>
              <td bgcolor="#FFFFCC">
                <!-- Print track number -->
                <xsl:number value="position()" format="1-"/>
                <!-- Print character content of track element -->
                <xsl:value-of select="e:title"/>
              </td>
              <td align="Right" bgcolor="#FFFFCC">
                <xsl:value-of select=
                     "substring-before(substring-after(duration,'T'),'M')"/>:
                <xsl:value-of select=
                     "substring-before(substring-after(duration,'M'),'S')"/>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </td>
    </tr>
    <!-- Perform the join for the reviews -->
    <xsl:variable name="r_query" select=
      "concat($query,'review[e:album/@albumNo="',@albumNo,'"]')"/>
    <xsl:variable name="reviews" select="document($r_query)//e:review"/>
    <xsl:if test="$reviews">
    <tr>
      <td><h3><br/>
        <font color="#COCOCO">Reviews</font></h3>
        <table width="100%">
          <xsl:for-each select="$reviews">
            <tr>
              <td bgcolor="#FFFFCC">
               <!-- Create link to review web page -->

               <xsl:call-template name="gen-link">
                 <xsl:with-param name="arc"
                  select="$linkbase/albumToReview"/>
                 <xsl:with-param name="keyvalue" select="e:album/@albumNo"/>
               </xsl:call-template>

     <!-- Perform the join for the critic -->
      <xsl:variable name="cr_query" select=
         "concat($query,'e:critic[@ID="',e:critic/@ID,'"]')"/>
     <xsl:variable name="critic" select="document($cr_query)//e:critic"/>
```

```
  <xsl:value-of select="$critic/e:name/e:first"/>
               <xsl:value-of select="$critic/e:name/e:last"/> -
               <xsl:call-template name="format-date">
                  <xsl:with-param name="date" select="e:pubDate"/>
               </xsl:call-template>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </td>
    </tr>
    </xsl:if>
   </table><br/>
   </xsl:for-each>
  </body>
</html>
</xsl:template>
<!-- Date formatting -->
<xsl:template name="format-date">
  <xsl:param name="date"/>
  <!-- Get month and convert into name -->
  <xsl:variable name="month" select="substring($date,6,2)"/>
  <xsl:choose>
    <xsl:when test="$month=1">January</xsl:when>
    <xsl:when test="$month=2">February</xsl:when>
    <xsl:when test="$month=3">March</xsl:when>
    <xsl:when test="$month=4">April</xsl:when>
    <xsl:when test="$month=5">May</xsl:when>
    <xsl:when test="$month=6">June</xsl:when>
    <xsl:when test="$month=7">July</xsl:when>
    <xsl:when test="$month=8">August</xsl:when>
    <xsl:when test="$month=9">September</xsl:when>
    <xsl:when test="$month=10">October</xsl:when>
    <xsl:when test="$month=11">November</xsl:when>
    <xsl:otherwise>December</xsl:otherwise>
  </xsl:choose>
  <xsl:text> </xsl:text>
  <!-- Get day -->
  <xsl:value-of select="substring($date,9,2)"/>,
  <!-- Get year -->
  <xsl:value-of select="substring($date,1,4)"/>
</xsl:template>
<!-- Link generation -->
<xsl:template name="gen-link"
      xmlns:xql="http://metalab.unc.edu/xql/"
      xmlns:xlink="http://www.w3.org/1999/xlink">
<!-- selected arc -->
  <xsl:param name="arc"/>
<!-- key value for xql filter expression
     if omitted we do not generate a filter expression -->
  <xsl:param name="keyvalue"/>
<!-- Title: if not defined use title of arc as default -->
```

```
  <xsl:param name="title" select="$arc/@xlink:title"/>
<!-- generate link only if arc is present -->
  <xsl:if test="$arc">
<!-- select target locator -->
    <xsl:variable name="to" select=
    "$arc/../*[@xlink:type='locator' and @xlink:label=$arc/@xlink:to]"/>
    <a>
<!-- generate href -->
      <xsl:attribute name="href" >
<!-- this is the target document -->
        <xsl:value-of select="$to/@xlink:href"/>
<!-- this is the filter -->
        <xsl:if test="$keyvalue and $arc/@xqlFilter">
         <xsl:value-of
            select="substring-before($arc/@xqlFilter,'$keyvalue')"/>
         <xsl:value-of select="$keyvalue"/>
         <xsl:value-of
            select="substring-after($arc/@xqlFilter,'$keyvalue')"/>
        </xsl:if>
<!-- this is the stylesheet -->
        <xsl:if test="$to/@xsl:stylesheet">
  <xsl:text>&amp;_xslsrc=</xsl:text>
         <xsl:value-of select="$to/@xsl:stylesheet"/>
        </xsl:if>
      </xsl:attribute>
      <xsl:choose>
        <xsl:when test="$title">
          <xsl:value-of select="$title"/>
        </xsl:when>
        <xsl:otherwise>
          <img border="0" src="{concat($ency,'/images/arrow.gif')}"/>
        </xsl:otherwise>
      </xsl:choose>
    </a>
  </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

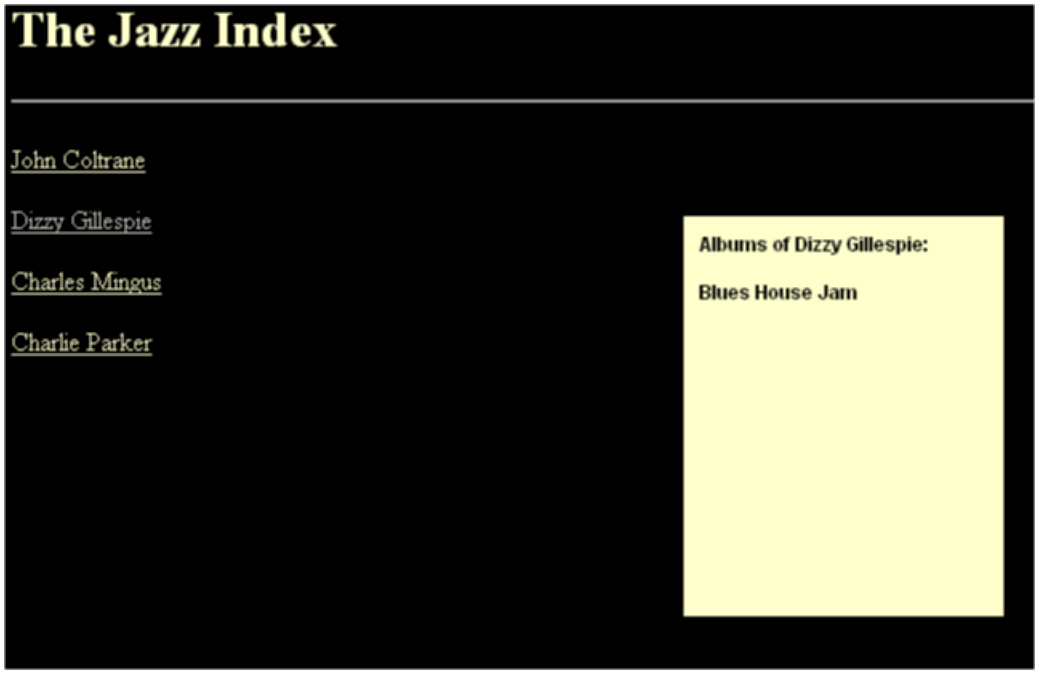
24 The Tamino JavaScript API

Although XSLT can generate all sorts of output from an original XML file, it does not provide a binding of the displayed elements to the original XML elements. When an XML document is converted into HTML, it is not possible to modify the HTML elements on the screen and automatically reflect the changes back to the original XML document. In fact, XSLT does not provide any facilities for processing end-user input. We have to employ other technology in order to do this.

One suitable technology is JavaScript. JScript, Microsoft's version of JavaScript, provides features that make it possible to access Tamino directly from the client's web page. The Tamino JavaScript API utilizes those features. Consequently, it can only be used for clients running on a Microsoft Windows platform with the Microsoft Internet Explorer.

Apart from this restriction, the Tamino JavaScript API offers lightweight access to all relevant Tamino functions. You can query, insert, update and delete objects and control transactional behavior directly from the client.

In the following example, we use the Tamino JavaScript API to make an interactive index web page. The stylesheet *jazzMusician-index.xsl* generates such a page that lists all the jazz musicians stored in the `encyclopedia` collection. When the user moves the mouse over these entries, the corresponding artist's albums are shown in a separate text area:



This is achieved by generating an event handler `onmouseover="javascript:displayAlbums()"` for each entry and passing the musician's ID to the function. The function first connects to Tamino, issues a query for a `collaboration` corresponding to the musician's ID, then uses the `result` element of each `collaboration` document to query for `album` documents. Finally, the title of each album is written into a special display area (implemented as `<div id="albumArea">`). Here is the complete code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
     xmlns="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="html" indent="yes"/>

<!-- define constant for tamino query string -->
<xsl:variable name="query">
http://localhost/servlets/com.softwareag.tamino.api.servlet.TaminoFilter/tamino/jazz/encyclopedia?_XQL=
</xsl:variable>
<xsl:variable
  name="ency">http://localhost/tamino/jazz/encyclopedia</xsl:variable>
```

```
<!-- define constant for pass-thru string -->
<xsl:variable name="sheet">&amp;_xslsrc=xsl:stylesheet/</xsl:variable>
```

```
<!-- Just a single rule for the root node -->
<xsl:template match="/">
<!-- Generate HTML document root -->
<html>
  <head>
    <SCRIPT LANGUAGE="JavaScript"
            SRC="{concat($ency,'/scripts/TaminoLib.js'}}"></SCRIPT>
    <script language="JavaScript"><![CDATA[
function displayAlbums(jmid,mname) {
// create title
 var aText = "Albums of "+mname+":<br>";
// construct database and collection name
// (just to have a single point of maintenance)
 var dbname=
]]>
<xsl:value-of select="concat('&quot;',$ency,'&quot;;')"/>
<![CDATA[
 // prepare query
 var QueryVal="e:collaboration[e:jazzMusician='"+jmid+"']";
 var pageSize=0;
 // create Tamino client
 var QueryObj = new TaminoClient(dbname, pageSize);
 // issue query
 var QueryResult = QueryObj.query(QueryVal);
 // strip off Tamino packaging
 var xqlResult=QueryResult.getResult();
 if (xqlResult) {
   // get collaboration nodes
   var collaborations=xqlResult.childNodes;
   // loop through each collaboration
   for (var i=0; i<collaborations.length; i++) {
     var collabSelected = collaborations.item(i);
     // get result node
   var title=albumSelected.getElementsByTagNameNS(
           "http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia",
                                                                    "title");
     // get album node
     var album=result.item(0).childNodes.item(0);
     // get albumNo
     var albumNo=album.getAttribute("albumNo");
          // construct query
     var QueryVal2="e:album[@albumNo='"+albumNo+"']";
     // issue query
     var QueryResult2 = QueryObj.query(QueryVal2);
     // strip off Tamino packaging
     var xqlResult2=QueryResult2.getResult();
     if (xqlResult2) {
       // get album nodes
       var albums=xqlResult2.childNodes;
```

```
       // loop through all album nodes
       for (var j=0; j<albums.length; j++) {
         var albumSelected = albums.item(j);
         // get title node
         var title=albumSelected.getElementsByTagNameNS(
               ↩
"http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia",
                                                                    ↩
"title");
   // add title node data to output string
         aText = aText+"<br>"+title.item(0).childNodes.item(0).data;
       }
     }
   }
 }
 // write output string into album area
 document.all.albumArea.innerHTML = aText;
}
  ]]></script>
```

```
  <title>The Jazz Index</title>
</head>
<body bgcolor="#000000" text="#FFFFCC" link="#FFFFCC" vlink="#C0C0C0">
  <div id="albumArea" style="position:absolute; top:150px; left:430px;
   width:200px; height:250px; background-color:#FFFFCC; color:#000000;
   font-family:Arial; font-size:9pt; font-weight:bold; padding:10px;">
    <layer id="lay1" bgcolor="#FFFFCC">
      Albums:
    </layer>
  </div>
  <h1>The Jazz Index</h1>
  <hr/>
  <xsl:for-each select="//e:jazzMusician">
    <xsl:variable name="mname">
      <xsl:value-of select="e:name/e:first"/>
        <xsl:if test="e:name/e:middle">
          <xsl:text> </xsl:text>
          <xsl:value-of select="e:name/e:middle"/>
        </xsl:if>
       <xsl:text> </xsl:text>
       <xsl:value-of select="e:name/e:last"/>
    </xsl:variable>
    <p>
      <a href=
"{concat($query,'e:jazzMusician[@ID=&quot;',@ID,'&quot;]',$sheet,'jazzMusician.xsl')}"
         onmouseover="javascript:displayAlbums('{@ID}','{$mname}')">

         <xsl:value-of select="$mname"/>
      </a>
    </p>
  </xsl:for-each>
```

```
</body>
</html>
```

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

25 XSLT Summary

In the previous sections, we have shown how XSLT can be used to create sophisticated webs from XML data stored in Tamino. However, we have also encountered some of the limitations of XSLT. Let us summarize the advantages and disadvantages of this technology:

Advantages

- XSLT processors are readily available on various platforms. Processors like XT, Saxon and XALAN are implemented in Java and run on any server. On Windows platforms, the MSXML DLL provides adequate support.
- An XSLT stylesheet can convert XML into almost any output format, from HTML over XHTML to WML, SVG, SMIL, etc. This allows support not only for HTML clients but also for mobile clients and multimedia applications.
- With the help of serialization and the XSLT server extension, XSLT stylesheets allow Tamino to deliver query results in customized formats.

Disadvantages

- Processing XSLT stylesheets consumes considerable CPU resources. Scalable applications often require a cache for transformation results in order to achieve the necessary throughput. Another throughput-enhancing option is the use of an XSLT compiler, for example the XSLT compiler contained in XALAN, which translates an XSLT stylesheet into a set of Java classes.
- XSLT does not provide a binding between the original XML document and the result of the transformation. Insert, update and delete operations require additional program logic.
- The XSLT standard has some limitations. Advanced transformations require the use of extensions (see section **Limitations of XSLT**).
- The XSLT syntax is hard to read, and rule-based programming is a concept alien to many programmers.

- WYSIWYG design of XSLT-generated HTML pages is possible with tools such as eXcelon's Stylus, Whitehill's XSL Composer and XML Spy Suite. However, such tools are not commonly used by most web designers.

26 Rapid Prototyping with XQuery 4

XQuery 4 is another alternative when building prototype web applications. In contrast to the XPath-like X-Query, XQuery allows complex output documents to be constructed within the query expression. Basically, XQuery combines the facilities of XPath and XSLT, but uses different syntax. Briefly, XQuery's advantages and disadvantages are as follows:

Advantages

- XQuery combines the processing power of XPath and XSLT into one consistent language.
- XQuery can convert XML into almost any output format, from HTML over XHTML to WML, SVG, SMIL, etc. This allows support not only for HTML clients but also for mobile clients, and for multimedia applications.
- User-definable XQuery functions allow for modular query expressions.
- The XML Schema type system and namespaces are fully supported.
- XQuery 4 features easy-to-read SQL-like syntax. An alternative syntax (XQueryX) allows the formulation of queries as XML documents.
- Processing can be embedded into the query. Extra stylesheets are not required. In contrast to W3C, XQuery 4 also supports insert, update and delete operations.

Disadvantages

- Currently, no WYSIWYG editors exist that can produce XQuery from a web page design. The same is true for QbE (Query by Example) front-ends.

In the following example we show how we can produce a web page with XQuery 4. The example is a simplified version of the first example shown in the section ***Procedural Transformation***.

```
default element ↩
namespace="http://www.softwareag.com/tamino/doc/examples/models/jazz/encyclopedia"

<html><head/><body>
    {for $a in input()/album
 return
    <table><tr><td>
      <table width="100%">
        <tr bgcolor="silver">
          <td>
            <h2>{string($a/title)}</h2><br/>
              { for $p in $a/publisher
                return
   string-join(("Publisher:",string($p))," ")
       }
            AlbumNo: {string($a/@albumNo)}
          </td>
          { for $c in $a/coverImage
            return
            <td>
              <img src="{string($a/coverImage)}" alt="{string($a/title)}"/>
            </td>
          }
        </tr>
      </table>
    </td></tr>
    <tr><td>
      <br/><h4>Tracks</h4>
      <table width="100%" >
 { for $track in $a/track
          return
          <tr bgcolor="silver">
            <td>
              { string($track/title) }
            </td>
            <td align="Right">
              { $track/duration }
            </td>
          </tr>
        }
      </table>
    </td></tr>
  </table>
  }
</body></html>

  ↩
```

Index

A

API
    JScript
        example of use, 191

C

composite key, 93
composition
    of documents, 99

D

data integrity, 67
document composition, 99
dynamic join, 100

E

efficiency
    in data modeling, 104
    in indexing, 105
    in queries, 106
element
    derived using server extension, 123
extensions
    Tamino extensions in XML Schema, 81

I

index
    definition, 89
integrity
    maintaining semantic integrity, 129

J

join
    dynamic, 100
JScript API
    example of use, 191

K

key
    composite, 93

N

namespace support, 87

P

performance issues, 111

Q

query
    efficiency, 103
    implemented as server extension, 117

S

schema
    mapping to web page
        example, 165
semantic integrity
    maintaining, 129
server extension
    sample derived element, 123
    sample query, 117
stylesheet
    using XSLT with Tamino, 162

U

UML
    from UML to XML, 71
unique IDs
    checking for, 68

X

XLink
    navigation, 175
XML Schema
    related web sites, 77
    Tamino-specific extensions, 81
XSLT
    introduction, 151
    rule-based transformation, 158
    using with Tamino, 162