This documentation describes the following libraries used in Quark:

- `xquery_model`
- `xquery_model_mm`
- `xquery_model_extended`

These libraries provide the underlying XQuery datamodel and function objects, to be used in parts of the YQGM graph.

A key feature of these libraries is the separation of the logical datamodel from the physical datamodel. A logical datamodel only provides the abstract interfaces required for the functions to operate on them, but does not specify how the data might be stored or represented. Representation of data is done by the physical datamodel.

`xquery_model` contains the abstract, or logical XQuery datamodel, whereas `xquery_model_mm` contains concrete implementations of the datamodel.

`xquery_model_extended` contains extended function objects and an abstract factory to instantiate these function objects. There is no `xquery_model_extended_mm`, because only additional functions are added to `xquery_model_mm`.

`xquery_model_extended` is used to implement functionality such as XPath expressions, that are not already defined in XQuery. It includes functions used in XPath expressions, like `translate()`.

1. Description of `xquery_model`

`xquery_model` contains the abstract datamodel of the elements that make up the XQuery model. It consists of the following sub-namespaces:

- `datamodel`
- `types`
- `functions`
- `factory`

Of the four, the `factory` sub-name is the only one that contains objects that make up the `xquery_model`. It contains a `factory` class to instantiate objects.

- `datamodel`
- `types`
- `functions`

This documentation describes the following libraries used in Quark: `xquery_model`, `xquery_model_mm`, `xquery_model_extended`. These libraries provide the underlying XQuery datamodel and function objects, to be used in parts of the YQGM graph.

A key feature of these libraries is the separation of the logical datamodel from the physical datamodel. A logical datamodel only provides the abstract interfaces required for the functions to operate on them, but does not specify how the data might be stored or represented. Representation of data is done by the physical datamodel.

`xquery_model` contains the abstract, or logical XQuery datamodel, whereas `xquery_model_mm` contains concrete implementations of the datamodel.

`xquery_model_extended` contains extended function objects and an abstract factory to instantiate these function objects. There is no `xquery_model_extended_mm`, because only additional functions are added to `xquery_model_mm`.

`xquery_model_extended` is used to implement functionality such as XPath expressions, that are not already defined in XQuery. It includes functions used in XPath expressions, like `translate()`.

1. Description of `xquery_model`

`xquery_model` contains the abstract datamodel of the elements that make up the XQuery model. It consists of the following sub-namespaces:

- `datamodel`
- `types`
- `functions`
- `factory`

Of the four, the `factory` sub-name is the only one that contains objects that make up the `xquery_model`. It contains a `factory` class to instantiate objects.
The `xquery_model::datamodel` contains the datamodel items used in XQuery. It is based on the datamodel hierarchy found in the "XQuery 1.0 and XPath 2.0 Data Model" document, found here: http://www.w3.org/TR/xpath-datamodel/.

Here is a brief overview of basic `xquery_model::datamodel` objects:

### Sequence
- The basic unit in XQuery. It is a tuple of Item objects.
- A Sequence is always "flat", i.e., a Sequence cannot be nested inside another Sequence object.

### Sequences
- A tuple of Sequence objects. It is used to represent arguments for an XQuery function. XQuery functions take in a tuple of Sequence objects, which are represented by a single Sequences object.

### ErrorSequence
- This is a special type of Sequence added by Quark. It indicates that the result of an XQuery function evaluation is actually an XQuery error.

### ValueSequence
- Another class added by Quark. This represents an actual Sequence that contains Items, as described in XQuery. This was added to distinguish a Sequence containing Items from an ErrorSequence containing an error.
- In this document, "Sequence" typically refers to ValueSequences.

### Item
- An Item could be either a Node or an AtomicValue.

#### Node
- Nodes represent the XML tree.
- Examples of Nodes are document nodes, element nodes, attribute nodes, text nodes, etc.

#### AtomicValue
- AtomicValues include strings, integers, date/time objects, etc.
- An AtomicValue usually has a type, and if the type is not specified or not known, its type would be `xdt:untypedAtomic`.

### XUnit
- The parent class of `datamodel::Sequence`, `types::SequenceType` and `datamodel::NodeID`.

Note that some datamodel objects such as date or time objects are not supported by Quark yet.

As mentioned earlier, the datamodel is purely abstract; it contains mostly abstract classes to be implemented by a concrete implementation such as `xquery_model_mm`. This way, the physical storage of data is independent of the properties of the data. The `xquery_model::function` function objects described in 1.3 could operate on the data using the abstract interfaces, without any knowledge of how the data is stored or represented.
1.1 XUnits

XUnits are a parent class of Sequence, SequenceType and NodeID so that the query evaluator, Quark, can extend certain operations to XUnits instead of just Sequences. In this way, it is possible to write an extended function that compares a Sequence to a SequenceType or a NodeID.

The Quark evaluator works on XUnits.

The NodeID is a unique identifier for each Node. In XQuery, every constructed node has its own unique NodeID, which is compared by the op:is-same-node() function. Atomic values, on the other hand, do not have unique NodeIDs.

1.1.2 ErrorSequence and ValueSequence

This was added by Quark to facilitate error handling. When a function evaluates to an error in the Quark evaluation model, it is necessary to have a mechanism for reporting the error. In this case, the ErrorSequence is used.

The use of ErrorSequences allows the evaluator to eliminate the ErrorSequence in the YQGM operators, when it determines that the predicate is true.

This is part of the mechanism for handling XQuery errors in Quark.

1.1.3 Numerics

Another addition to Quark is the Numeric parent class. It is the parent of Integer, XSFloat and Double classes. All numeric types are either built-in or created from these classes. The super class exposes numeric methods that are used to simplify the evaluation of mathematical functions. For example, some arithmetic operations like square root and division require specialized handling and operations that are not supported by the datamodel classes. In this case, Quark uses the types in the XQuery numerics classes to perform such operations.

1.2 xquery_model::types

Objects in this namespace are used for the typing system. Currently Quark only supports the predefined types and does not support user-defined or complex types.

The C++ class hierarchy for types is very similar to the C++ class hierarchy for the datamodel. Type-objects are designed to be singletons: they can only be created by the TypeManager class. Thus we could have multiple instances of, for example, an datamodel::String class, but they would all reference the same type xs::StringType object. This is ensured by making the types' constructors private, while adding the TypeManager as a friend class.
Unlike the datamodel, the type objects are concrete classes, and they are not intended to be derived from.

A diagram of the type hierarchy could be found here:
http://www.w3.org/TR/xquery-operators/

In XQuery, the only concrete built-in derived type is xs:integer, which is derived from xs:decimal. Currently, we do not have an xs:decimal datamodel class, though we do have the type class. A createDerivedByRestrictionType() method creates a DerivedDecimal type object.

1.1.1 SequenceType and XUnitType

Other type objects not described in the XQuery type hierarchy is SequenceType. A SequenceType is described in XQuery as a tuple of an Item type and an occurrence indicator, which describes the multiplicity of each Item in the Sequence. Sequence objects are declared by SequenceType objects. Sequence objects are immutable, they are serialized for typing and can be decoded into values.

Since Quark works with XUnits, there is also an XUnitType. An XUnitType has a getXUnitTypeKind() method, which returns which XUnit class the type describes. Currently, there are only three XUnitTypes: NodeIDType for NodeIDs, SequenceType for Sequences, and SequenceTypeType for SequenceTypes. Note that SequenceType is both an XUnit and an XUnitType.

Other classes in xquery_model::types are included for future use with XSLT, though as present they are not implemented.

1.2. xquery_model::functions

This class contains function objects that implement the XQuery functions.

All function objects are derived from the XQueryFunction class.

XQuery functions are functions that take one or more values, and return another value. They are called with an XUnit type at run-time, though they do not store any values. They store no state, and it does not matter which instance of the function is called, as long as they are of the same class.

Functions which can take multiple numbers of arguments (for example, xsl:substring) could take an object or a number of arguments. They implement the methods evaluateFunction(), getReturnType() and getSequenceType().

They return a return type, which is a function type object, which is a function type object, which is a function type object.

They may be expanded to support types such as SequenceType.
evaluateFunction() executes the implementation of the XQuery function. It is protected and is
exposed to callers through XQueryFunction::evaluate() which calls castSequences() on the input
sequences before passing them to the evaluateFunction() method. The
restrictions the type constraints are defined in
http://www.w3.org/TR/xquery20/

In Debug mode, the return values are also checked with type checking.

All the function object classes here are concrete; they do not much is the derived from two
concrete implementations. Rather, they work with the above interfaces presented in the
expected model and respective abstract interfaces (as shown in section 1.3). The same
trait applies to the function object classes presented in this section.

Note that the general xQueryFunction::evaluate() method checks if the input Sequences contain
an ErrorSequence, and if so, it simply returns the ErrorSequence without calling the
evaluateFunction() method.

In a concrete or any other package or function class, the caller (which is not the
xQueryFunction::evaluate() method) may throw exceptions or raise
errors. Thus the general evaluate() never throws exceptions (nor may these be
caught). In contrast, the xQueryFunction::evaluate() method may have exceptions, but does not
throw any exceptions.

1.3.1 Function Signatures

All functions have a static FUNCTION_NAME field. This is used to represent the name
of the function, as well as the namespace URI and local name of the function.

A FunctionSignature contains a FunctionName, which contains the namespace URI and local
name of the function (note: the namespace URI is always in the form of a QName, whereas
the local name is the actual name of the function).

The Arity could be fixed, meaning that the function takes in
precisely a fixed number of arguments, or it could represent any number of arguments, for
special functions such as fn:concat(), which could take in any number of arguments.

In this way, the FunctionSignature uniquely identifies the function, even if the same function
name could be used for two XQuery functions with different arities.

The FunctionSignature contains a fixed vector of XUnitTypes, describing the types of the
function arguments. For XQuery functions, the XUnitTypes are always instances of
SequenceTypes, but for extended functions they could include NodeIDType or
SequenceType.

Note that for functions that could take in any number of arguments, the minimum number of
arguments is the number of types given in the function signature, minus one. This allows one to
define types for the first few required arguments to a function, followed by any number of
arguments of a particular type. This is very expressive than the function signatures required for
XQuery, but is much more like the extended function

1. The datamodel factory creates all the datamodel items given their values. It is used in the function implementations to create the datamodel items (e.g., for exiting, for creating union values).

The function factory is used by other users of xquery_model to create the function objects.

1.5 The datamodel factory creates all the datamodel items given their values. It is used in the function implementations to create the datamodel items (e.g., for exiting, for creating union values).

The function factory is used by other users of xquery_model to create the function objects.

1.5 Other helper classes
There are several helper classes used in the datamodel and functions.

datamodel::Caster casts types from one atomic type to another. Its behavior is exactly like the XQuery type-cast, and would throw an exception if the casting could not be done.

datamodel::Comparator compares one atomic value to another. Its behavior is exactly like the value comparison operators (e.g., lt, le) defined in XQuery. When comparing two atomic values of different types, an exception would be thrown if the two types are not compatible.

functions::FunctionCallHelper performs the type-checking and type-casting used by the function implementations. It is used in xquery_model to perform the type-checking and type-casting used by the function implementations.

2. xquery_model_mm
This is the concrete main-memory implementation of xquery_model. It only really needs to extend the datamodel and functions::FunctionCallHelper.

2.2 xquery_model_mm::datamodel
This contains the concrete implementation of the datamodel items, stored in main-memory.

Most of the main-memory node items, such as MMElementNode and MMDocumentNode, contain constructors that implement various functionality, such as recursively converting from the Xerces DOM nodes (which we use for parsing XML files) to our own physical datamodel.
The _mm suffix indicates that the datamodel exists in main memory. This is done through the use of a MemoryContext, which would be cleared at the end of the query evaluation.

2.3 xquery_model_mm::factory

These contain the concrete implementations of the datamodel factory and the function factory. These are also used to implement certain functionality in the YQGM graph. For example, the create-xml() function is used by the Dependency rule in the rewrite.

3. xquery_model_extended

This contains function objects that are used by Quark in addition to the XQuery functions, and an abstract factory to instantiate them. They are also used to implement various functionality in the YQGM graph. For example, the exact-equal() function is used by the Decorrelate rule in the rewriter.

The extended function objects are used for various things that are implemented as functions in Quark but are implemented as objects in Quark. For example, the XPath expression is implemented as an extended function object.

Note that these functions are derived from xquery_model::ExtendedXQueryFunction, which allows them to be compiled and run as functions in the YQGM graph. For example, the isType() function takes in a Sequence and a SequenceType, and returns true if the former matches the latter.

There are three main sub-classes of ExtXQueryFunction: ScalarFunction, SuperScalarFunction, and AggregateFunction.

ScalarFunction takes in Sequences, and returns a Sequence. It is also used for TeXQuery functions.

SuperScalarFunction takes in a single Sequence, and returns one or more Sequences. It is also used for TeXQuery functions.

AggregateFunction does not have an evaluate() method. Rather, it has init() and accum() methods.

The SuperScalarFunction and AggregateFunction objects are used for YQGM.

There are no extended datamodel objects.

4. Extending the xquery_model

There are two types of possible extensions to xquery_model: adding a new concrete data representation, and adding new functions.

These types are extended class of ExtXQueryFunction: ScalarFunction, SuperScalarFunction, AggregateFunction.
4.1 Adding new concrete data representation

To add a new concrete data representation, the implementation inherits the various abstract classes from the datamodel, and implements the various abstract methods. The functions would automatically work, since they only rely on the getter methods in the datamodel classes to perform their operations.

The datamodel factory would also have to be inherited, in order to create the new concrete datamodel items.

Finally, the new datamodel factory would be passed to the xquery_yqgm::XQueryYQGMFactory, so that it would create the new datamodel items instead of the _mm ones.

4.2 Adding new functions

This is somewhat simpler. The new function classes would have to be created, inheriting either from XQueryFunction or ExtXQueryFunction depending on whether they work only with Sequences or with XUnits.

A new function map class has to be created. This could be done by inheriting from the FunctionMap template in xquery_model::functions, and adding the functions to the constructor of the map.

Next, a new function factory class is created to wrap around the new function map.

Finally, the xquery_yqgm::XQueryYQGMFactory class has to be updated, so that it would also use the new function factory as well.