Dolby ADM Library

This document explains the design and implementation of the Dolby ADM library, a reusable code module written in C++ and C for generating, parsing and using the immersive audio portions of the Audio Definition Model. The ADM library has replaced the PMD Library's original ADM module, starting with the 2.0.0 release.

Motivation

Dolby's original ADM implementation was bundled inseparably with our implementation of our own Professional Metadata (PMD) system. The ADM implementation was grafted onto the PMD codebase in a later stage of development, and the system uses a common internal representation for both metadata formats. This representation was designed for PMD, which is a simpler format than ADM. When ADM is ingested, some information is lost, and it is not possible to regenerate exactly the same ADM on output. This complicates verification, and confuses users who are expecting to get out what they put in.

In addition, the original ADM implementation is opaque, incomplete, and has a number of bugs. The ADM standard is continuing to evolve, and we need to be able to maintain and extend our implementation to match that evolution. Programmers have found it very difficult to add or correct ADM parsing and generation in the v1.7.x codebase. The 2.0.0 implementation is considerably easier to maintain and extend.

An additional benefit of using the new ADM library is performance: it is more than three times faster than the prior implementation.

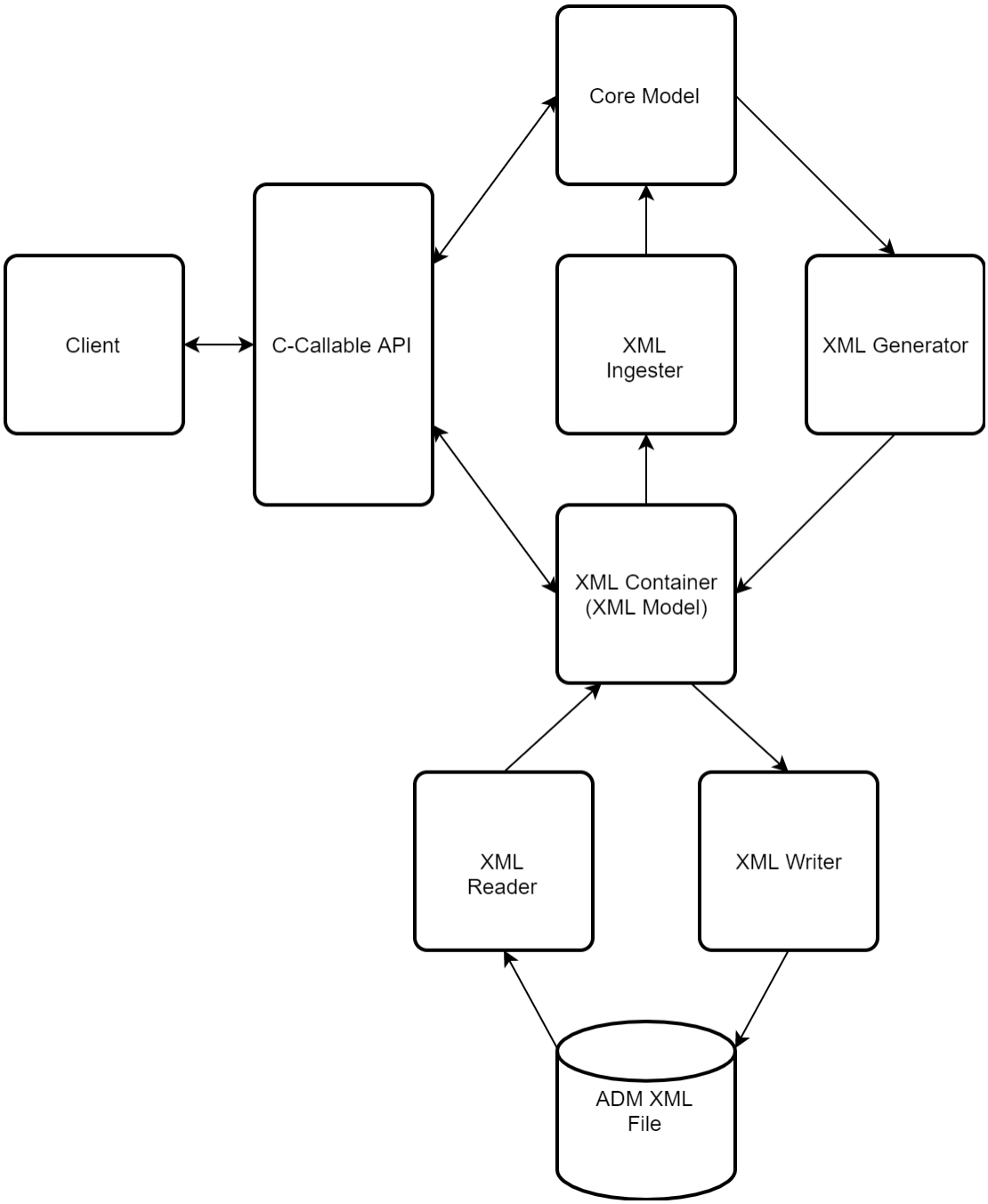
Design Philosophy

Our goal is to build a maintainable, extendable ADM system using efficient modern implementation methods and tools. Elements of the design philosophy to accomplish this goal include:

- Implementation in C++, with a C-callable external API.
- Appropriate use of well-established and reliable open-source code (boost).
- Table-based representation techniques, similar to database programming, instead of traditional C-style structures.
- Declarative programming, with a simplified core representation filled in with tables of data. This makes adding or correcting ADM elements as simple as adding or modifying entries in a table or two.

System Architecture

This block diagram is a simplified view of the ADM library components:



- Client processes interact with the ADM library via a C-callable API.
- The API interacts primarily with the XML Container and the Core Model.
- The XML Container contains the XML Model, which reflects the structure of the XML representation.
- The Core Model contains data which reflect the entities in the domain, for example, beds and objects.
- An XML Reader processes the external ADM representation in an XML file, adding elements to the XML Container.
- An XML Writer traverses the elements in the XML Container and writes the external ADM representation into an XML file.
- An XML Ingester traverses the XML elements in the XML Container and inserts domain elements into the Core Model.
- An XML Generator traverses the domain elements in the Core Model and inserts XML elements into the XML Container.

ADM XML Model

Conceptually, XML is a simple representational mechanism. There are elements, which may have internal attributes, which may have a value or may contain other elements in a tree structure. ADM is represented in XML, and adds one more concept: the reference. This is an element that references another element from a different part of the tree, via a unique identifier for each referenced entity. The reference allows the expression of relationships other than containment, and allows the expression of both one-to-many and many-to-many relationships. Here is a brief example:

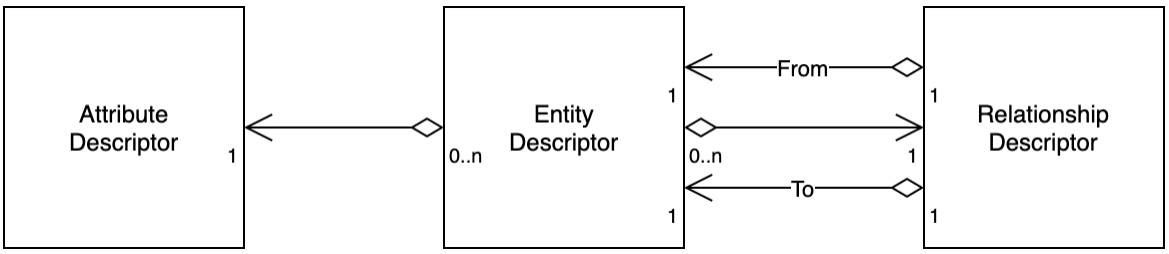
```
<audioProgramme audioProgrammeID="APR_1001" audioProgrammeName="English" audioProgrammeLanguage="en">
  <audioProgrammeLabel language="en">English</audioProgrammeLabel>
  <audioContentIDRef>ACO_1001</audioContentIDRef>
  <audioContentIDRef>ACO_1002</audioContentIDRef>
</audioProgramme>
<audioProgramme audioProgrammeID="APR_1002" audioProgrammeName="Francais" audioProgrammeLanguage="fr">
  <audioProgrammeLabel language="fr">Francais</audioProgrammeLabel>
  <audioContentIDRef>ACO_1001</audioContentIDRef>
  <audioContentIDRef>ACO_1003</audioContentIDRef>
</audioProgramme>
<audioContent audioContentID="ACO_1001" audioContentName="Main Stereo Bed">
  <audioObjectIDRef>AO_1001</audioObjectIDRef>
  <dialogue mixedContentKind="2">2</dialogue>
</audioContent>
<audioContent audioContentID="ACO_1002" audioContentName="English Dialog">
  <audioObjectIDRef>AO_1002</audioObjectIDRef>
  <dialogue dialogueContentKind="1">1</dialogue>
</audioContent>
<audioContent audioContentID="ACO_1003" audioContentName="Dialogue Francaise">
  <audioObjectIDRef>AO_1003</audioObjectIDRef>
  <dialogue dialogueContentKind="1">1</dialogue>
</audioContent>
```

There are two audioProgramme elements and three audioContent elements. Each one of these five elements has several attributes, for example, audioProgramme has audioProgrammeID and audioProgrammeName. Each one has at least one sub-element, and at least one reference. The audioProgramme with ID APR_1001 refers to the audioContent instances with IDs ACO_1001 and ACO_1002, while audioProgramme APR_1002 refers to audioContent ACO_1001 and ACO_1003.

This example is describing two presentations, one in English and one in French, each composed of the same stereo bed plus differing dialog objects. The connections between the major elements are represented by reference elements containing the unique identifier assigned to each major element. This allows the same entity to be referenced from multiple, unconnected locations, and enables single-pass, linear parsing of the ADM XML representation.

Declarative Representation of ADM Elements

The simplicity of the representational mechanism leads to simplicity of implementation. To represent the rules of construction for ADM XML elements, we need descriptors for three things: elements (entities), attributes and relationships.



Entities

ADM has a finite, known set of element types; the set may be expanded or modified as new versions of the specification are released, or as Dolby takes on additional ADM workflows or profiles. The element (entity) types are represented in an enumeration:

```
typedef enum
{
    DLB_ADM_ENTITY_TYPE_ILLEGAL = -1,
    DLB_ADM_ENTITY_TYPE_VOID,
    DLB_ADM_ENTITY_TYPE_TOPLEVEL,
    DLB_ADM_ENTITY_TYPE_XML,
    DLB_ADM_ENTITY_TYPE_FIRST = DLB_ADM_ENTITY_TYPE_XML,

    /* Entities with ADM identifiers */
    DLB_ADM_ENTITY_TYPE_FRAME_FORMAT,
    DLB_ADM_ENTITY_TYPE_FIRST_WITH_ID = DLB_ADM_ENTITY_TYPE_FRAME_FORMAT,
    DLB_ADM_ENTITY_TYPE_TRANSPORT_TRACK_FORMAT,
    DLB_ADM_ENTITY_TYPE_PROGRAMME,
    DLB_ADM_ENTITY_TYPE_CONTENT,
    DLB_ADM_ENTITY_TYPE_OBJECT,
    DLB_ADM_ENTITY_TYPE_PACK_FORMAT,
    DLB_ADM_ENTITY_TYPE_STREAM_FORMAT,
    DLB_ADM_ENTITY_TYPE_CHANNEL_FORMAT,
    DLB_ADM_ENTITY_TYPE_TRACK_FORMAT,
    DLB_ADM_ENTITY_TYPE_BLOCK_FORMAT,
    DLB_ADM_ENTITY_TYPE_ALT_VALUE_SET,
    DLB_ADM_ENTITY_TYPE_TRACK_UID,
    DLB_ADM_ENTITY_TYPE_LAST_WITH_ID = DLB_ADM_ENTITY_TYPE_TRACK_UID,

    /* Component entities (no ADM identifier) */
    DLB_ADM_ENTITY_TYPE_ITU_ADM,
    DLB_ADM_ENTITY_TYPE_CORE_METADATA,
    DLB_ADM_ENTITY_TYPE_FORMAT,
    DLB_ADM_ENTITY_TYPE_FRAME,
    DLB_ADM_ENTITY_TYPE_FRAME_HEADER,
    DLB_ADM_ENTITY_TYPE_CHANGED_IDS,
    DLB_ADM_ENTITY_TYPE_AUDIO_TRACK,
    DLB_ADM_ENTITY_TYPE_AUDIO_FORMAT_EXTENDED,
    DLB_ADM_ENTITY_TYPE_PROGRAMME_LABEL,
    DLB_ADM_ENTITY_TYPE_CONTENT_LABEL,
    DLB_ADM_ENTITY_TYPE_OBJECT_LABEL,
    DLB_ADM_ENTITY_TYPE_GAIN,
    DLB_ADM_ENTITY_TYPE_SPEAKER_LABEL,
    DLB_ADM_ENTITY_TYPE_DIALOGUE,
    DLB_ADM_ENTITY_TYPE_CARTESIAN,
    DLB_ADM_ENTITY_TYPE_POSITION,
    DLB_ADM_ENTITY_TYPE_EQUATION,
    DLB_ADM_ENTITY_TYPE_DEGREE,
    DLB_ADM_ENTITY_TYPE_ORDER,
    DLB_ADM_ENTITY_TYPE_NORMALIZATION,
    DLB_ADM_ENTITY_TYPE_FREQUENCY,
    DLB_ADM_ENTITY_TYPE_LAST = DLB_ADM_ENTITY_TYPE_FREQUENCY,

    /* How many distinct values in this enum? */
    DLB_ADM_ENTITY_TYPE_COUNT

} DLB_ADM_ENTITY_TYPE;
```

Information about an entity is represented using this struct:

```
    struct EntityDescriptor
    {
        std::string          name;              // Must be unique
        DLB_ADM_ENTITY_TYPE  entityType;
        bool                 xmlTypeComposite;
        bool                 hasADMIdOrRef;
        bool                 isReference;
        DLB_ADM_TAG          distinguishedTag;  // ID or value
    };
```

There is a list of EntityDescriptor instances, one for each ADM entity. At system startup, this list is loaded into a table. Here are a few examples of the entity descriptors:

```
    {
        "audioProgramme",
        DLB_ADM_ENTITY_TYPE_PROGRAMME,
        true,
        true,
        false,
        DLB_ADM_TAG_PROGRAMME_ID
    },
    {
        "audioContent",
        DLB_ADM_ENTITY_TYPE_CONTENT,
        true,
        true,
        false,
        DLB_ADM_TAG_CONTENT_ID
    },
    {
        "audioObject",
        DLB_ADM_ENTITY_TYPE_OBJECT,
        true,
        true,
        false,
        DLB_ADM_TAG_OBJECT_ID
    },
    {
        "audioTrackUID",
        DLB_ADM_ENTITY_TYPE_TRACK_UID,
        true,
        true,
        false,
        DLB_ADM_TAG_TRACK_UID_UID
    },
```

Relationships

There are two types of relationships between entities, containment and reference, plus their inverses. This is represented in an enumeration. An allowable relationship between instances of two entity types is described using these constructs:

```
    enum class ENTITY_RELATIONSHIP
    {
        NONE,
        CONTAINS,
        CONTAINED_BY,
        REFERENCES,
        REFERENCED_BY,
    };

    struct RelationshipArity
    {
        int     minArity;
        int     maxArity;

        static const int ANY;
    };

    struct RelationshipDescriptor
    {
        DLB_ADM_ENTITY_TYPE  fromType;
        DLB_ADM_ENTITY_TYPE  toType;
        ENTITY_RELATIONSHIP  relationship;
        RelationshipArity    arity;

        bool operator<(const RelationshipDescriptor &x) const;
    };
```

The RelationshipArity struct describes how many instances of the relationship may exist between two entities. For example, an audioProgramme entity may contain zero or more audioProgrammeLabel entities, so the relationship's arity is 0..ANY. An audioProgramme entity may reference many audioContent entities, but must reference at least one, so the relationship's arity is 1..ANY. An audioContent may contain at most one dialogue element, so the relationship's arity is 0..1.

The RelationshipDescriptor struct describes the relationship between two entity types. Only one relationship is allowed between two types, and all of the relationships are declared in a list, also loaded into a table at system startup. Here are some examples:

```
    // audioProgramme
    {
        DLB_ADM_ENTITY_TYPE_PROGRAMME,
        DLB_ADM_ENTITY_TYPE_PROGRAMME_LABEL,
        ENTITY_RELATIONSHIP::CONTAINS,
        { 0, RelationshipArity::ANY },
    },
    {
        DLB_ADM_ENTITY_TYPE_PROGRAMME,
        DLB_ADM_ENTITY_TYPE_CONTENT,
        ENTITY_RELATIONSHIP::REFERENCES,
        { 1, RelationshipArity::ANY },
    },

    // audioContent
    {
        DLB_ADM_ENTITY_TYPE_CONTENT,
        DLB_ADM_ENTITY_TYPE_CONTENT_LABEL,
        ENTITY_RELATIONSHIP::CONTAINS,
        { 0, RelationshipArity::ANY },
    },
    {
        DLB_ADM_ENTITY_TYPE_CONTENT,
        DLB_ADM_ENTITY_TYPE_OBJECT,
        ENTITY_RELATIONSHIP::REFERENCES,
        { 1, RelationshipArity::ANY },
    },
    {
        DLB_ADM_ENTITY_TYPE_CONTENT,
        DLB_ADM_ENTITY_TYPE_DIALOGUE,
        ENTITY_RELATIONSHIP::CONTAINS,
        { 0, 1 },
    },
```

Attributes

An ADM entity may have attributes. Each attribute in the system has a unique ID, or tag, represented in an enumeration:

Attribute tags

```
typedef enum
{
    DLB_ADM_TAG_UNKNOWN,

    /* xml */
    DLB_ADM_TAG_XML_VERSION,
    DLB_ADM_TAG_XML_ENCODING,

    /* ituADM */
    DLB_ADM_TAG_ITU_ADM_XMLNS,

    /* frameFormat */
    DLB_ADM_TAG_FRAME_FORMAT_ID,
    DLB_ADM_TAG_FIRST = DLB_ADM_TAG_FRAME_FORMAT_ID,
    DLB_ADM_TAG_FRAME_FORMAT_TYPE,
    DLB_ADM_TAG_FRAME_FORMAT_START,
    DLB_ADM_TAG_FRAME_FORMAT_DURATION,
    DLB_ADM_TAG_FRAME_FORMAT_TIME_REFERENCE,
```

```
DLB_ADM_TAG_FRAME_FORMAT_FLOW_ID,

/* transportTrackFormat */
DLB_ADM_TAG_TRANSPORT_TRACK_FORMAT_ID,
DLB_ADM_TAG_TRANSPORT_TRACK_FORMAT_NAME,
DLB_ADM_TAG_TRANSPORT_TRACK_FORMAT_NUM_IDS,
DLB_ADM_TAG_TRANSPORT_TRACK_FORMAT_NUM_TRACKS,

/* audioProgramme */
DLB_ADM_TAG_PROGRAMME_ID,
DLB_ADM_TAG_PROGRAMME_NAME,
DLB_ADM_TAG_PROGRAMME_LANGUAGE,

/* audioContent */
DLB_ADM_TAG_CONTENT_ID,
DLB_ADM_TAG_CONTENT_NAME,

/* audioObject */
DLB_ADM_TAG_OBJECT_ID,
DLB_ADM_TAG_OBJECT_NAME,
DLB_ADM_TAG_OBJECT_L_START,
DLB_ADM_TAG_OBJECT_L_DURATION,

/* audioTrackUID */
DLB_ADM_TAG_TRACK_UID_UID,
DLB_ADM_TAG_TRACK_UID_SAMPLE_RATE,
DLB_ADM_TAG_TRACK_UID_BIT_DEPTH,

/* audioPackFormat */
DLB_ADM_TAG_PACK_FORMAT_ID,
DLB_ADM_TAG_PACK_FORMAT_NAME,
DLB_ADM_TAG_PACK_FORMAT_TYPE_LABEL,
DLB_ADM_TAG_PACK_FORMAT_TYPE_DEFINITION,

/* audioStreamFormat */
DLB_ADM_TAG_STREAM_FORMAT_ID,
DLB_ADM_TAG_STREAM_FORMAT_NAME,
DLB_ADM_TAG_STREAM_FORMAT_FORMAT_LABEL,
DLB_ADM_TAG_STREAM_FORMAT_FORMAT_DEFINITION,

/* audioChannelFormat */
DLB_ADM_TAG_CHANNEL_FORMAT_ID,
DLB_ADM_TAG_CHANNEL_FORMAT_NAME,
DLB_ADM_TAG_CHANNEL_FORMAT_TYPE_LABEL,
DLB_ADM_TAG_CHANNEL_FORMAT_TYPE_DEFINITION,

/* audioTrackFormat */
DLB_ADM_TAG_TRACK_FORMAT_ID,
DLB_ADM_TAG_TRACK_FORMAT_NAME,
DLB_ADM_TAG_TRACK_FORMAT_FORMAT_LABEL,
DLB_ADM_TAG_TRACK_FORMAT_FORMAT_DEFINITION,

/* audioBlockFormat */
DLB_ADM_TAG_BLOCK_FORMAT_ID,
DLB_ADM_TAG_BLOCK_FORMAT_RTIME,
DLB_ADM_TAG_BLOCK_FORMAT_DURATION,

/* alternativeValueSet */
DLB_ADM_TAG_ALT_VALUE_SET_ID,

/* audioTrack */
DLB_ADM_TAG_AUDIO_TRACK_ID,

/* audioFormatExtended */
DLB_ADM_TAG_AUDIO_FORMAT_EXT_VERSION,

/* audioProgrammeLabel */
DLB_ADM_TAG_PROGRAMME_LABEL_VALUE,
DLB_ADM_TAG_PROGRAMME_LABEL_LANGUAGE,

/* audioContentLabel */
```

```
    DLB_ADM_TAG_CONTENT_LABEL_VALUE,
    DLB_ADM_TAG_CONTENT_LABEL_LANGUAGE,

    /* audioObjectLabel */
    DLB_ADM_TAG_OBJECT_LABEL_VALUE,
    DLB_ADM_TAG_OBJECT_LABEL_LANGUAGE,

    /* gain */
    DLB_ADM_TAG_SPEAKER_GAIN_VALUE,
    DLB_ADM_TAG_SPEAKER_GAIN_UNIT,

    /* speakerLabel */
    DLB_ADM_TAG_SPEAKER_LABEL_VALUE,

    /* dialogue */
    DLB_ADM_TAG_DIALOGUE_VALUE,
    DLB_ADM_TAG_DIALOGUE_NON_DIALOGUE_KIND,
    DLB_ADM_TAG_DIALOGUE_DIALOGUE_KIND,
    DLB_ADM_TAG_DIALOGUE_MIXED_KIND,

    /* cartesian */
    DLB_ADM_TAG_CARTESIAN_VALUE,

    /* position */
    DLB_ADM_TAG_POSITION_VALUE,
    DLB_ADM_TAG_POSITION_COORDINATE,
    DLB_ADM_TAG_POSITION_SCREEN_EDGE_LOCK,

    /* equation */
    DLB_ADM_TAG_EQUATION_VALUE,

    /* degree */
    DLB_ADM_TAG_DEGREE_VALUE,

    /* order */
    DLB_ADM_TAG_ORDER_VALUE,

    /* normalization */
    DLB_ADM_TAG_NORMALIZATION_VALUE,

    /* frequency */
    DLB_ADM_TAG_FREQUENCY_VALUE,
    DLB_ADM_TAG_FREQUENCY_TYPE_DEFINITION,

    DLB_ADM_TAG_LAST = DLB_ADM_TAG_FREQUENCY_TYPE_DEFINITION

} DLB_ADM_TAG;
```

All attributes in XML have values of type string, but it is convenient for us to have a set of value types with automatic translation to and from strings:

```
typedef enum
{
    DLB_ADM_VALUE_TYPE_BOOL,
    DLB_ADM_VALUE_TYPE_UINT,
    DLB_ADM_VALUE_TYPE_INT,
    DLB_ADM_VALUE_TYPE_FLOAT,
    DLB_ADM_VALUE_TYPE_AUDIO_TYPE,
    DLB_ADM_VALUE_TYPE_TIME,
    DLB_ADM_VALUE_TYPE_STRING,
} DLB_ADM_VALUE_TYPE;

typedef uint8_t         dlb_adm_bool;
typedef uint32_t    dlb_adm_uint;
typedef int32_t     dlb_adm_int;
typedef float       dlb_adm_float;

typedef enum
{
    DLB_ADM_AUDIO_TYPE_NONE,
    DLB_ADM_AUDIO_TYPE_DIRECT_SPEAKERS,
    DLB_ADM_AUDIO_TYPE_MATRIX,
    DLB_ADM_AUDIO_TYPE_OBJECTS,
    DLB_ADM_AUDIO_TYPE_HOA,
    DLB_ADM_AUDIO_TYPE_BINAURAL,
    DLB_ADM_AUDIO_TYPE_LAST_STD = DLB_ADM_AUDIO_TYPE_BINAURAL,
    DLB_ADM_AUDIO_TYPE_FIRST_CUSTOM = 0x1000,
    DLB_ADM_AUDIO_TYPE_LAST_CUSTOM = 0xffff
} DLB_ADM_AUDIO_TYPE;

typedef struct
{
    uint8_t          hours;
    uint8_t          minutes;
    uint8_t          seconds;

    uint32_t         fraction_numerator;
    uint32_t         fraction_denominator;

} dlb_adm_time;
```

An attribute for an ADM entity is described in a struct:

```
    struct AttributeDescriptor
    {
        DLB_ADM_ENTITY_TYPE      entityType;
        std::string              attributeName;
        DLB_ADM_TAG              attributeTag;
        DLB_ADM_VALUE_TYPE       attributeValueType;

        bool operator<(const AttributeDescriptor &x) const;
    };
```

There is a list of attribute descriptors, which is loaded into a table at system startup. Here are some examples:

```
    /* audioProgramme */
    { DLB_ADM_ENTITY_TYPE_PROGRAMME, "audioProgrammeID",       DLB_ADM_TAG_PROGRAMME_ID,
DLB_ADM_VALUE_TYPE_STRING },
    { DLB_ADM_ENTITY_TYPE_PROGRAMME, "audioProgrammeName",     DLB_ADM_TAG_PROGRAMME_NAME,
DLB_ADM_VALUE_TYPE_STRING },
    { DLB_ADM_ENTITY_TYPE_PROGRAMME, "audioProgrammeLanguage", DLB_ADM_TAG_PROGRAMME_LANGUAGE,
DLB_ADM_VALUE_TYPE_STRING },

    /* audioContent */
    { DLB_ADM_ENTITY_TYPE_CONTENT, "audioContentID",   DLB_ADM_TAG_CONTENT_ID,   DLB_ADM_VALUE_TYPE_STRING },
    { DLB_ADM_ENTITY_TYPE_CONTENT, "audioContentName", DLB_ADM_TAG_CONTENT_NAME, DLB_ADM_VALUE_TYPE_STRING },
```

Adding a new entity

Adding a new entity to the ADM XML modeling system becomes a matter of adding additional entries to each of the relevant lists. No other programming is needed in order for the code to start reading and writing the new entity. Here is what needs to be done:

- Add a new value to the DLB_ADM_ENTITY_TYPE enumeration.
- Add a value to the DLB_ADM_TAG enumeration for each attribute of the new entity type.
- Add a new EntityDescriptor record to the initializer list.
- Add a new record to the RelationshipDescriptor initializer list for each relationship to other entity types.
- Add a new record to the AttributeDescriptor initializer list for each attribute of the new entity type.

Note

It is important to note at this point, the constructs we have described constitute metadata describing the rules of syntactic construction for ADM XML, hence the class naming scheme:

- EntityDescriptor
- AttributeDescriptor
- RelationshipDescriptor

Table Implementation Using boost::multi_index_container

boost is an open-source, peer-reviewed collection of portable C++ source libraries. The boost libraries are heavily based on templates and meta-programming. The ADM library makes extensive use of the multi_index_container as a high-performance table and index-based data structure. The multi_index_container has been developed and refined over the course of more than forty releases of the boost collection. It is often used in "big data" applications for high-speed data processing using C++, where SQL would be too awkward or slow.

A multi_index_container instantiation provides a collection, or bag, to contain instances of a C++ class or struct, plus one or more indices with different sorting and access semantics. The contained class must have copy semantics (minimally, a copy constructor and assignment operator). It is possible to write tremendously complex containers with this template mechanism. Fortunately, the containers in the ADM library are rather simple and straightforward. Here is an example:

```
#include "EntityDescriptor.h"

#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/member.hpp>

namespace DlbAdm
{
    using namespace boost::multi_index;

    struct EntityIndex_Name {};
    struct EntityIndex_Type {};

    typedef multi_index_container<
        EntityDescriptor,
        indexed_by<
            // Name index
            ordered_unique<tag<EntityIndex_Name>, member<EntityDescriptor, std::string, &EntityDescriptor::
name> >,
            // Entity type index -- not unique because entity and reference to entity use the same type
            ordered_non_unique<tag<EntityIndex_Type>, member<EntityDescriptor, DLB_ADM_ENTITY_TYPE,
&EntityDescriptor::entityType> >
        >
    > EntityIndex;

    typedef EntityIndex::index<EntityIndex_Name>::type EntityIndex_NameIndex;
    typedef EntityIndex::index<EntityIndex_Type>::type EntityIndex_TypeIndex;

        .
        .
        .
}
```

This defines a container called EntityIndex that holds instances of the class EntityDescriptor and has two indices:

- The first index is unique: it defines a complete ordering over all the instances in the container, and a query using this index will return at most one instance. The index may be accessed using its tag, which is the struct EntityIndex_Name. There is one element on which to sort, which is declared to be a member of the EntityDescriptor struct, of type std::string, and is the "name" member. The index will use the std::less() template

function for sorting, which requires the underlying type (std::string) to have a "less-than" comparison operator or function defined for it. Thus, we require every EntityDescriptor to have a unique name, and we can look up an EntityDescriptor by its name using this index.
- The second index is non-unique: it defines a partial ordering over the contents of the container, and a query using this index will return zero or more instances. This index sorts on the entityType instance variable of EntityDescriptor.

Because referring to the type of an index can be quite long-winded, we usually define shorter type names for the indices of a container, which is the purpose of the two typedef statements below the EntityIndex definition.

Queries

The multi_index_container is fashioned to look like and operate similarly to the container classes in what was originally the C++ Standard Template Library (STL), which is now part of the C++ language standard itself. Both the container and all of the indices have iterators. The container iterators are usually not very useful, as they only allow one to iterate over the contents in no particular order. The index iterators, on the other hand, are very useful; we use them in conjunction with the functions find(), lower_bound(), upper_bound(), and equal_range() to search the container for matching items.

For example, to find the EntityDescriptor record matching a particular name, we use this function:

```
int GetEntityDescriptor(EntityDescriptor &d, const std::string &name)
{
    if (theADMEntityIndex.size() == 0)
    {
        InitializeEntityIndex();
    }

    EntityIndex_NameIndex &index = theADMEntityIndex.get<EntityIndex_Name>();
    EntityIndex_NameIndex::iterator it = index.find(name);
    int status = DLB_ADM_STATUS_NOT_FOUND;

    if (it != index.end())
    {
        d = *it;
        status = DLB_ADM_STATUS_OK;
    }

    return status;
}
```

There is a singleton instance of an EntityIndex, which is initialized lazily. If it has no entries, we assume it has not been initialized and we do that. We obtain a reference to the index using the get<>() template function, instantiated with the tag of the index we want (in this case, EntityIndex_NameIndex). Because it is a unique index, we can safely use a single iterator and the find() function to do the lookup. find() will return an iterator pointing to the unique record matching the name, or a special value meaning "not there", which can be obtained by a call to the index's end() function. If a matching record was found, we copy it to the caller via the d argument to the function.

Here is another example, on the non-unique entity type index:

```
int GetEntityDescriptor(EntityDescriptor &d, DLB_ADM_ENTITY_TYPE eType)
{
    if (theADMEntityIndex.size() == 0)
    {
        InitializeEntityIndex();
    }

    EntityIndex_TypeIndex &index = theADMEntityIndex.get<EntityIndex_Type>();
    auto range = index.equal_range(eType);
    int status = DLB_ADM_STATUS_NOT_FOUND;

    if (range.first != range.second)
    {
        d = *range.first;
        if (++range.first == range.second)
        {
            status = DLB_ADM_STATUS_OK;
        }
        else
        {
            status = DLB_ADM_STATUS_NOT_UNIQUE;
        }
    }

    return status;
}
```

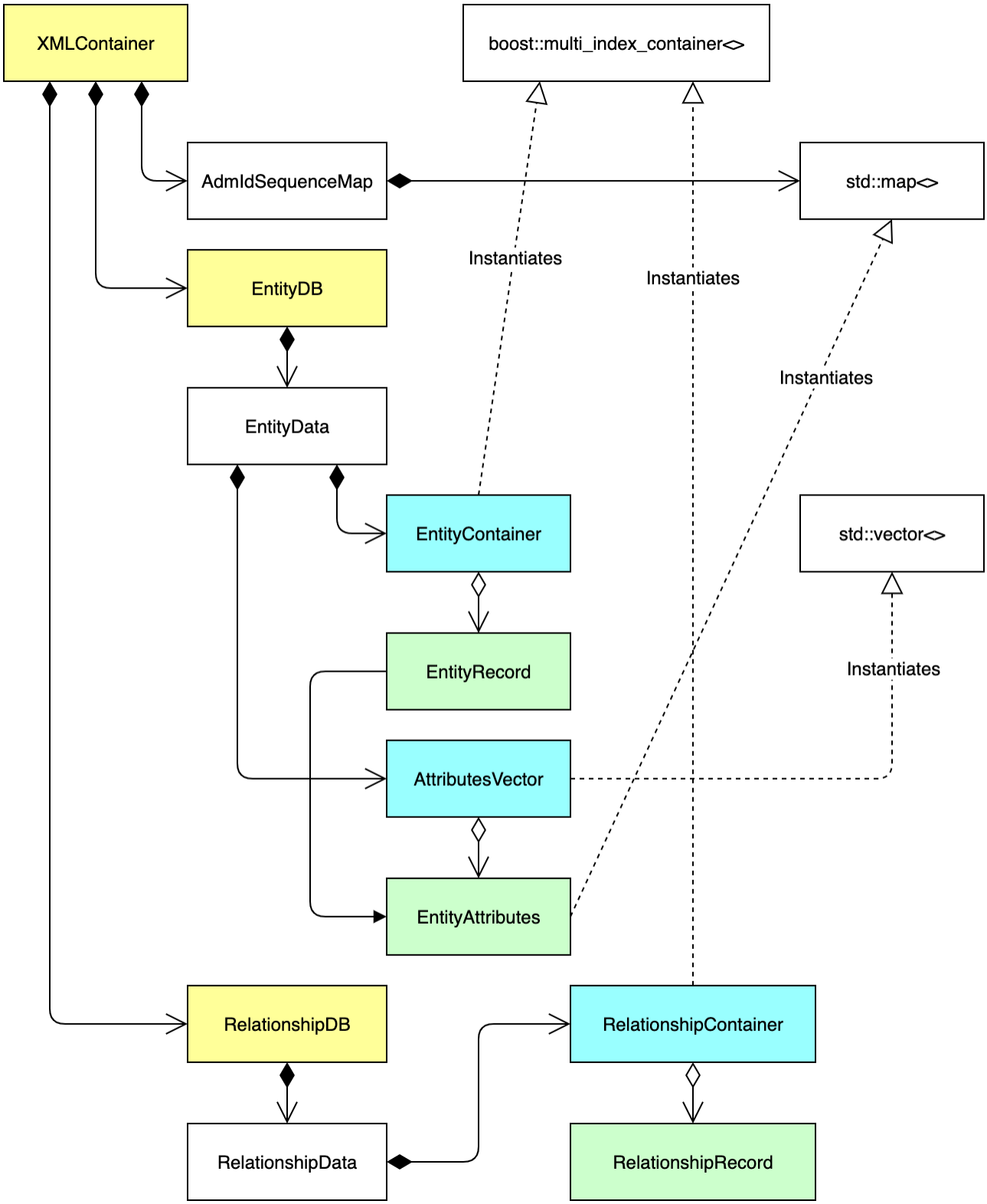
This overload of the GetEntityDescriptor() function will return one of three values: "not found", "OK", and "not unique", meaning "no matching record", "one matching record", and "two or more matching records" respectively. When there is just one, a copy is returned to the caller; when more than one, the first is copied and returned.

This time we get a reference to the entity type index via the EntityIndex_Type tag. Then we call equal_range() on the desired entity type value, which returns a pair of iterators. The type specification for this pair is long and complicated, so it's much easier just to use the C++ "auto" type and let the compiler figure it out. The first iterator of the pair will be positioned at the first (lowest) matching record, and the second will be positioned one beyond the last (highest) matching record. If there is no match, the two iterators will be equal.

In general, the container queries in the ADM library are wrapped in functions similar to these two examples. This gives more clarity to the desired semantics, without exposing the details of the multi_index_containers to the rest of the system.

Representation of ADM XML Content

Here we are changing our focus from describing the structure of the ADM XML, to describing the content of a particular ADM XML file. An instance of the class XMLContainer aggregates all the information contained in a single ADM file. This is a class diagram of XMLContainer and its related classes:



Briefly:

- XMLContainer has three main components:
  - AdmIdSequenceMap, which manages sequential numbering of entity identifiers.
  - EntityDB, which has records of entities and attributes:
    - EntityContainer:
      - Is a collection of EntityRecord instances, one per entity.
    - AttributesVector:
      - Is a collection of EntityAttributes instances, one per entity that has attributes:
        - EntityAttributes is a map between attribute tags and AttributeValue instances.

- RelationshipDB, which records all relationships between the entities:
    - RelationshipContainer:
        - Is a collection of RelationshipRecord instances, one for each relationship.

Unique Identifier for Entities

Each entity in an XMLContainer instance has a unique identifier. Some of the entity types are used in references, and require a representation for the unique identifier that encodes the identifier used in the XML, for consistent translation back and forth. It is convenient to have a similar ID for those entities not used in references, as we can then use it as the key for a row in a table describing that entity. We use a 64-bit unsigned integer to encode the IDs, as integer IDs speed up table insertion, indexing and queries.

The AdmIdTranslator class handles translation between the external, string representation and the internal, integer representation for those IDs having external representations. It also provides ways to construct a generic ID for other entity types, and to extract the entity type from an ID.

XMLContainer

XMLContainer is the class containing all the information about the XML representation of a particular ADM model. Typically, the content in an XMLContainer instance is the result of parsing an XML input file, or it is generated from the content in a Core Model instance. Theoretically, it is possible to generate a model in an XMLContainer instance by adding entities, attributes and relationships one at a time, but this is not an intended use for the class. XMLContainer has three main components: AdmIdSequenceMap, EntityDB and RelationshipDB.

AdmIdSequenceMap

This class manages sequential numbering of entities, and is composed of two maps:

- A map from an entity type to an unsigned integer, called the sequence map. It is used to record the largest sequence number used for an entity of a given type. When parsing an XML file, this will contain references to entity types that do not have externally-represented ADM identifiers. *Note: the AdmIdSequenceMap class is also used in the Core Model, to generate sequential identifiers for all entity types.*
- A map from an entity identifier to an unsigned integer, called the subcomponent map. Some ADM model entities, notably `audioChannelFormat`, have sequentially-numbered subcomponents. Similar to the sequence map, the subcomponent map records the largest number used for a subcomponent identifier for a particular parent component identifier.

The AdmIdSequenceMap class provides member functions to obtain the next sequence or subcomponent number for a particular entity type or parent identifier.

EntityDB

The EntityDB class, via its implementation class EntityData, records all the information for the XML entities in an ADM model. It has methods for adding and getting entities, getting attribute descriptors, setting and getting attribute values, traversing all entities of a given type, and traversing the attributes of a particular entity. It has two components: EntityContainer and AttributesVector.

EntityContainer

EntityContainer is a multi-index container holding instances of the EntityRecord type. It is searchable by entity identifier or type. EntityRecord holds the identifier, status, and index in the AttributesVector of the collection of attributes for the entity.

AttributesVector

This is simply a vector of EntityAttributes instances. The EntityAttributes type is a map from an attribute tag to an attribute value. Each entity records the index in the vector of its collection of attributes. The attribute values are kept outside the EntityContainer to avoid potentially expensive updates to the multi-index container when adding or changing attributes and values.

RelationshipDB

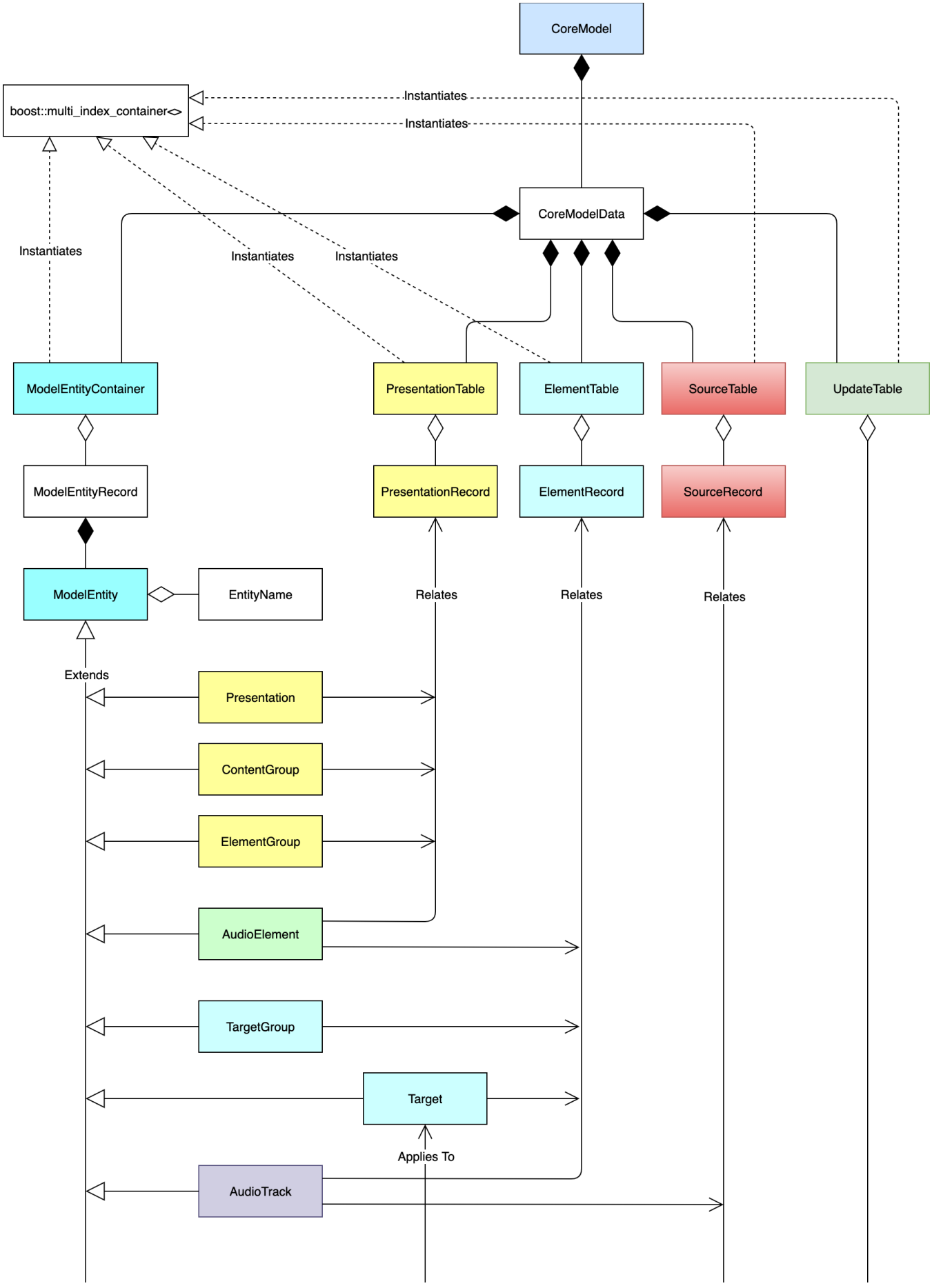
The RelationshipDB class, via its implementation class RelationshipData, records all the relationships between entities in an ADM model. There is one component, RelationshipContainer.

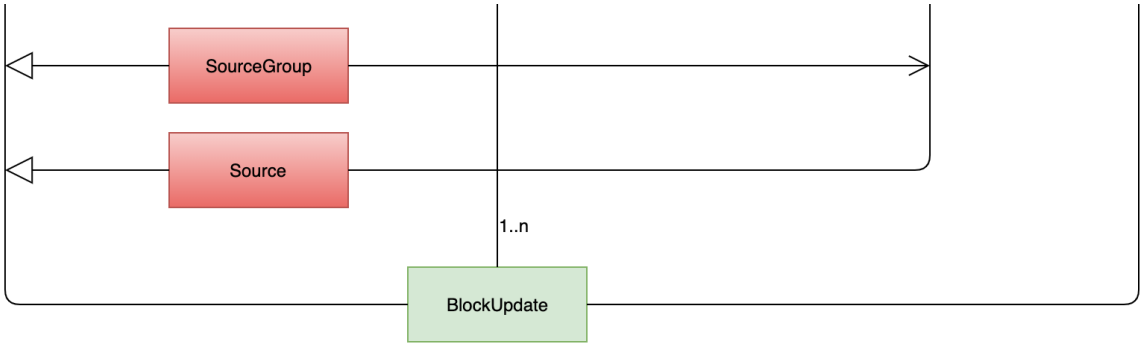
RelationshipContainer

This is the most complex multi-index container in the ADM Library. It has a composite key, which allows for matching partial keys when done from left to right. This allows for highly useful queries into the relationships between entities, such as, "give me all the relationships from this specific entity, for this particular relationship type", or, "give me all the relationships from this specific entity to other entities of a particular entity type."

Domain Model

The domain model is a set of classes and relationships that represent the information in an ADM model in the language of the underlying domain, object-based immersive audio. The base class for the entities in the domain model is called ModelEntity. There is also a container for the ModelEntity instances and relationship tables called CoreModel.





ModelEntity and its Subclasses

ModelEntity is the base class for the domain model entities. It has these responsibilities:

- Contains the unique identifier for the entity.
- Manages names and labels for the entity.
    - Several ADM elements have name attributes, sometimes there is also a language code indicating in which language the name is expressed.
    - Two ADM elements (audioProgramme and audioContent) may have "label" sub-elements, with name and language values.

All of the ModelEntity instances for a model are recorded in the ModelEntityContainer component of the CoreModel.

Presentation

The Presentation class represents a presentation in the domain, or the entire collection of audio information that forms a complete version of a program. An encoded audio stream may contain several presentations of a program, each containing a unique collection for a specific purpose. An instance of the Presentation class corresponds to an audioProgramme instance in an ADM model, and has the same unique identifier as the ADM model instance.

ContentGroup

The ContentGroup class represents a collection of related audio elements and groups. An instance of the ContentGroup class corresponds to an audioContent instance in an ADM model, and has the same unique identifier as the ADM model instance. It has this responsibility:

- Describes the nature of the content in the group (for example: music, dialog, etc.).

ElementGroup

The ElementGroup represents a group of audio elements to which a common gain value is applied. This is rather like a channel group controlled by a fader on a mixing console. An instance of the ElementGroup class corresponds to an audioObject instance in an ADM model, and has the same unique identifier as the ADM model instance. The audioObject instance may have references only to other audioObject instances, it may not have a reference to an audioPackFormat instance.

AudioElement

The AudioElement contains a bed element or an object element in the immersive audio domain, and a gain value to apply to the element. An instance of the AudioElement class corresponds to an audioObject instance in an ADM model, and has the same unique identifier as the ADM model instance. The audioObject instance may have references only to an audioPackFormat instance, it may not have references to other audioObject instances.

AudioTrack

The AudioTrack element describes a virtual track in the domain. It holds sample rate and bit depth attributes for the audio channel. It is the connection point between elements describing the format of the audio element and a Source element describing from where to acquire the actual audio. An instance of the AudioTrack class corresponds to an audioTrackUID instance in an ADM model, and has the same unique identifier as the ADM model instance.

TargetGroup

The TargetGroup element groups the Target elements for a bed or an object in the immersive audio domain. It has these responsibilities:

- Keeps track of whether the group represents a bed or an object.
- For a bed, keeps track of the speaker configuration.
- For an object, keeps track of the object class (dialog, voiceover, generic, etc.).

An instance of the TargetGroup class corresponds to an audioPackFormat instance in an ADM model, and has the same unique identifier as the ADM model instance.

Target

An instance of the Target element represents a single output channel. For a Target that is a member of a TargetGroup representing a bed, this will be a speaker position. For an object, it will be the object's output channel, and may or may not be able to have dynamic updates over time. An instance of the Target class corresponds to an audioChannelFormat instance in an ADM model, and has the same unique identifier as the ADM model instance.

SourceGroup

The SourceGroup class represents a group of related physical audio channels, such as a MADI interface. It aggregates a group of Source instances having the same interface ID. An instance of the SourceGroup class corresponds to a transportTrackFormat instance in a S-ADM model, and has the same unique identifier as the S-ADM model instance.

Source

An instance of the Source class represents a channel assignment from a particular input audio interface to one or more virtual track elements (AudioTrack instances). The instance keeps track of the source group ID (audio interface number) and the 1-based channel number in that interface. An instance of the Source class corresponds to an audioTrack instance in an ADM model.

BlockUpdate

An instance of the BlockUpdate class represents position, gain and other information about a Target instance. For a bed channel or a non-dynamic object, there is only one set of values applied across all of time. For a dynamic object, each instance will have start time and duration attributes defining the block of time to which it is applicable. An instance of the BlockUpdate class corresponds to an audioBlockFormat instance in an ADM model, and has the same unique identifier as the ADM model instance.

Domain Model Relationships

The domain model consists of ModelEntity instances, having attributes specific to each subclass, plus table-based representations of the relationships between the entity instances. These tables record the relationships using the unique identifiers for the instances, not the instances themselves. Each row in a table corresponds to a unique set of related entities.

PresentationTable

The PresentationTable has four columns. Each row in the PresentationTable describes the relationship between a presentation and an audio element, including the one or two intermediary objects. Here is an example:

```
<audioProgramme audioProgrammeID="APR_1001" audioProgrammeName="English" audioProgrammeLanguage="en">
  <audioProgrammeLabel language="en">English</audioProgrammeLabel>
  <audioContentIDRef>ACO_1001</audioContentIDRef>
  <audioContentIDRef>ACO_1002</audioContentIDRef>
</audioProgramme>
<audioContent audioContentID="ACO_1001" audioContentName="Main Stereo Bed">
  <audioObjectIDRef>AO_1001</audioObjectIDRef>
  <dialogue mixedContentKind="2">2</dialogue>
</audioContent>
<audioContent audioContentID="ACO_1002" audioContentName="English Dialog">
  <audioObjectIDRef>AO_1002</audioObjectIDRef>
  <dialogue dialogueContentKind="1">1</dialogue>
</audioContent>
<audioObject audioObjectID="AO_1001" audioObjectName="Main Stereo Bed">
  <gain gainUnit="dB">0.000000</gain>
  <audioPackFormatIDRef>AP_00011001</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000001</audioTrackUIDRef>
  <audioTrackUIDRef>ATU_00000002</audioTrackUIDRef>
</audioObject>
<audioObject audioObjectID="AO_1002" audioObjectName="English Dialog">
  <gain gainUnit="dB">0.000000</gain>
  <audioPackFormatIDRef>AP_00031002</audioPackFormatIDRef>
  <audioTrackUIDRef>ATU_00000003</audioTrackUIDRef>
</audioObject>
```

The "English" presentation for this program is represented in two rows of the PresentationTable:

Presentation	ContentGroup	ElementGroup	AudioElement
APR_1001	ACO_1001	null	AO_1001
APR_1001	ACO_1002	null	AO_1002

ElementTable

The ElementTable has four columns. Each row represents one channel of an audio element. A mono object will have one row in the table, and a bed will have one row for each channel in the speaker configuration. Here is an example:

```
    <audioObject audioObjectID="AO_1001" audioObjectName="Main Stereo Bed">
      <gain gainUnit="dB">0.000000</gain>
      <audioPackFormatIDRef>AP_00011001</audioPackFormatIDRef>
      <audioTrackUIDRef>ATU_00000001</audioTrackUIDRef>
      <audioTrackUIDRef>ATU_00000002</audioTrackUIDRef>
    </audioObject>
    <audioTrackUID UID="ATU_00000001">
      <audioChannelFormatIDRef>AC_00011001</audioChannelFormatIDRef>
      <audioPackFormatIDRef>AP_00011001</audioPackFormatIDRef>
    </audioTrackUID>
    <audioTrackUID UID="ATU_00000002">
      <audioChannelFormatIDRef>AC_00011002</audioChannelFormatIDRef>
      <audioPackFormatIDRef>AP_00011001</audioPackFormatIDRef>
    </audioTrackUID>
    <audioPackFormat audioPackFormatID="AP_00011001" audioPackFormatName="RoomCentric_2.0" typeLabel="0001"
typeDefinition="DirectSpeakers">
      <audioChannelFormatIDRef>AC_00011001</audioChannelFormatIDRef>
      <audioChannelFormatIDRef>AC_00011002</audioChannelFormatIDRef>
    </audioPackFormat>
    <audioChannelFormat audioChannelFormatID="AC_00011001" audioChannelFormatName="RoomCentricLeft" typeLabel="
0001" typeDefinition="DirectSpeakers">
                ...
    </audioChannelFormat>
    <audioChannelFormat audioChannelFormatID="AC_00011002" audioChannelFormatName="RoomCentricRight" typeLabel="
0001" typeDefinition="DirectSpeakers">
                ...
    </audioChannelFormat>
```

The "Main Stereo Bed" audio element has two rows in the ElementTable:

AudioElement	TargetGroup	Target	AudioTrack
AO_1001	AP_00011001	AC_00011001	ATU_00000001
AO_1001	AP_00011001	AC_00011002	ATU_00000002

SourceTable

The SourceTable has three columns. Each row represents the relationship between a SourceGroup instance, a Source instance and an AudioTrack instance. A SourceGroup will have a number of associated Sources, this represents the actual audio channels in a single audio interface. A virtual track (an AudioTrack instance) is connected to one actual track (a Source instance), and a number of virtual tracks may be connected to one actual track. Here is an example:

```
<transportTrackFormat transportID="TP_0001" transportName="Interface 1" numIDs="3" numTracks="2">
  <audioTrack trackID="1">
    <audioTrackUIDRef>ATU_00000001</audioTrackUIDRef>
  </audioTrack>
  <audioTrack trackID="2">
    <audioTrackUIDRef>ATU_00000002</audioTrackUIDRef>
    <audioTrackUIDRef>ATU_00000003</audioTrackUIDRef>
  </audioTrack>
</transportTrackFormat>
```

The "Interface 1" device is represented by three rows in the source table:

SourceGroup	Source	AudioTrack
TP_0001	1	ATU_00000001
TP_0001	2	ATU_00000002
TP_0001	2	ATU_00000003

UpdateTable

The UpdateTable has two columns. Each row represents the relationship between a Target instance and a BlockUpdate instance. Objects with dynamic update capability may have multiple rows, other targets will have only one. Here is an example:

```
    <audioChannelFormat audioChannelFormatID="AC_00011001" audioChannelFormatName="RoomCentricLeft" typeLabel="
0001" typeDefinition="DirectSpeakers">
      <audioBlockFormat audioBlockFormatID="AB_00011001_00000001">
        <speakerLabel>RC_L</speakerLabel>
        <gain gainUnit="dB">0.000000</gain>
        <cartesian>1</cartesian>
        <position coordinate="X">-1.000000</position>
        <position coordinate="Y">1.000000</position>
        <position coordinate="Z">0.000000</position>
      </audioBlockFormat>
    </audioChannelFormat>
```

This target represents the left speaker position of a stereo pair, and has one entry in the UpdateTable:

Target	BlockUpdate
AC_00011001	AB_00011001_00000001

CoreModel

The CoreModel class is the container for the elements of the domain model, and provides the C++ interface for interacting with an ADM model at the domain level. It is composed of the ModelEntityContainer and the four relationship tables, PresentationTable, ElementTable, SourceTable and UpdateTable.

What's Missing

- Logging
- Better error handling and notification (see "logging").
- External memory allocation.