Java Integration of XQuery - an Information Unit-Oriented Approach

Hans-Jürgen Rennau, parsQube GmbH
<hans-juergen.rennau@parsqube.de>

Abstract

An infrastructure for integrating XQuery into Java systems is described. The infrastructure comprises a new API (XQJPLUS, built on the standard API XQJ) and a tool for Java code generation. The basic idea of the approach is to deliver query results not in terms of query result items, but in terms of “information units”, ready-to-use entities assembled from the result items. The assembly process is guided by control information embedded into the query result, so that the query controls exactly what will be delivered, and in which form. Information units can represent information in a great variety of forms, including many map types and custom objects. An information unit associates its contents with a name, type information and meta data. The information units produced by a query are collected into a special container ("info tray") which offers name-based, intuitive access to the units. The query-specific structure of an info tray may be formally defined by a “tray schema” from which an "info shape" can be generated, a Java class representing a specific kind of info tray and offering compiler checked data access. As an approach to data integration, info trays are gathered into an "info space", a collection of possibly very heterogeneous data which can be addressed in a uniform way by string expressions ("info path" expressions).

Table of Contents

Introduction

To model information as XML offers remarkable advantages: the information is itemized in a way that endows each item with a name and an identity; each item has a context which in turn can be identified; and all items are exposed to navigation and querying.

These gains are not limited to the scope of single documents – they scale globally, thanks to the concepts of document URIs and namespaces. Using these concepts together makes it possible to fuse the total set of accessible XML resources into one single, homogeneous information space, within which document borders are crossed by single steps. Those resources need not store the data as XML: it suffices to *expose* them as XML, e.g. creating XML documents on the fly and supporting an XPath or XQuery interface. Relational databases, comma-delimited files and web services are all examples of resources which can be exposed as XML.

XQuery W3C XQuery is a language designed to live in and work with this XML-based information space. Many tasks of information processing can be handled by XQuery with amazing conciseness and simplicity. Much software could delegate certain information processing tasks to XQuery, thus using XQuery in a similar way to a client delegating processing tasks to a server. This integration is helped by the fact that XQuery is not constrained to produce XML. XQuery generates instances of the XDM data model, which means it can produce heterogeneous sequences of nodes and atomic values. So XQuery can deliver what is needed by its “clients” in a fine-tuned way.

Object-oriented programs are among those that can benefit from using XQuery. In the case of Java programs, this possibility is made more attractive by the existence of a standardized XQuery API – the XQJ XQJ Spec. This API fully supports the fine-grained data exchange between the languages required to let XQuery deliver exactly what is needed - for example a sequence of numbers, and only that.

Imagine a Java application that must process large and complex XML resources. Assume a message describes an approaching freight train, and the message must serve as the data source for many kinds of information, like arrival times, properties and quantities of goods, numbers and technical details of wagons, etc. Efficiency is highest and complexity is lowest if we let XQuery perform the largest possible chunks of processing, producing many of those “partial” results in a single pass.

The XQJ makes this possible in principle. It is very easy – although very tedious – to write the required Java code, using as input the XQuery result and producing as output the various objects desired (for instance documents, string collections, string-to-string maps, string-to-node maps, Properties objects, …). The code would iterate over the result items and handle them in accordance with their item type and the position in the result stream. (Yes, some agreement between Java developer and query author would be necessary in order to guide these decisions – e.g. “first the strings of code list #1, then an agreed upon separator element, then the terms of code list #2, then a separator, then alternating document names and document nodes”.)

In principle it is very easy, but it is very tedious, and it is extremely brittle: the slightest change in the query result – for example, a change in the sequence of the XML documents or code lists – will likely break the Java code. This breaking might easily go unnoticed, as the Java method might still produce sets and a map, only with incorrect content.

It's clear that the option to let the query produce complex multi-item results is problematic as long as there is no new channel opened through which the query author can push his intent clearly and explicitly “up” into the Java layer, structuring and associating content with meaning, as he would do within an XML document. This paper describes a new API, XQJPLUS, that addresses the issue of how XQuery can deliver structured, self-describing results in a secure and efficient way. Key aspects will be the simplicity, robustness and safety of the Java code, as well as the new responsibilities of the query code.

Conceptual framework

XML models information in terms of the Infoset and information items W3C Information Set. XPath 2.0 extends the model and defines the XDM W3C XDM, a model expressing information as a sequence of items which are nodes and/or atomic values.

The XDM is an achievement that can hardly be overestimated. It allows for representing all kinds of commonly used information in a unified, platform-independent way. However, computer programs are written in program languages, most of which cannot express XDM values directly. Therefore, XDM-supplied information must be “translated” into items of the computer language's internal data model. In Java, for example, the standard API for XQuery (XQJ) translates XDM items into a set of Java data items like primitive type items and objects of various types like String, Integer, QName, Element, etc.

XQuery programs produce a sequence of XDM items. Integration of XQuery into other languages should translate the items one-to-one into language-native data items – which is what XQJ accomplishes. So far, so good. But is this enough?

Information units

Information *items* are not necessarily the desired **information units**. For instance, the unit actually wanted might be an array of strings, or a string-to-string map, a string-to-node map, a Properties object etc. Any language not based on the XDM itself will use units of information which are not an unambiguous one-to-one mapping of an XDM item type, but correspond to an aggregation or transformation of itemized information. In this sense, every computer language can be regarded as a specific information platform with its own set of information unit types.

XQuery integration based on information units

To maximize the benefits of XQuery integration, the assembly of information items into commonly used information units should not be left to individual applications, as this would amount to frequent reinvention of the wheel or, worse, cause unwillingness to integrate XQuery at all. An alternative would be an **information unit-oriented API.** This API should be able to deliver useful information units like string-to-string maps, implementing their assembly from query result items behind the API’s façade. But before attempting to design the assembly and retrieval of information units, the information content of such units should be modeled.

Information unit: properties

An information unit contains information, but what properties does it have beside that? How about a name and meta data, how should the unit type be modeled? A good starting point is to model information units as components with properties.

Information unit: assembly

Information units are not necessarily one-to-one mappings of the XDM items. In particular, they may represent aggregations and transformations of XDM items. How are these items assembled into information units? Probably the most general and flexible way is to apply the concept of “markup” to result sequences: the XQuery result augments the target information with meta information controlling the unit assembly.

Information unit: access via information tray

XML exposes the information items as members of an Infoset, that is, as parts of a tree. What should be the structural unit that contains a number of information units? The approach described in this paper introduces the concept of an “information tray” (**info tray**, for short) which is a generic collection of information units, offering name-based access to information units as well as the possibility to expose them as nested structures. The word “tray” is meant to combine the idea of “tree” with a notion of convenience.

Tray schema

An info tray is a generic, name-aware data container, comparable to maps and XML documents. As with maps and documents, the meaning of names depends on the tray instance one is dealing with. Within a particular instance, the unit name “SampleDescription” might address a unit which is an element node, a name “ProjectActivities” might address a unit which is a string-to-string map, while a third name “DepartmentList” might address nothing, as it is not used by the tray. To express this situation, some kind of tray schema should define which names are used and what exactly they address. Note that if an XQuery program is designed to create a tray, the query result itself may be modeled in terms of a tray schema.

From info tray to info shape

Consider an XQuery API delivering query results as info trays. Further consider the availability of a tray schema which defines for a particular query the names and data types of the information units which it produces. Then it is a small step to imagine that such a tray schema could be compiled into a query-specific variant of a tray which would expose an access interface exactly fitted to the available information units. Their names and types could be combined into signatures which exclude any name/type mismatches. Such a tray variant which represents the exact “shape” of the information produced by a particular query might be called an **info shape**. It would offer type-safe, compiler-guarded access to information units, for example like this:

```
String getOrganizatonLongName();
Map<String,String> getProjectActivities();
```

which amounts to a Java binding not of Infosets in terms of info items, but query results in terms of ready-to-use information units.

Info path and info space

Like an XML document, an info tray associates information with names and supports the nesting of named information. When we associate the tray as a whole with a name, then any information unit within a set of info trays can be described in terms of a simple path consisting of a tray name and one or more unit names. If we add to this path a trailing part navigating *into* the unit, we arrive at the concept of an **info path** addressing contents found in a set of info trays in a similar way to an XPath addressing the contents of an XML document. Thus an **info space** simply denotes a set of info trays, associating each tray with a name.

XQJPLUS

The remainder of this paper introduces XQJPLUS, which is several things: an API and its implementation, a tool for generating Java components and, arguably, a style for integrating XQuery into Java. XQJPLUS is an attempt to elaborate and implement the conceptual framework sketched in the preceding section. The framework will serve as a reference when describing XQJPLUS, a single context within which to inspect various details.

XQJPLUS is not limited to the use of info trays

XQJPLUS offers many features which facilitate the work with XQJ and which are independent of whether one actually uses info trays. Examples are a query registration facility, automatic adaptation of the query base URI to the query file URI, simplified parameter type handling, and simplified result retrieval in the case of homogeneously-typed query results. These aspects are

ignored in the rest of this paper which concentrates on the possibilities offered by info trays and info shapes.

Getting started with XQJPLUS - an illustrative example

The goal of this section is to demonstrate how Java programs can be simplified by integrating XQuery in an "information unit-oriented way", receiving from XQuery information in the form of named, ready-to-use entities. So the example should be studied with two questions in mind: How would I get the work done without XQuery? And how would I do it using XQuery, but without information unit-oriented access to the query result? See Appendix B, *XQuery code performing the evaluations described in the example section* for a complete listing of the XQuery program which performs the evaluations used by the Java code.

The example considers the evaluation of data obtained from "National Water Information System (NWIS) Water-Quality Web Services". See USGS Web Services for technical documentation on how to use these web services. More specifically, we evaluate an XML file obtained from the NWIS Result Service. It contains water quality sampling results reported for the first three months of 2010, collected in the state of Minnesota. The data can be obtained from the following URI:

```
http://qwwebservices.usgs.gov/Result/search?organization=USGS-MN&startDateLo=
01-01-2010&startDateHi=03-31-2010&mimeType=xml
```

See Appendix A, *Sample of the data evaluated in the example section* for a sample of the data. The structure can be summarized as a sequence of activity reports ((`<Activity>` elements), each one containing a description of the activity (`<ActivityDescription>`) and a sequence of result reports (`<Result>` elements). Let us start with a set of simple evaluations.

A set of simple evaluations

evaluation "projectIds"

Desired result: a sorted list of project identifiers (`<ProjectIdentifier>`), delivered as a `SortedSet<String>` object. A string representation of the result might look like this:

```
860700319
860700379
NAWQA
```

evaluation "projectActivities"

Desired result: a nested map, where outer keys are project identifiers, inner keys are timestamps (obtained by concatenating `<ActivityStartDate>` and `<Time>`) and inner values are activity identifiers.

Desired result data type: `Map<String, Map<String,String>>`. A string representation might look like this:

```
860700319= (2 entries)
   2010-01-21#10:20:00=sun1dmnspl.01.01000102
   2010-02-09#09:30:00=sun1dmnspl.01.01000105

860700379= (1 entries)
   2010-01-22#12:01:00=sun1dmnspl.01.01000103
```

```
NAWQA= (2 entries)
   2010-01-20#09:20:00=sun1dmnspl.01.01000101
   2010-02-10#11:00:00=sun1dmnspl.01.01000104
```

evaluation "fractionResults"

Desired result: a nested map with outer keys specifying the sample kind (concatenation of `<CharacteristicName>` and `<ResultSampleFractionText>`), inner keys specifying the timestamp (concatenation of `<ActivityStartDate>` and `<Time>`) and inner values a list of result values (each one a concatenation of `<ResultMeasureValue>` and `<MeasureUnitCode>`) obtained for this combination of sample kind and time.

Desired result data type: `Map<String, Map<String, String[]>>`. A string representation might look like this:

```
…
Ammonia and ammonium/Dissolved=
   2010-01-20#09:20:00=
      0.49 [mg/l NH4]
      0.384 [mg/l as N]
   2010-01-21#10:20:00=
      0.283 [mg/l as N]
      0.36 [mg/l NH4]
   2010-01-22#12:01:00=
      0.08 [mg/l NH4]
      0.062 [mg/l as N]
   2010-02-09#09:30:00=
      0.307 [mg/l as N]
      0.40 [mg/l NH4]
   2010-02-10#11:00:00=
      0.55 [mg/l NH4]
      0.431 [mg/l as N]

Ammonia and ammonium/Total=
   2010-01-21#10:20:00=
      0.29 [mg/l as N]
      0.38 [mg/l NH4]
   2010-02-09#09:30:00=
      0.33 [mg/l as N]
      0.42 [mg/l NH4]
…
```

evaluation "activityResults"

Desired result: a map associating `UsgsResult` object arrays with string keys. The key specifies the activity identifier and the objects are custom objects representing an activity as described by one `<Activity>` element. It is assumed that `UsgsResult` objects can be loaded from `<Activity>` element information items.

Desired result data type: `Map<String, UsgsResult[]>`. A string representation of the result might look like this:

```
key=sun1dmnspl.01.01000101
result object:
==========================================
ResultValueTypeName=Calculated
ResultStatusIdentifier=Preliminary
MeasureUnitCode=mg/l NH4
```

```
AnalysisStartDate=--
ResultSampleFractionText=Dissolved
CharacteristicName=Ammonia and ammonium
PreparationStartDate=--
USGSPCode=71846
ResultMeasureValue=0.49

...
```

Obtaining the results via info tray

XQJPLUS offers the possibility to let the query create an info tray, a collection of information units which may be conveniently accessed. The following code snippet demonstrates how the evaluations defined in the preceding section might be performed from a Java developer's perspective.

```
// *** prepare query
// *****************
xq.registerQueryFile(queryName, queryFile);

// *** load info tray
// ******************
InfoTray tray = xq.execQuery2InfoTray(queryName, new FileInputStream(inputFile)

// *** read results
// *****************
SortedSet<String> projectIds =                        tray.getStringSortedSetObj
Map<String, Map<String, String>> projectActivities = tray.getNestedMapString2St
Map<String, Map<String, String[]>> fractionResults = tray.getNestedMapString2St
Map<String, Object[]> activityResults =              tray.getMapString2CustomOb

// *** process results
// *******************
// ...
```

The various results are "information units" which the Java code may conveniently pick from the tray: selecting the appropriate getter method (depending on the unit's data type) and specifying the result name as a parameter.

Obtaining the results via generated info shape

Using an info tray, Java code can contain two kinds of errors not detected at compile time: using a wrong access method for a given unit name, and using a unit name which does not occur at all in the tray. If this is an issue, one may work with *info shapes* instead of info trays. An info shape is a query-specific Java class plus interfaces, providing type-safe access exactly tailored to fit the query results. The Java code is generated from a succinct "tray schema" describing the result. A minimal version of such a schema would just state the names and data types of the units, as well as a distribution of the units over one or more read interfaces. See the section called "Tray schema" for a listing of the schema used to generate the Java resources used in this section. The schema is compiled into two Java interfaces and a back-end Java class. As an example, an excerpt of interface `ProjectData` looks like this:

```
public interface ProjectData {

//data access
    public SortedSet<String> getProjectIds() throws XQPException;
    public Map<String,Map<String,String>> getProjectActivities() throws XQPExce
```

```
//meta data access
    public Map<String,String> getProjectIdsMetaData() throws XQPException;
    public Map<String,String> getProjectActivitiesMetaData() throws XQPExceptio

// more signatures left out …
}
```

These generated Java components can be used for even simpler and safer Java access to the query results. This time, we let the query generate an info shape, rather than an info tray. The info shape is an object of the generated class `ResultReport`, from which we obtain the interfaces:

```
// *** create and load info shape
// ******************************
ResultReport resultReport =
   xq.execQuery2InfoShape(queryName, new FileInputStream(inputFile), null, new

ResultData  resultData  = resultReport.getInterfaceResultData();
ProjectData projectData = resultReport.getInterfaceProjectData();
```

We use these interfaces to read the query results:

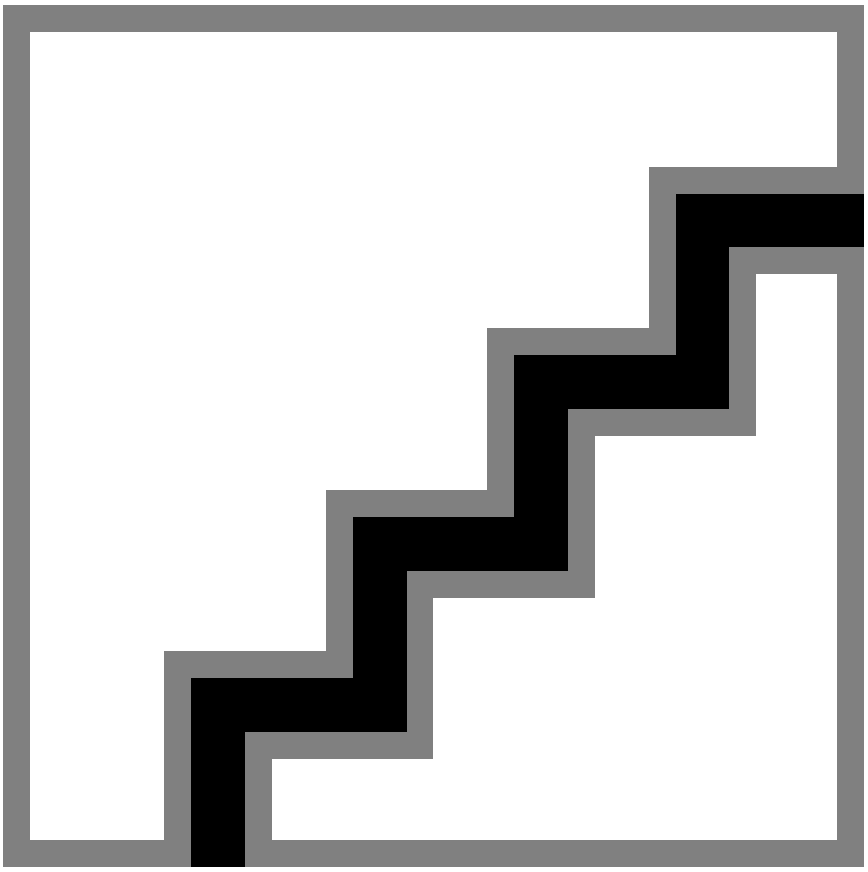
```
// *** read results
// ****************
SortedSet<String> projectIds                       = projectData.getProjectIds(
Map<String, Map<String, String>> projectActivities = projectData.getProjectActi
Map<String, Map<String, String[]>> fractionResults = resultData.getFractionResu
Map<String, UsgsResult[]> activityResults          = resultData.getActivityResu
```

When using such an info shape, rather than an info tray, mismatches between unit name and the unit's data type are excluded. A further advantage is the delivery of custom objects as instances of their specific type, rather than instances of `Object`.

Information unit-oriented access

Whether working with an info tray or an info shape - the API client deals with information units, rather than information items. The units are named and have a granularity adapted to the client's requirements, rather than being dictated by the XQuery data model. Figure 1, “From XDM items to information units” summarizes the process which enables this information unit-oriented access.

Figure 1. From XDM items to information units



Schematic representation of how query execution is coupled to the generation of an info tray. A query – myquery.xq – is executed and produces a sequence of XDM items. These are passed to the assembly machine, which transforms them into a set of named information units. The units are stored for later delivery. Delivery is enabled by a spectrum of get methods, each one dedicated to a specific unit data type.

Summing up

The example demonstrated the possibility to supply Java code with the results of XQuery evaluations in a convenient and intuitive way. What was the price to pay for this convenience, how much effort had to be made in the XQuery layer? The query constructing the info tray (or the info shape, if the tray schema was compiled into Java code) comprises 111 lines of code, including comments. (See Appendix B, *XQuery code performing the evaluations described in the example section* for a full listing.) An experienced XQuery developer can write the code in an hour or two. Adding in the fact that on the Java developer's side no learning or understanding of XQuery was required, leads to the conclusion that the integration of XQuery into Java may offer an interesting perspective in general - and especially so if realized in an information unit-oriented way.

Information units

This section deals with several aspects of information units. It starts with an elaboration of the concept, followed by an overview of the data types which an information unit may represent. The main part is a description how information units are assembled from XQuery result items. Finally, the use of meta data is explained.

Concept

The term ***information unit*** should capture the idea of a convenient entity to use when accessing, processing or delivering information. Of course, what is convenient depends on the situation and on taste. But as a starting point one may postulate that XDM items – nodes and atomic values – certainly *are* information units, but not the only ones one might wish for. A node or, say, a string can be exactly what one wants to deal with. Note however the general limitation that the XDM model has only two ways to express aggregation: (a) by a tree structure found within the content of a node, and (b) by regarding a sequence of items as an XDM value. The first must be unravelled using tree navigation, and the second is a primitive collection approach. From an application programmer's point of view, this may be insufficient.

Motivating example

Consider the situation that some processing requires as input a particular string-to-string mapping, for example mapping project identifiers to project names. Such a mapping can easily be represented by an XML structure, for example:

```
<projects>
   <project id="…" name="…"/>
   <project id="…" name="…"/>
</projects>
```

This ease of representation should not be confused with ease of usage. The Java type `Map<String, String>` is tailored exactly for the purpose of working with a string-to-string mapping. At least in situations where only key-based access is desired (as opposed to querying the sets of keys or values), the map may be considered simpler and more straightforward. After all, working with the XML representation one would have to be aware of three paths: one leading from the map root to the nodes representing map entries, one leading from the entry root to the key and one leading from the entry root to the value. (In the example, the paths would be "project", "@id" and "@name".) These paths would have to be supplied either by convention (for example using standardized element names "map" and "entry" and attribute names "key" and "value"), or by explicit meta information. So the XML representation needs meta information. Another aspect: the navigation from key to value can be achieved in different ways, e.g. via XPath or via DOM methods. So the XML representation is not unambiguous and explicit as to the appropriate usage style. Apart from that, the map access is probably more efficient. Taking these aspects together, one may conclude that XML's wonderful ability to represent almost any kind of information does not imply that it is always the most appropriate format for using or processing that information. This is where "information units" come into play.

Before continuing, let's briefly compare the use of information units with data binding. Data binding replaces an XML structure by a one-to-one object representation. This is like a "push" model: the XML comes first and dictates the object representation, which is essentially a translation of the item tree into an object tree. Information units, on the other hand, constitute a "pull" model: what comes first is the desired representation of information; it is assembled by *using* XDM items, rather than echoing them.

Modelling information units

An information unit encapsulates data. It should also contain information about the data, as becomes obvious when one considers an element information item as a general prototype of information units. An element has

- content, which is the data represented by its child nodes
- a name, serving to distinguish it from other (sibling) elements
- attributes, a set of named atomic values associated with the element

- a type, which is a description of what kind of content to expect

XQJPLUS defines the following generic model of an information unit: it has

- *content*, which is an instance of a programming language's data type
- a *name*, serving to distinguish it from other, sibling units
- *meta data*, which is a set of named string values associated with the unit
- a *type description* composed of three components:
  - a physical type (QName)
  - a semantic type (QName)
  - an implementation type (a string)

content

XQJPLUS constrains the content to be a single Java object, or an array of Java objects.

name

The name distinguishes an information unit among siblings. The concept of siblings relates to a collection of units, which is not an inherent part of the model of an information unit. XQJPLUS introduces a distinct concept – the info tray – to model collections of information units. So the name of an information unit serves to distinguish it from other units within the same info tray. Info trays will be discussed in the section called “Information tray (info tray)”.

meta data

Any information unit is associated with a set of named values, modelled as a QName-to-string mapping.

type description

The type description comprises three names. The contents are guaranteed to be an instance of the ***physical type***, which is the data type defined in the programming language. Note that this may be an interface type or a base type of the data type that is used, in which case the underlying data type is only partially revealed by the physical type. The underlying type is however fully revealed by the ***implementation type.***

The ***semantic type*** is a name meant to identify what the content “means”. For example, while the physical type may specify the content to be a string-to-string map, the semantic type might identify the contents to be a mapping of project identifier to project name. To specify this, a name like “{http://www.example.com/ns/proj}projectIdTable” might be invented. Note that the semantic type only identifies by specifying a QName, it does not define or describe. XQJPLUS does not define the interpretation of the semantic type.

n physical type names

XQJPLUS implements the integration of XQuery into Java – therefore the physical type amounts to a Java type. However, the concept of an information unit and its properties is not tied to a specific language, and XQJPLUS attempts to preserve the language-independence of the concept as far as possible. Therefore the physical type of an information unit is not defined to be a Java type name, but a QName representing an “abstract” type that might be implemented by different languages. For example, a string-to-string map can be represented by many other languages, too. The physical type name is “map_string_to_string_object”, while the corresponding Java type name is “`java.util.Map<String,String>`”. If, for example, a Perl

implementation of XQJPLUS existed, the corresponding Perl type name would probably be `Hash`.

The physical type is defined to be a QName. XQJPLUS uses physical type names without a namespace URI, suggesting this pattern: standard physical types are in no namespace; any custom physical type (that is, physical types introduced by project-specific extensions of XQJPLUS) must use a name with a namespace URI.

Available physical types

As the results of a query execution are delivered as information units, a key question is: what Java types are available? How is the concept of an information unit implemented in terms of Java types? If we compare XQJPLUS with a restaurant where the XQuery developer is the cook and the Java developer a potential guest, the range of available Java types is like the menu. Without a glance at that range, the value of XQJPLUS cannot be assessed.

Note: The preceding subsection introduced the notion of physical types which are mapped to Java type names. In the current subsection the Java type names are used, although the physical type is defined as a QName which is usually different from the Java type name.

As a starting point, consider the Java types which the underlying XQJ API delivers.

Table 1.

Summary of Java types delivered by XQJ.

category	**types**
atomic types	Boolean, BigDecimal, BigInteger, Byte, Double, Duration, Float, Integer, Long, QName, Short, String, XMLGregorianCalendar
node types	Document, Element, Attr, Text, Comment, ProcessingInstruction

This spectrum is included and extended by XQJPLUS.

Table 2.

Summary of Java types delivered by XQJPLUS. Note that for each type a single object variant and an array variant is supported, e.g. String and String[].

category	**types**
atomic types	Boolean, BigDecimal, BigInteger, Byte, Double, Duration, Float, Integer, Long, QName, Short, String, XMLGregorianCalendar
node types	Document, Element, Attr, Text, Comment, ProcessingInstruction
maps	Map<String,T>, T=String, String[], Node, Node[], Boolean, Object, Object[] Map<String,Map<String,T>>, T=String, String[], Node, Object
collections	List<String>, List<Node>, Set<String>, Set<Node>, SortedSet<String>, Queue<String>
miscellaneous	java.util.Properties

category	**types**
custom types	(any type implementing LoadableFromXML)
exceptions	InfoTrayLoadingException
info tray	InfoTray

The extension compared to XQJ has four aspects:

- XQJPLUS units can have either a single object type or an array type
- XQJPLUS adds further type categories (maps, collections, …)
- XQJPLUS units can have custom types (!)
- XQJPLUS units can be nested (using InfoTray or InfoTray[] as physical types)

support for array types

The query result which XQJ delivers is one single, unstructured sequence of items. It is not possible to express any grouping of items into sub sequences. For example, if the result consists of 10 strings, there is no way to communicate explicitly that the first six strings should be viewed as one array, followed by a single item, followed by an array of three items. In XQJPLUS, on the other hand, the result could be delivered as three units, two of which are arrays.

added categories

Map types are thought to be especially useful, as XML evaluations often yield information which can be modelled as a map. The map keys are strings, the map values can be strings, booleans, nodes or custom objects (see below). Maps can be simple or nested. *Collection types* are probably less important than maps, due to the general support for array types. The type `java.util.Properties` is currently the only type of category "miscellaneous", but the addition of other types is intended. Admitting *exception objects* as information units enables query authors to communicate error conditions in a unified way.

custom types

Information units can be custom type objects. Custom types are (presently) required to implement the interface `LoadableFromXML`, which is part of the main package of XQJPLUS. It contains a single method `loadFromXML` which loads an object with data received as an XML node. The construction of custom type objects will be discussed in a later section.

nested units

An information unit may be an `InfoTray` object, which represents a collection of named information units. (See the section called “Information tray (info tray)” for a discussion of info trays.) This amounts to supporting a type of unit which is a tree consisting of `InfoTray` objects as internal nodes and non-`InfoTray` objects as leaf nodes. In fact, a query result as a whole can be regarded as such a tree. Note that the tree of info trays and non-tray leaves must not be confused with an XML tree. A leaf of the latter contains an atomic value; a leaf of the former may, for example, contain an XML document, or a mapping of strings to XML documents.

Information units are retrieved from an info tray by calling a specific getter method which has the unit’s physical type as return type. See the section called “Read access” for more information about API access to information units.

The assembly process - basics

After deciding which physical types XQJPLUS must support, we can investigate their assembly. The assembly is wholly controlled by the XQuery code. XQuery thus provides both the *information*

content and the *control information* required to transform this content into a collection of information units.

Let us look at an example. The following XQuery snippet produces two result items – a *control item* and a *data item* - which together will command the assembly of an information unit consisting of a string-to-string map:

```
<xqjp:part name="activityTimes"
           type="map_string_to_string_object"
           entryPath="activity"
           keyPath="@id"
           valuePath="@time"
           sematype="usgs:activityTimes" />,

<activities>{
   for $a in //Activity/ActivityDescription return
      <activity id="{$a/ActivityIdentifier}"
                time="{concat($a/ActivityStartDate, 'T', $a/ActivityStartTime/T
}</activities>
```

The control item `<xqjp:part>` announces the construction of an information unit with name “activityTimes” and physical type “map_string_to_string_object”. This type name corresponds to the Java type `Map<String,String>`. The data to be written into the map follow in the next result item, which is a *data item*, rather than a control item. The assembly of the information unit (a map object) from the contents of the data item (an element node) is configured by the control item’s attributes “entryPath”, “keyPath” and “valuePath”. The values of these attributes are interpreted as XPath expressions. The entry path is applied to the data item’s root and leads to the nodes representing the map entries. The other paths are applied to these nodes, one at a time, yielding a map entry’s key and value, respectively.

Another example illustrates the mapping of a result item sequence to several information units. The query generates three information units, named “projectList”, “organizationProfiles” and “countActivities". To achieve this, it produces a stream of data items with control items interspersed.

```
<xqjp:part name=”projectList”
           type=”strings”/>,
distinct-values(//ProjectIdentifier),

<xqjp:part name=”organizationProfiles”
           type=”nodes”/>
<orgProfile>…</orgProfile>,
<orgProfile>…</orgProfile>,
<orgProfile>…</orgProfile>,

<xqjp:part name=”countActivities” type=”integer”>,
count(//ActivityIdentifier)
```

The examples illustrate the basic rules of the assembly process.

Rule #1: control information is placed in top-level control items

Control information is contained by top-level control items. Each result item is either a data item or a control item. Data and control information is never mixed within result items.

Rule #2: special control items define information units

Each information unit created by the query is defined by a control item (`<xqjp:part>` element) which specifies all unit properties: name, meta information, type (physical, semantic, implementation types). Exceptions to this rule are units defined to be an info tray. The assembly of such complex units is discussed in a special section.

Rule #3: the source data belonging to an information unit follow the unit's definition

The source data belonging to a unit are given by all the result items which immediately *follow* the unit-defining control item and come before the next control item. So the basic structure of a query result generating an info tray looks like this:

```
...
<xqjp:part name="x" type="..."/>,
- source data items for unit "x",

<xqjp:part name="y" type="..."/>,
- source data items for unit "y",

<xqjp:part name="z" type="..."/>,
- source data items for unit "z",
...
```

Two exceptions to the rule are units which are themselves info trays and unit definitions which reference control variables (see next section).

Note: For a comprehensive list of all control items defined by XQJPLUS, see Appendix C, *The control items defined by XQJPLUS*.

The assembly process - advanced

Namespace context

Unit assembly is often controlled by XPath expressions. The namespace context in which to evaluate those expressions is created by special control items. Example:

```
<xqjp:setNamespaceContext>
   <xqjp:binding prefix="data" uri="http://qwwebservices.usgs.gov/schemas/WQX-O
   <xqjp:binding prefix="eval" uri="http://www.bits-ac.com/usgs"/>
</xqjp:setNamespaceContext>
```

Control variables

Since the data items of the query result are the source data for unit assembly, you may want to ***reuse*** source data in the assembly of different units. A simple example is a query which produces two string-to-string maps, where the second interchanges the keys and values used by the first – for example a "code2text" map and a "text2code" map. To achieve this without repeating the source data one may use control variables. A control variable is declared by a special control item (`<xqjp:setVar>`), which specifies a name and minimal type information (atomic or node, single value or sequence). During the assembly process, all data items following the declaration and preceding the next control item are bound to a variable of the declared name. Thereafter, they can be referenced by unit-defining control items (`<xqjp:part>`) via a special attribute ("source"). The following XQuery code illustrates this possibility by generating two maps backed by the same data item. Map "activity2time" is string-to-string, whereas the inverse map "time2activity" is string-to-string[]. Both maps are assembled from the contents of control variable "activityDates".

```
(: set variable "activityDates"
 : ========================== :)
<xqjp:setVar name="activityDates" vtype="node" />,

<activityDates>{
   for $date in distinct-values(//ActivityStartDate)
   let $activities := //ActivityStartDate[. eq $date]/../ActivityIdentifier
   return
```

```
      <activityDate date="{$date}">{$activities}</activityDate>
}</activityDates>,

(: define information units
 : ======================= :)
<xqjp:part name="time2activities"
           type="map_string_to_strings_object"
           sematype="wq:timeActivities"
           impltype="java.util.TreeMap"
           source="$activityDates"
           entryPath="activityDate"
           keyPath="@date"
           valuePath="ActivityIdentifier" />,
<xqjp:part name="activity2time"
           type="map_string_to_string_object"
           sematype="wq:activityTimes"
           source="$activityDates"
           entryPath="activityDate/ActivityIdentifier"
           keyPath="."
           valuePath="../@date" />
```

Information units that are themselves trays

As an information unit may itself be an info tray, the result item stream must somehow delimit the begin and end of such an embedded tray. To do so, use a `<xqjp:complexPartBegin>` and a `<xqjp:complexPartEnd>` control item. As an embedded tray is an information unit, the begin marker also plays the role of a unit-defining item. It must specify the unit name and may specify a semantic type and meta data, just like any other unit-defining item.

The assembly process - how XQuery constructs Java custom objects

As mentioned before, an information unit may contain custom objects, that is, objects of a type which is not a Java standard type. Here are a few physical types involving custom objects:

custom_object	single custom object
custom_objects	an array of custom objects
map_string_to_custom_object_object	a string-to-customObject map
map_string_to_custom_object_objects	an array of string-to-customObject maps
nested_map_string_to_custom_object_object	a string-to-(string-to-customObject) map

Presently, XQJPLUS allows only those types as custom types which implement the interface `LoadableFromXML`. This interface is defined in the main package of XQJPLUS and contains a single method signature:

```
void loadFromXML(Node source);
```

An underlying assumption is that the objects can be built in a two-step way, first instantiating them using a parameter-less default constructor, then calling the load method. Later versions of XQJPLUS may support other categories of custom types which can be constructed using different interfaces.

The actual custom type is a special construction attribute (“customType”). This construction attribute is always required if the physical type involves custom objects. Example:

```
<xqjp:part name="placeObjects"
           type="custom_objects"
           customType="com.bits_ac.xqjplus.appl.Place"
/>,
```

The query code has to provide the data source in the usual way, that is, either by subsequent data items or using the “source” attribute. The XML nodes to be used for loading the custom objects must of course provide the XML data structure expected by the particular custom type’s “loadFromXML” method.

Meta data

An information unit is associated with meta data, which are a set of named values. The names are QNames, and the values are strings. The meta data fall into two groups:

- *standard meta data* have XQJPLUS-defined semantics
- *custom meta data* have custom semantics

Most standard meta data are automatically assigned by XQJPLUS. Custom meta data are always assigned by the unit-defining control item.

Custom meta data

Associating a unit with custom meta data is straightforward: they are specified either as attributes or as child elements of the unit-defining control item (`<xqjp:part>` or `<xqjp:complexPartBegin>`). The name of a meta data item is the respective attribute’s or element’s name; the value is its string value. If a meta data item is specified by an attribute, the attribute name must have a non-empty namespace URI. If a meta data item is specified as a child element, the element name may or may not be in a namespace.

The following example demonstrates the use of both, attributes and child elements:

```
<xqjp:part name="startDates"
           type="string_sortedset_object"
           sematype="usgs:activityStartTimes"
           bi:cr="{current-date()}" >
    <bi:min>{min($startDates)}</bi:min>
    <bi:max>{max($startDates)}</bi:max>
    <meta:documentation>The activity start dates, as reported by the NWIS result
    service.</meta:documentation>
</xqjp:part>,)
```

The definition of unit “startDates” specifies three custom meta data items (“cr”, “min” and “max”) and a standard meta data item “documentation”.

Standard meta data

Standard meta data have a name in the namespace “http://www.bits-ac.com/xqjplus/meta”. With the exception of “documentation”, which is provided by the query in the same way as custom meta data, standard meta data are automatically assigned by XQJPLUS when transforming a query result into an info tray. Currently, there are four kinds of automatically assigned standard meta data: “length” and the three type specifications “ptype”, “stype”, “itype”.

Standard meta data item “length”

It contains the number of objects contained by the unit: if the unit has an array type, it is the number of array items, otherwise it is “1” or “0” depending on whether the unit actually contains data or is empty.

Standard meta data “ptype”, “stype”, “itype”

They contain the names of physical type, semantic type and implementation type, respectively. So the type information is available as meta data.

Note. The implementation type can be specified by the unit-defining control item, using attribute “impltype”. Strictly speaking, this is not meta data information, but assembly control information. Therefore the value of “itype” may depend on the query text, although it is automatically assigned.

Information tray (info tray)

When an information unit is created, it is placed on an information tray (info tray). This section introduces info trays. After defining the concept, the information flow into and out of the tray is described: the loading of a tray with information units, and the reading of units contained by the tray. The final subsection introduces tray-level meta data.

Concept

An info tray is a collection of information units. Viewed as a component, its only properties are

- a partially ordered list of information units
- tray-level meta data
- an optional resource ID

The list of *information units* is partially ordered, as the order of two information units is irrelevant if they have different names, but relevant if they have the same name. *Meta data* are a set of named string values providing information about the tray. The *resource ID* is a QName identifying the tray instance within a set of tray instances.

The info tray concept is independent of the mechanisms of loading the tray. XQJPLUS represents the general concept of an info tray by a class `InfoTray`; and it models the loading from an XQuery result by providing a subclass of `InfoTray` implementing such a loading method. Other subclasses of `InfoTray` might be developed which load the tray from, say, an XSLT or XProc result, or a set of delimited files or yet other kinds of data sources. Note that the read access methods are all harboured by the `InfoTray` level, as they reference only the assembled information units, never any information about the assembly process. In consequence, once the tray is loaded the actual loading mechanism is invisible from the Java perspective and may be replaced at any time.

The interface of an info tray contains

- a method for *loading* the information units from a data source
- methods for *adding* information units
- methods for *reading* information units
- a method for *validating* the info tray against a tray schema

The validation of an info tray will be dealt with in the section called “Tray validation”.

The load process

An info tray is loaded by adding information units one by one. The intended pattern is to let subclasses of `InfoTray` define a data source-specific load method which assembles the units in a source-specific way and adds them to the tray by calling the base class's `add` methods.

`add` methods

The source-specific loading is made as simple as possible by arming `InfoTray` with a comprehensive set of source-independent `add` methods. These methods receive a unit's completely assembled data content, along with meta data and unit name. For each supported physical type, there is a special `add` method. Some examples:

```
void addString(String partName,
               String content,
               Map<String,String> meta);

void addStrings(String partName,
                String[] content,
                Map<String,String> meta);

void addMapString2Node(String partName,
                       Map<String,Node> content,
                       Map<String,String> meta);
```

Note: the semantic type is communicated as a meta data item, while the physical type is implied by the choice of the method and the implementation type is determined by the actual content.

load methods

Source-dependent load methods are defined by subclasses of `InfoTray`. XQJPLUS provides the subclass `XQueryResponse`, defining a method

```
void loadXQuerySequence(XQSequence seq);
```

The method iterates over the result sequence, assembling information units as defined by the control items and adding them to the tray by calling the base class's `add` methods.

Read access

The info tray offers convenient read access to its information units. Access is always to one unit at a time. The access to a unit can be grouped as follows:

- reading the unit contents
- reading the unit type information
- reading the unit meta data

The unit is usually specified by name – see below for details.

reading the unit contents

For each supported physical type of information units, there is an access method returning an instance of the physical type and accepting as input a unit name. (Use Clark notation for names in a namespace.) When there are several units with that name, the first is delivered. Examples for methods reading unit contents:

```
String        getString(String name);
```

```
String[]      getStrings(String name);
Node          getNode(String name);
Node[]        getNodes(String name);
Properties    getPropertiesObject(String name);
Properties[]  getPropertiesObjects(String name);
Object        getCustomObject(String name);
InfoTray      getInfoTray(String name);
Map<String,String>     getMapString2StringObject(String name);
Map<String,String[]>   getMapString2StringsObject(String name);
Map<String,String[]>[] getMapString2StringsObjects(String name);
```

A tray may indeed contain several units with the same name. In order to retrieve them, first obtain a cursor and then use it to iterate over those units. The code can rely on all units with a common name to have the same physical type, as a violation of this constraint would be detected during tray loading and generate an exception. In the following example code, the common physical type is a single node:

```
InfoTrayCursor cursor = getCursorByPartName("laboratory");
while (cursor.hasNext()) {
   Node currentUnit = getNode(cursor.next());
   // process the contents …
}
```

This pattern is based on the fact that for each physical type there is a second read method expecting a unit index as input, rather than a unit name. The unit index is a non-negative number identifying the unit within the tray independently of its name.

Note: When using a unit name repeatedly, each unit with that name has its own meta data, just like any other unit.

reading the unit type information

The info tray offers separate methods for reading a unit's physical, semantic or implementation type. Only name-based access is offered, as the type information must not vary between units which have the same name.

reading the unit meta data

Unit meta data can be retrieved on bloc as a `Map<String,String>`, on bloc as a `Map<QName,String>` or itemwise, specifying the meta data item name as a QName or a Clark string. As when reading unit content, the unit is identified either by name or by passing a cursor.

reading tray meta data

Tray meta data are retrieved in a similar way as unit meta data.

Meta data

Not only the individual information units are associated with meta data, but also the info tray as a whole (and implicitly, also any info shape generated for the info tray). Standard meta data are distinguished from custom meta data.

Custom meta data

Tray-level custom meta data are specified via a dedicated control item (`<xqjp:trayMeta>`). The mapping of attributes and child elements to meta data items is the same as with unit-related meta data.

Standard meta data

The namespace of tray-level standard meta data is the same as of unit-level ones. Two standard meta data items can be specified by the query – “documentation” and “name”. The latter is meant to identify the specific type of a tray created by a specific query, comparable to the name of an XML document’s root element. Other standard meta data items are assigned automatically. These are:

- “length” specifies the number of top-level units
- “unitNames” is the sorted set of top-level unit names, rendered as a white-space separated list

Tray schema

Like an XML document, an info tray has a structure that can be described independently of the actual data content. XQJPLUS supplies a simple XML vocabulary that can be used to specify a tray schema which may (a) constrain the tray structure, (b) control the generation of an ***info shape***, which is Java code offering enhanced safety and convenience of data access. An info shape consists of a Java class plus interfaces representing an info tray with a specific structure.

Purpose

There are three main purposes for creating a tray schema:

- to facilitate the cooperation between Java developer and XQuery developer
- to validate a tray instance, e.g. before starting to use it
- to generate an info shape

Constraining the tray structure

To constrain the tray structure, the following kinds of information can be expressed:

- the names and types of contained information units
- cardinality constraints referring to the repeated use of unit names
- cardinality constraints referring to the contents of units
- meta data constraints (tray-level and unit-level meta data)

Controlling the Java binding

If the tray is used as input for generating Java code (or other language code), the tray schema must contain further information controlling the language binding. In case of Java code, this consists of class and interface names, as well as a distribution of the unit names over the supporting interfaces.

A simple example

Here is a minimal schema from which the info shape demonstrated in the section called “Obtaining the results via generated info shape” was generated:

```
<ts:traySchema xmlns:ts="http://www.bits-ac.com/xqjplus/traySchema"
               xmlns:java="http://www.bits-ac.com/xqjplus/traySchema/java-bindi

   <!-- ============== -->
   <!-- tray structure -->
   <!-- ============== -->

   <ts:tray>
```

```
      <ts:part name="fractionResults" type="nested_map_string_to_strings_object
      <ts:part name="activityResults" type="map_string_to_custom_objects_object
               customType="com.bits_ac.xqjplus.appl.UsgsResult" />
      <ts:part name="projectIds" type="string_sortedset_object" />
      <ts:part name="projectActivities" type="nested_map_string_to_string_objec
   </ts:tray>

   <!-- ============ -->
   <!-- Java binding -->
   <!-- ============ -->

   <!-- implementation class -->
   <java:trayBinding className="ResultReport"
                     package="com.bits_ac.xqjplus.appl" >

   <!-- interfaces -->
      <java:iface name="ResultData" package="com.bits_ac.xqjplus.appl">
         <java:part ref="fractionResults"/>
         <java:part ref="activityResults"/>
      </java:iface>

      <java:iface name="ProjectData" package="com.bits_ac.xqjplus.appl">
         <java:part ref="projectIds"/>
         <java:part ref="projectActivities"/>
      </java:iface>
   </java:trayBinding>

   <!-- mapping customType => className -->
   <java:customTypeBindings>
      <java:customTypeBinding customType=".*" className="$1"/>
   </java:customTypeBindings>
</ts:traySchema>
```

Constraining meta data

The structure part of the tray schema shown above is minimal, as it does not constrain the use of meta data. Meta data can be constrained by

- stating names of mandatory meta data
- constraining the values of meta data items
- excluding the use of other meta data names apart from the schema-specified names

The following example adds more constraints:

```
<ts:part name="projectIds" type="string_sortedset_object"
      unitOccurs="+"
      contentOccurs="?">
   <ts:meta name="bi:priorityRemark" use="required" />
   <ts:meta name="bi:projectCodeList" use="required" >
      <ts:enumeration value="internal"/>
      <ts:enumeration value="external"/>
   </ts:meta>
   <ts:meta name="bi:source" use="optional">
      <ts:pattern value="usgs-..-\d{8}.xml"/>
   </ts:meta>
</ts:part>
```

The “unitOccurs” attribute (which defaults to “1”) specifies that the tray may contain more than one unit with name “projectIds”. According to “contentOccurs” (also defaulting to “1”), each unit may contain zero or one data items. (Note that non-array physical types can have only “?” or “1” as value of “contentOccurs”.) Two meta data items are mandatory (“bi:priorityRemark” and “bi:projectCodeList”), for one of which value constraints are specified. The meta data item “bi:source” is optional, but when it occurs it must match the specified constraint.

By default, meta data are “open” – there may be more meta data than specified in the tray schema. To exclude this possibility, use the attribute “metaClosed" with a value of "true" on the `<ts:part>` element.

Tray validation

The `InfoTray` class offers a method `validate`, which validates the tray contents against a tray schema. Note that this validation ignores all schema parts relating to the language binding. If errors are detected, the validation writes one or more `InfoTrayLoadingException` units, so that after validation the diagnostics can be read from those units.

The main purpose of validation is to support the development process. When the query has reached maturity, it should guarantee to either deliver a tray that conforms to the agreed upon schema, or it should itself construct `InfoTrayLoadingException` units with appropriate diagnostics. In other words it should not leave the construction of exceptions to the validation. Following these steps for tray processing will help minimize problems:

- *always* check for buffered exceptions before starting to use the tray
- in the pre-production phase call `validate` before checking for buffered exceptions; later this call can be removed

Suggested sample code:

```
tray.validate();                    // remove this line when the underlying query
if (tray.hasLoadingExceptions()) { // but ALWAYS check for buffered exceptions!
   for (InfoTrayLoadingException e: tray.getLoadingExceptions()) {
   // process exception ...
} else {
   // normal processing of tray contents ...
}
```

Info shape

Code generation

Once a tray is constrained by a tray schema, the possible use of the tray is defined exactly, and exact answers are implied to the following questions:

- which unit names are available
- which physical type belongs to a given unit name
- which meta data items can be expected

If, for example, the tray schema determines that the unit name “projectIds” uses a physical type “strings”, then it is clear that data retrieval for this unit name (a) is possible and (b) must use the getter variant which delivers a string array:

```
String[] pnames = tray.getStrings (“projectNames“);
```

To pass this unit name to a different getter method – say, the variant delivering a node – will result in an exception. From here it is a small step to generate a Java class which exposes the precise set of possible read methods, making their signatures error-proof by removing the name parameter as a parameter and shifting it into the method name:

```
String[] pnames = shape.getProjectNames();
```

This way, the name and physical type of the unit have become inseparable, excluding mismatches. The generated implementation of this method consists in a call of the generic tray method:

```
public String[] getProjectNames() throws XQPException {
   return dataSource.getStrings("projectNames");
}
```

Such code generation is supplied by the code generator of XQJPLUS. The code generator takes a tray schema as input and produces an **info shape**: a query-specific class which is backed by an `InfoTray` member and defines the specific access methods as described above. Along with this class, interfaces are generated according to the directions given in the tray schema.

Info shape

Whereas an info tray is a generic data container, an *info shape* is a specific data container, reflecting particular choices concerning the names and types of information units. Hence the name - it reflects the particular "shape" resulting from those choices. As described in the preceding section, info shapes are generated by the code generator of XQJPLUS.

To give an example, the simple tray schema shown in the section called “A simple example” implies the following info shape:

```
public class ResultReport implements LoadableFromInfoTray, ResultData, ProjectDa
    protected InfoTray dataSource;
    ...
//data access
    public Map<String,Map<String,String[]>> getFractionResults()   throws XQPExc
    public Map<String,UsgsResult[]>         getActivityResults()   throws XQPExc
    public SortedSet<String>                getProjectIds()        throws XQPExc
    public Map<String,Map<String,String>>   getProjectActivities() throws XQPExc

//meta data access
    public Map<String,String> getFractionResultsMetaData()   throws XQPExceptio
    public Map<String,String> getActivityResultsMetaData()   throws XQPExceptio
    public Map<String,String> getProjectIdsMetaData()        throws XQPExceptio
    public Map<String,String> getProjectActivitiesMetaData() throws XQPExceptio

//deliver interfaces
    public ResultData getInterfaceResultData() {...}
    public ProjectData getInterfaceProjectData() {...}
    ...
}
```

The advantages for the Java developer are obvious – compiler-guarded access to the information unit with no worries about using the right names and the right physical types. Further advantage: when a unit’s physical type uses custom objects, the generated method delivers instances of the custom type, rather than of type `Object`:

```
    public Map<String,UsgsResult[]> getActivityResults() throws XQPException {.
```

The tray schema controls the generation of interfaces which contain only subsets of the info shape’s read methods. This approach parcels out the results of complex XQuery processing in such a way

that Java components can reduce their dependence on the tray design: if the structure of the tray is changed, for example a unit is removed or added, the impact is limited to the users of particular interfaces.

Info path and info space

This section briefly explores how info trays create a new possibility of data integration. Note that this aspect of info trays does not depend on whether the trays are constructed by XQuery or any other means. Indirectly, however, the possibility of loading info trays with XQuery further increases the potential for data integration, as it favours the use of XML resources as input, a good approach to integration in the first place (compare the section called “Introduction”).

The introduction of this paper emphasized the way XML and XPath together create a uniform information space encompassing any XML resource accessible via document URI. On the other hand, from an application programmer’s point of view it is often desirable to access information in formats other than XML documents – hence the concepts of information units and info trays described in this paper. And, of course, this also explains the popularity of data binding. So we can ask: do info trays fall out of that uniform information space, in a similar way to the products of data binding?

Compared with data binding, the introduction of info tray objects creates a radically different situation, as info trays are generic data containers. Trays identify their units by QNames, and so the contents of an info tray can be addressed by path expressions similar to XPath expressions. If we associate info trays as a whole with an identifying QName – call it a “resource ID” – we can collect any number of info trays into a set so that the content of any one of those trays is addressable in a uniform way.

For this purpose XQJPLUS introduces the concepts of an **info path** and an **info space**, and it supplies the definition of an info tray with a property called resourceID, typed as a QName.

Info space

An info space is simply a set of info trays. Within the set, the resource IDs must be unique, so that an info space is essentially a mapping of resource IDs to owner trays. The name “info space” is meant to evoke the notion of navigation and uniform accessibility: any content of any member tray can be addressed in a uniform way, using info paths as detailed below.

Info path

An info path is a string expression addressing data within an info space. Like an XPath, it is composed of steps, and the trailing part of it can in fact be a real XPath, as long as the info path points into an information unit containing nodes. Outwardly, an info path looks like an XPath: it is a sequence of steps, and these steps can be grouped into three consecutive sections:

- **(1) tray selection**

  - the first step identifies a particular info tray within the info space.

- **(2) information unit selection**

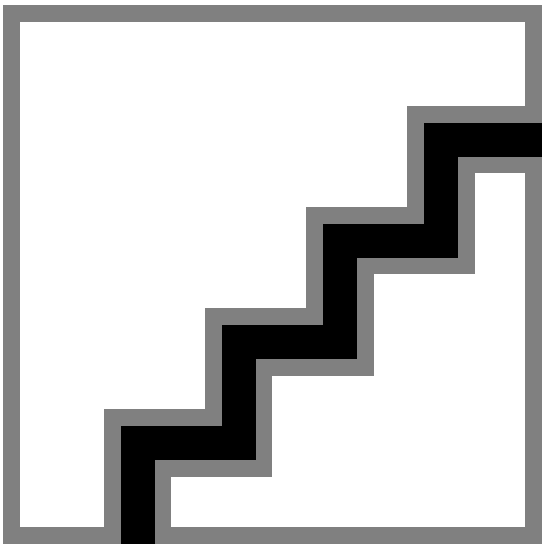
  - one or more steps identify a particular information unit within the tray.

- **(3) intra-unit navigation**

  - an optional trailing part of the info path navigates into the particular unit identified by the preceding steps.

See Figure 2, “Example of an info path” for an example.

Figure 2. Example of an info path



This info path selects the value of a map entry with key "Ammonium"; the map is selected as a specific information unit found in a specific info tray.

The steps selecting info tray and information unit consist of QNames. The trailing steps defining intra-unit navigation, on the other hand, have a syntax and semantics which are determined by the physical type of the unit in question. To achieve a precise understanding of how an info path is decomposed and evaluated, consider this string:

```
infospace://a/b/c/d
```

The prefix “infospace://” indicates that the string represents an info path, rather than an XPath. The path is evaluated in the context of a specific info space. This space cannot be inferred from the path expression. The step “a” identifies a tray within that info space. The next step “b” identifies a unit within that tray. If this unit does not contain nested units (in other words, if it is not a tray or an array of trays), then the remaining steps “c/d” navigate into that unit. On the other hand, if the unit identified by “b” does contain nested units, than “c” identifies a unit within “b”, and the interpretation of the next step “d” again depends on the type of the unit identified by the preceding step. In summary, the unit selection comprises one or more steps, and it ends with the first step identifying a “primary unit”, a unit not containing other units. If this step is not the last step of the info path, the remaining steps constitute an intra-unit path; otherwise, the path identifies that unit as a whole, rather than parts of it.

As already mentioned, the semantics of the intra-unit path depend on the unit’s physical type. The following table gives a few examples.

Table 3.

Interpretation of intra-unit paths - examples.

Unit type (physical)	**Interpretation of the intra-unit path**
node	XPath
string-to-string map	step 1: map key; no further steps allowed
string-to-node map	step 1: map key; remaining steps: XPath
string-to-(string-to-string map)	step 1: outer map key; step 2: inner map key
string-to-(string-to-node map)	step 1: outer map key; step 2: inner map key; remaining steps: XPath

Unit type (physical)	Interpretation of the intra-unit path
properties	property name
string	not applicable
date	not applicable

Note on multiplicity. The unit to which the intra-unit path must be applied can be a sequence of objects, rather than a single object. There are two sources of multiplicity: (i) several units can have the same name, and (ii) the physical type may be an object array, rather than a single object. If the intra-unit path is applied to more than one object, the evaluation is repeated for each one and the results are united.

Note on step indexes and unit selection. For steps dedicated to unit selection, a special index syntax applies (for example: a/b[2.3]/c), allowing to target both sources of multiplicity (see above) separately.

Class InfoSpace

The XQJPLUS class `InfoSpace` represents an info space. Besides methods for adding and removing trays from the space, it offers several methods for evaluating info paths, distinguished by the result type (string, string array, node, node array), as well as a method for setting/extending the namespace context for info path evaluation. An `InfoSpace` object represents a collection of information which combines high flexibility (and possibly great heterogeneity) of information models with a unified, string-based access model. This makes `InfoSpace` objects an option for the design of highly configurable software: configurations supply info path strings which connect generic functionality to actual input collected from info spaces. As well as this, unified addressability of data supports data integration in general. Just as XPath provides this addressability for XML data, info path provides this addressability for info space data.

The cooperation of Java and XQuery developers

XQJPLUS is designed to facilitate the cooperation of the Java and XQuery developers. Typically, the cooperation between developers is based on interfaces. The “interface” to the XQuery developer’s work is a tray design, which can be formally captured by a tray schema. Optionally, this tray schema can in turn be compiled into a set of Java interfaces. A suggested workflow is:

1. The Java and XQuery developers discuss the information to be extracted from a set of XML resources.
2. They design the tray and document it using a *tray schema*.
3. The tray schema is compiled into an *info shape*, a Java class and interfaces.
4. The XQuery developer writes a prototype query which produces a valid tray.
5. The Java developer can begin to code against the info shape.
6. The XQuery developer produces incremental versions of the query, eventually replacing the prototype with a real implemention.

Reusing XQJPLUS in other integration infrastructures

XQuery is integrated by XQJPLUS in such a way as to maximize the possibility of reuse when developing other integration software that is based on information units. The prospect of reuse exists

on three levels: changing the underlying XQuery API (currently: XQJ), changing the type of information resource (currently: XQuery) and changing the programming language integrating the information resource (currently: Java).

Replacing the XQJ API

XQJPLUS is based on XQJ. However, it would be easy to create a version based on a different XQuery API, e.g. Saxon's s9api S9API. Only the tray loading method would have to be adapted, and in doing so the only part that would need to be changed is the "extraction" of a Java data item from the Java representation of an XDM item.

Replacing XQuery

XQJPLUS integrates XQuery. A similar integration of other information resources, e.g. XSLT or XProc results, could reuse a major part of the software, namely the `InfoTray` class. In fact, only a new subclass of `InfoTray` would have to be created, which implements the loading of a tray from the chosen type of information resource. In particular note that the generation of Java code (info shapes) does not need to be changed, as it is independent of the information resource underlying the tray. The major problem to be solved would be to control the assembly of information units: the control exerted by XQuery result items might prove difficult to attain with other sources.

Replacing Java

The integration of XQuery into other languages – e.g. C# or C++ - could be modelled on XQJPLUS. First, produce a language-specific version of `InfoTray`. Then implement the loading of the tray from an XQuery result, using the `InfoTray`'s generic `add` methods. Finally, extend the tray schema language by a new language binding vocabulary, and implement the code generation controlled by that vocabulary.

Information units versus data binding

Information units are read from an info tray or an info shape. In both cases, "getter" methods are used to obtain data originating from XML resources. This is an apparent similarity to data binding. But there are two fundamental differences.

evaluation versus repetition

Info trays and shapes represent evaluation results, whereas the objects created by data binding represent the original resources, and echo their structure. Data binding essentially repeats information, rather than evaluates it.

information units versus items

As data binding reflects the original information content, it also reflects its structure as a tree of atomic items. Information units, on the other hand, are entities designed to support the Java code in an optimal way, free to use alternatives to atomic values and trees.

These differences can be made less obvious by the many forms of customizing which different data binding technologies support. They introduce limited possibilities to avoid a one-to-one repetition of structure and information content. But this amounts to variations on the theme of repetition, rather than a transition from repetition to evaluation. This becomes especially clear when considering the price to be paid for any deviation from the structure and content implied by the original resource.

If the main interest lies in accessing single items that can be identified by paths known at compile time – for example extracting parameters from a SOAP request or a configuration - data binding offers considerable advantages to the Java developer, whereas information units are of very limited use. But evaluating the XML resources and creating new information from them is an activity data binding is little concerned with. If such evaluation is important, XQuery in general, and XQuery-powered information units in particular offer an option that should be considered.

Conclusion

The integration of XQuery into Java may be viewed as a specimen of a very general kind of integration: the integration of “pure information processing” into object-oriented languages. Here, “pure information processing” denotes a side-effect free processing of information. As mentioned earlier, the W3C-developed XDM is a unified information model applicable to a vast spectrum of information, accessible to processing languages marked by precision, conciseness and efficiency. As a matter of fact, “information processing”, as it has become possible thanks to the work done by W3C, is a new kind of program activity, which can be distinguished from other activities conceptually, and structurally factored out. It is a promising perspective to practise such separation where feasible, factoring out information processing, delegating it to specialized components – e.g. XQuery code – in as large chunks as possible, and integrating the results in a seamless way. This is an architectural pattern which XQJPLUS strives to support.

Acknowledgements

David A. Lee gave me advice concerning the technical preparation of the manuscript, and I-Lin Kuo checked my use of the NWIS Water Quality Web Services. Lauren Wood helped me generously with the English text, sacrificing much time to the elimination of errors and awkward phrasing. Xia Li helped me in every way: taking part in the technical preparation of the manuscript, reading drafts, discussing issues, making many valuable suggestions and simply encouraging me with her vivid, tireless interest. To all of you - thank you!

A. Sample of the data evaluated in the example section

The data evaluated in the section called “Getting started with XQJPLUS - an illustrative example” were obtained from the Result Service of “National Water Information System (NWIS) Water-Quality Web Services” USGS Web Services. They contain water quality sampling results reported for the first three months of 2010, collected in the state of Minnesota, including both final and preliminary results. The data can be obtained from the following URI:

```
http://qwwebservices.usgs.gov/Result/search?organization=USGS-MN&startDateLo=
01-01-2010&startDateHi=03-31-2010&mimeType=xml
```

Here comes a small sample of the data. "..." indicate left out parts.

```
<WQX xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="http://qwwebservices.usgs.gov/schemas/WQX-Outbound/2_0/"
     xsi:schemaLocation="http://qwwebservices.usgs.gov/schemas/WQX-Outbound/2_0/
     http://qwwebservices.usgs.gov/schemas/WQX-Outbound/2_0/index.xsd">
   <Organization>
      <OrganizationDescription>
         <OrganizationIdentifier>USGS-MN</OrganizationIdentifier>
         <OrganizationFormalName>USGS Minnesota Water Science Center</Organizati
      </OrganizationDescription>
      ...
      <Activity>
         <ActivityDescription>
            <ActivityIdentifier>sun1dmnspl.01.01000101</ActivityIdentifier>
            <ActivityTypeCode>Sample-Routine</ActivityTypeCode>
            <ActivityMediaName>Water</ActivityMediaName>
```

```
      <ActivityMediaSubdivisionName>Surface Water</ActivityMediaSubdivisi
      <ActivityStartDate>2010-01-20</ActivityStartDate>
      <ActivityStartTime>
         <Time>09:20:00</Time>
         <TimeZoneCode>CST</TimeZoneCode>
      </ActivityStartTime>
      <ProjectIdentifier>NAWQA</ProjectIdentifier>
      <ActivityConductingOrganizationText>U.S. Geological Survey-Water Re
      Discipline</ActivityConductingOrganizationText>
      <MonitoringLocationIdentifier>USGS-05288705</MonitoringLocationIden
      <ActivityCommentText>A-0210094 Attention: Charlie Patton TPCN (2) T
      each</ActivityCommentText>
      <HydrologicCondition>Stable, low stage</HydrologicCondition>
      <HydrologicEvent>Under ice cover</HydrologicEvent>
   </ActivityDescription>
   <SampleDescription>
      <SampleCollectionMethod>
         <MethodIdentifier>15</MethodIdentifier>
         <MethodIdentifierContext>USGS parameter code 82398</MethodIdenti
         <MethodName>Equal width increment, non-isokinetic</MethodName>
      </SampleCollectionMethod>
      <SampleCollectionEquipmentName>Open-Mouth Bottle</SampleCollectionE
   </SampleDescription>
   <Result>
      <ResultDescription>
         <CharacteristicName>Depth</CharacteristicName>
         <ResultMeasure>
            <ResultMeasureValue>0.30</ResultMeasureValue>
            <MeasureUnitCode>ft</MeasureUnitCode>
         </ResultMeasure>
         <ResultStatusIdentifier>Preliminary</ResultStatusIdentifier>
         <ResultValueTypeName>Actual</ResultValueTypeName>
         <USGSPCode>81903</USGSPCode>
      </ResultDescription>
      <ResultLabInformation>
         <LaboratoryName>U.S. Geological Survey-Water Resources Disciplin
         <AnalysisStartDate>--</AnalysisStartDate>
      </ResultLabInformation>
      <LabSamplePreparation>
         <PreparationStartDate>--</PreparationStartDate>
      </LabSamplePreparation>
   </Result>
   <Result>
      <ResultDescription>
         <CharacteristicName>Nitrite</CharacteristicName>
         <ResultSampleFractionText>Dissolved</ResultSampleFractionText>
         <ResultMeasure>
            <ResultMeasureValue>0.087</ResultMeasureValue>
            <MeasureUnitCode>mg/l</MeasureUnitCode>
         </ResultMeasure>
         <ResultStatusIdentifier>Preliminary</ResultStatusIdentifier>
         <ResultValueTypeName>Calculated</ResultValueTypeName>
         <USGSPCode>71856</USGSPCode>
      </ResultDescription>
      <ResultLabInformation>
         <AnalysisStartDate>--</AnalysisStartDate>
      </ResultLabInformation>
      <LabSamplePreparation>
```

```
                  <PreparationStartDate>--</PreparationStartDate>
               </LabSamplePreparation>
            </Result>
            <Result>
               <ResultDescription>
                  <CharacteristicName>Nitrate</CharacteristicName>
                  <ResultSampleFractionText>Dissolved</ResultSampleFractionText>
                  <ResultMeasure>
                     <ResultMeasureValue>1.66</ResultMeasureValue>
                     <MeasureUnitCode>mg/l</MeasureUnitCode>
                  </ResultMeasure>
                  <ResultStatusIdentifier>Preliminary</ResultStatusIdentifier>
                  <ResultValueTypeName>Calculated</ResultValueTypeName>
                  <USGSPCode>71851</USGSPCode>
               </ResultDescription>
               <ResultLabInformation>
                  <AnalysisStartDate>--</AnalysisStartDate>
               </ResultLabInformation>
               <LabSamplePreparation>
                  <PreparationStartDate>--</PreparationStartDate>
               </LabSamplePreparation>
            </Result>
            <Result>
               <ResultDescription>
                  <CharacteristicName>Ammonia and ammonium</CharacteristicName>
                  <ResultSampleFractionText>Dissolved</ResultSampleFractionText>
                  <ResultMeasure>
                     <ResultMeasureValue>0.49</ResultMeasureValue>
                     <MeasureUnitCode>mg/l NH4</MeasureUnitCode>
                  </ResultMeasure>
                  <ResultStatusIdentifier>Preliminary</ResultStatusIdentifier>
                  <ResultValueTypeName>Calculated</ResultValueTypeName>
                  <USGSPCode>71846</USGSPCode>
               </ResultDescription>
               <ResultLabInformation>
                  <AnalysisStartDate>--</AnalysisStartDate>
               </ResultLabInformation>
               <LabSamplePreparation>
                  <PreparationStartDate>--</PreparationStartDate>
               </LabSamplePreparation>
            </Result>
            ...
         ...
      </Organization>
   </WQX>
```

B. XQuery code performing the evaluations described in the example section

The query performed the evaluations described in the section called “Getting started with XQJPLUS - an illustrative example”. It produces a sequence of control items (`<xqjp:XXX>`) and data items, which generic Java code (XQJPLUS) assembles into an info tray.

```
declare namespace xqjp="http://www.bits-ac.com/xqjplus/control";
declare namespace meta="http://www.bits-ac.com/xqjplus/meta";
```

```
declare default element namespace "http://qwwebservices.usgs.gov/schemas/WQX-Ou

declare variable $projectIds :=
   for $n in distinct-values(//ActivityDescription/ProjectIdentifier)
   order by $n return $n;

(: *** namespace context ***
 : ======================== :)
<xqjp:setNamespaceContext>
   <xqjp:binding prefix="data" uri="http://qwwebservices.usgs.gov/schemas/WQX-O
   <xqjp:binding prefix="meta" uri="http://www.bits-ac.com/xqjplus/meta"/>
   <xqjp:binding prefix="usgs" uri="http://www.bits-ac.com/usgs"/>
</xqjp:setNamespaceContext>,

(: *** tray meta data ***
 : ===================== :)
<xqjp:trayMeta meta:name="balisageDemoTray"
               meta:documentation="Creates the info tray used in section 'Getti

(: *** info unit "projectIds"
 : ========================= :)
<xqjp:part name="projectIds"
           type="string_sortedset_object"
           sematype="usgs:projectIds"/>,
$projectIds,

(: *** info unit "projectActivities"
 : ================================ :)
<xqjp:part name="projectActivities"
           type="nested_map_string_to_string_object"
           impltype="java.util.TreeMap"
           sematype="usgs:projectActivities"
           outerEntryPath="data:project"
           outerKeyPath="@id"
           innerEntryPath="data:activity"
           innerKeyPath="@time"
           innerValuePath="@activityId"/>,

<projectData>{
   for $p in $projectIds
   let $myActivities := //Activity[ActivityDescription/ProjectIdentifier eq $p]
   return
      <project id="{$p}">{
         for $a in $myActivities
         return
            <activity>{
               attribute time {$a/ActivityDescription/concat(ActivityStartDate,
               attribute activityId {$a/ActivityDescription/ActivityIdentifier}
            }</activity>
      }</project>
}</projectData>,

(: *** info unit "activityResults"
 : ============================ :)
<xqjp:part name="activityResults"
           type="map_string_to_custom_objects_object"
           impltype="java.util.TreeMap"
           customType="com.bits_ac.xqjplus.appl.UsgsResult"
```

```
            sematype="usgs:projectActivities"
            entryPath="data:activity"
            keyPath="@id"
            valuePath="data:Result"/>,

<resultObjects>{
   for $a in //Activity
   return
      <activity id="{$a//ActivityIdentifier}">{$a/Result}</activity>
}</resultObjects>,

(: *** info unit "fractionResults"
 : ============================== :)
<xqjp:part name="fractionResults"
           type="nested_map_string_to_strings_object"
           impltype="java.util.TreeMap"
           sematype="usgs:fractionResults"
           outerEntryPath="data:fractionResults"
           outerKeyPath="@name"
           innerEntryPath="data:entry"
           innerKeyPath="@time"
           innerValuePath="data:result/@uvalue"/>,

<allFractionResults>{
   for $f in distinct-values(//ResultDescription/CharacteristicName/string-join
   let $charName := replace($f, "/.*", "")
   let $fractionText := if (contains($f, "/")) then replace($f, ".*/", "") else
   let $myResults := //Result[ResultDescription/CharacteristicName eq $charName
                      not($fractionText) and not(ResultDescription/ResultSample
                      or ResultDescription/ResultSampleFractionText eq $fractio
   let $times :=
      for $t in distinct-values($myResults/../ActivityDescription/concat(Activi
      order by $t return $t
   order by lower-case($f)
   return
      <fractionResults name="{$f}">{
         for $time in $times
         let $myTimeResults :=
            $myResults[../ActivityDescription/concat(ActivityStartDate, "#", Ac
         order by $time
         return
            <entry time="{$time}">{
               for $r in $myTimeResults
               let $rvalue := $r/ResultDescription/ResultMeasure/(ResultMeasure
               let $rvalue :=
                  if ($rvalue) then $rvalue else $r/ResultLabInformation//(Meas
               return
                  <result uvalue="{concat($rvalue[1], " [", $rvalue[2], "]")}"/
            }</entry>
      }</fractionResults>
}</allFractionResults>
```

C. The control items defined by XQJPLUS

In the section called “The assembly process - basics” and the section called “The assembly process - advanced” several code examples show how XQuery code controls the construction of an info tray.

Those sections describe the main principles. This appendix provides a brief description of all control items defined by XQJPLUS.

xqjp:setNamespaceContext

When interpreting namespace-sensitive contents of control items - e.g. construction attributes containing XPath expressions, or attribute `sematype` which specifies the semantic type - XQJPLUS uses a special namespace context which is *not* the namespace context of the respective control item itself. This second namespace context is set or extended by `xqjp:setNamespaceContext` control items. Example:

```
<xqjp:setNamespaceContext>
   <xqjp:binding prefix="meta" uri="http://www.bits-ac.com/xqjplus/meta"/>
   <xqjp:binding prefix="usgs" uri="http://www.bits-ac.com/usgs"/>
   <xqjp:binding prefix="tns" uri="http://qwwebservices.usgs.gov/schemas/WQX-Ou
</xqjp:setNamespaceContext>
```

xqjp:part

The item defines an information unit. The name and physical type are given by the attributes “name” and “type”, and the semantic type and implementation type by “sematype” and “impltype”. Note that the latter can be used to control the type actually used to instantiate an interface, for example a collection interface. Construction attributes may control the assembly process; the range of construction attributes and their default values are specific to each physical type. A “source” attribute may reference the contents of a control variable as a data source. Further attributes must be in a namespace and specify the meta data of the unit. Child elements, if present, specify further meta data. Example:

```
<xqjp:part xmlns:bi="http://www.bits-ac.com/usgs"
           name="fractionResults"
           source="$fractionResults"
           type="nested_map_string_to_strings_object"
           sematype="bi:fractionResults"
           impltype="java.util.TreeMap"
           outerEntryPath="d:fractionResults"
           outerKeyPath="@name"
           innerEntryPath="d:entry"
           innerKeyPath="@time"
           innerValuePath="d:result/@uvalue"
           bi:source="mn-20100101-200330"
           bi:cr="{current-date()}"
>
   <laboratories>{string-join(//LaboratoryName, “; “)</laboratories>
</xqjp:part>
```

The defined unit has the name “fractionResults” (which has no namespace URI), a physical type corresponding to the Java type `Map<String,<String, String[]>>`, as well as semantic and implementation types as specified by the attributes “sematype” and “impltype”. The example specifies all five construction attributes defined for the actual physical type (“outerEntryPath”, “outerKeyPath”, “innerEntryPath”, “innerKeyPath”, “innerValuePath”) . If not specified, they would have defaulted to the values “*”, “@outerKey”, “*”, “@key” and “*”, respectively. As meta data, “bi:source”, “bi:cr” and “laboratories” are specified. Note that a special namespace treatment is applied to namespace-sensitive contents, e.g. to construction attributes which are defined to contain an XPath expression - see preceding section.

xqjp:setVar

The item defines a control variable. The name and variable type are given by the attributes “name” and “vtype”. The variable type must be one of: node, nodes, string, strings. Example:

```
<xqjp:setVar name="fractionResults" vtype="node"/>
```

xqjp:error

The item signals an error which makes normal tray loading impossible. The item specifies an error code and detail information, both of which will be used on the Java side when constructing an exception object. The details are represented by the control item's child elements. A special attribute "abort" specifies if the assembly process shall be continued or abandoned after processing this control item. (Continuation might be desirable in order to collect further error information.) Example:

```
<xqjp:error code="SOURCE_NOT_AVAILABLE" abort=”true”>
   <xqjp:sourceURI>foo.xml</xqjp:sourceURI>
</xqjp:error>
```

xqjp:trayMeta

The item provides tray-level meta data. The data can be specified by attributes or child elements, just like unit-level meta data are specified by an `<xqjp:part>` element. Note that the standard meta data "name" and "documentation" have names in the namespace {http://www.bits-ac.com/xqjplus/meta}. Example:

```
<xqjp:trayMeta smeta:name="bi:usgsResultEvaluation"
               bi:vsn="0.41"
               xmlns:smeta=”http://www.bits-ac.com/xqjplus/meta”
               xmlns:bi=”http://www.bits-ac.com/xqjplus/usgs”/>
```

xqjp:complexPartBegin

The item opens the definition of a unit whose physical type is either a single info tray or an array of info trays. The name of the unit is given by attribute "name", the physical type by attribute "type". (The physical type must be either "infotray" or "infotrays".) An attribute `sematype` may specify the semantic type, and further attributes and child elements may specify meta data of the unit in the same way as with `xqjp:part` items. Examples:

```
<xqjp:complexPartBegin name="msgEvaluations" type="infotrays"/>,
<xqjp:complexPartBegin name="msgEvaluation" type="infotray" sematype="ws:eval" |
```

xqjp:complexPartEnd

The item closes the definition of a unit which was opened by a preceding `<xqjp:complexPartBegin>` item. The unit closed is the innermost unit not yet closed. There are no attributes allowed. Example:

```
<xqjp:complexPartEnd />
```

Bibliography

[S9API] Michael Kay. Technical Documentation. http://www.saxonica.com/documentation/using-xquery/api-query.html.

[USGS Web Services] U.S. Geological Survey (USGS). Technical Documentation of USGS Water-Quality Web Services. http://qwwebservices.usgs.gov/technical-documentation.html .

[W3C Information Set] John Cowan and Richard Tobin, eds. XML Information Set W3C Recommendation 4 February 2004. http://www.w3.org/TR/xml-infoset/.

[W3C XDM] Mary Fernandez et al, eds. XQuery 1.0 and XPath 2.0 Data Model (XDM) W3C Recommendation 23 January 2007. http://www.w3.org/TR/xpath-datamodel/.

[W3C XQuery] Scott Boag et al, eds. XQuery 1.0: An XML Query Language W3C Recommendation 23 January 2007. http://www.w3.org/TR/xquery/.

[XQJ Spec] Jim Melton et al, eds. JSR 225: XQuery API for JavaTM (XQJ) 1.0 Specfication. http://jcp.org/en/jsr/detail?id=225.