Picture Metadata Toolkit

V 1.4

User's Guide

Document Version: *1.4*

Document Authors: *Dan Rupe, George Sotak, Ricardo Rosario*

Date: *November 19, 2003*

Table of Contents

Revision History

VERSION	DATE	AUTHOR	DESCRIPTION
0.1	8/28/2000	George Sotak	Initial Draft
0.2	9/27/2000	Diane J. Duda	Updates
0.3	10/11/2000	George Sotak	Many, many changes
0.8	11/06/2000	George Sotak	Updated to reflect changes in the API
1.0	11/10/2000	George Sotak	Final edits for release.
1.1	02/08/2001	George Sotak	Updates to correct errors and reflect changes in API.
1.2	03/30/2001	George Sotak	Updates to reflect changes since PMT V1.0.1
1.3	05/15/2001	George Sotak	Added section on instance serialization. Added section on copy all metadata from source to dest.
1.4	10/09/2001	Dan Rupe	Added PMT_READWRITE parameter to getAccessor call in "Copy All Metadata from Source to Destination" section.
2.0	11/04/2002	Dan Rupe	Major rewrite – most of document new.

2.1	11/11/2002	Dan Rupe	Minor updates.
1.1	2/25/2003	Ricardo Rosario	Updating for release 1.1. Now using all the OpenSource toolkits. Plus, support for Xerces 2. Note: Changed document version number to synchronize with PMT's version number.
1.2	5/15/2003	Ricardo Rosario	Updated for release v1.2.
1.3	8/16/2003	Ricardo Rosario	Updated for release v1.3
1.4	11/19/2003	Ricardo Rosario	Updated for release v1.4
1.4	11/20/2003	Sam Fryer	Minor Revisions

1. Introduction

1.1 Document Purpose & Scope

The purpose of this document is to provide the users of PMT with a comprehensive reference to the use of the PMT toolkit. This covers how to obtain, build, use, and extend PMT. This document is the first place to start when beginning to use PMT.

The PMT toolkit is implemented in the C++ language. Code examples in this document are in C++ syntax. PMT makes use of the Standard Template Library (STL). The reader is referred to [1] for information on the use of the STL.

Section 6 Compiling & Linking with PMT covers the steps required to set up your system to build with PMT.

1.2 PMT Features

The Picture Metadata Toolkit (PMT) is an object-oriented toolkit that provides functionality for the easy creation, manipulation, and persistence of image metadata. (Understanding what "image metadata" is will be covered in section 2.1 Metadata Overview.) This section covers some of PMT's features.

1.2.1 Standard Representation of Metadata

PMT presents metadata in a consistent fashion, through use of abstract PmtMetadata objects. The way an application deals with metadata is consistent, regardless as to the source of the metadata. For example, working with Exif vs. TIFF files is seamless to the PMT user. (**Exif** files are JPEG files – files typically with a **.jpg** extension – that contain metadata.) PMT's metadata objects are treated consistently inside your application, even if they originated from different types of image files.

1.2.2 File Format Details Handled by PMT

PMT eliminates the need for software packages to interface with specific file formats.

The persistence and use of metadata has been a barrier to imaging applications. There are multiple file formats in existence that persist metadata. If an application wants to use metadata from various file formats, it needs to deal with the different file formats itself. This can be a tedious task. Even if image file specific toolkits are used that handle those formats, the updating of toolkit versions and the use of new toolkits for new file formats (if even available) must be done. PMT helps these issues by providing the handling of file format details. One toolkit deals with all those details.

1.2.2.1 Easy Sharing of File Formats

There are many non-standard file formats that various applications use to store metadata. In order to use another application's metadata, the proprietary format of that metadata must be considered. It's difficult for metadata to be shared between different applications when it's stored in a proprietary fashion. PMT addresses this issue. When an application uses PMT to store its data, other applications can easily use the same metadata via PMT.

1.2.3 XML Enabled

XML and XML Schema support are natively built into PMT. PMT can always persist its metadata objects to XML instance documents.

Creating your own XML documents for use by PMT is a straightforward process.

1.2.4 Extendible - New File Formats Can be Handled

As time unfolds, new ways of persisting metadata will be progressively introduced. For example, the JPEG2000 file format is currently coming into use. PMT can be updated to provide access to JPEG2000 files. PMT has been designed to allow easy integration and usage of new file formats via an object-oriented file interface, called PmtAccessor.

In the case of XML files, no PMT code changes are required to support new file formats. Only a new XML Schema file need be provided to support persistence for new XML documents.

1.2.5 Automatic Memory Management

Since metadata objects in PMT are self-destructing, they can be easily passed from application to application (or library to library) without concern for memory clean up.

1.2.6 Open Source Standard – Helps Metadata Persistence And Use

Digital imaging has become a very open system. There are various stages a digital image can go through during its lifetime: capture, storage, processing, output, and transmission. Ensuring the persistence of metadata is essential to preserving the efforts placed into capturing it. Since PMT is an Open Source software project, the chances of it being adopted by the industry as a whole are enhanced. It is hoped that PMT will become widespread in use, and that many software packages will at least persist (and better yet, actively use) metadata. PMT makes metadata persistence and manipulation an easy task.

1.3 Supported Platforms

The following platforms, with the following C++ compilers, are supported by PMT.

- Windows - Compilers: Visual C++ 6 with SP5, Visual C++ 7.0, and Visual C++ 7.1
- Linux/Solaris UNIX - Compilers: GNU GCC version 2.95.2, version 2.95.3, and greater
- MacIntosh OS X – GNU GCC 3.2.2 or greater

1.4 Obtaining PMT

The PMT toolkit is publicly available for download at the following Web address:

http://sourceforge.net/projects/picturemetadata

PMT is an Open Source project and its source code is available under Open Source License. This license allows for toolkit usage in commercial application, and enables users to make publicly available contributions to the toolkit. Please refer to the file "license.html" in one of the downloadable distributions of PMT for license details.

2. PMT Concepts Overview

2.1 Metadata Overview

2.1.1 What is Metadata?

In the most general sense, metadata simply means "data about data". However, in the context of PMT, the term metadata refers to *image* metadata – data about (or associated with) an image. Image metadata can simplistically be thought of as all the non-pixel data associated with an image.

Examples of metadata can include camera settings such as aperture, shutter speed, or time and date of image capture. All instances of metadata are not necessarily created at the same time the image is. Metadata can also be generated after the image has been created. An example of this is an application that allows scene annotations or event descriptions to be added to images. Other examples of metadata include customer order information or processing information.

2.1.2 Why Use Metadata?

The use of metadata is very important in today's imaging world. There are endless possibilities for the use of image metadata, but here are few examples: it can be used by image scientists to improve the quality of images; it can be used by consumers to annotate their picture collections; metadata can be used to process film development orders; it can be used by software packages to improve the performance of image manipulation.

In short, the importance of metadata is escalating to a basic essential of dealing with images. Most, if not all, digital cameras record metadata. Metadata is an integral part of images.

2.2 PMT Metadata Objects

2.2.1 PmtMetadata Objects

PMT always uses metadata objects to present metadata to the PMT user. PMT metadata objects are typically associated with metadata items that have been obtained from a traditional image file, such as Exif or TIFF. All PMT metadata objects are of the type PmtMetadata. PmtMetadata is an abstract C++ class, defining the interface common to all metadata objects. PmtMetadata objects provide a consistent way in which an application deals with metadata.

Regardless as to the source of metadata (i.e. Exif, TIFF, or other) the metadata is always represented to a PMT user via a PmtMetadata object. This provides the desired abstraction from file-format details surrounding metadata.

PmtMetadata object instances can be one of two sub-class types: PmtCompositeMetadata or PmtMetadataT<TYPE>. The PmtCompositeMetadata and PmtMetadataT<TYPE> classes inherit from the PmtMetadata class.

2.2.2 PmtCompositeMetadata Objects

A PmtCompositeMetadata object is an object that can contain other PmtMetadata objects. It never contains a metadata value. PmtCompositeMetadata objects allow us to group other metadata objects into logical categories. PmtCompositeMetadata objects can contain other PmtCompositeMetadata objects or PmtMetadataT<TYPE> objects.

2.2.3 PmtMetadataT<TYPE> Objects

A PmtMetadataT<TYPE> object is an object that cannot contain other objects, but does contain a value associated with a particular metadata item. PmtMetadataT<TYPE> is a templated class, and its value can be one of several types, as stored in the class's <TYPE> type parameter.

2.2.4 Primarily Use PmtMetadata Interface

When a user wants to deal with metadata objects in general, the base PmtMetadata interface will be the primary interface of use. Applications never need to cast to the PmtCompositeMetadata type. Interactions with PmtCompositeMetadata can be handled exclusively through the PmtMetadata interface. And most dealings with PmtMetadataT<TYPE> objects can be handled through the PmtMetadata interface.

For dealing with values via string arguments, the base PmtMetadata class provides methods for inspecting or changing values on PmtMetadataT<TYPE> objects (getValueStr() and setValueStr()). Generally, using the interfaces of the PmtCompositeMetadata or PmtMetadataT<TYPE> classes is discouraged. Applications are encouraged to use the base PmtMetadata interface as much as possible.

2.2.4.1 Obtaining Values via PmtMetadataT<TYPE> Casting

Although the PmtMetadata interface should usually be used, since a metadata object's value is stored in a PmtMetadataT<TYPE> specialization, there are times when casting to the appropriate PmtMetadataT<TYPE> object will be necessary. It is encouraged that such casts should be done only when working with values (via the value() method), and the **type** of value is needed. An example of how to perform this type of casting will be seen later in section 3.5.2 Values via PmtMetadataT<TYPE>.

2.3 Keys Overview

2.3.1 ASCII Identifiers

Metadata keys are ASCII strings used to identify PmtMetadata objects. Since PmtMetadata objects are typically associated with metadata items that have been obtained from a traditional image file, this means that metadata keys typically also refer (indirectly) to image file metadata items. In other words, a metadata key simultaneously represents the PmtMetadata object and its associated metadata item in an image file.

Keys are used to communicate metadata objects to PMT's API. For example, you'll later see that the getMetadatum(…) and getMetadata(…) methods on the interface of the PmtMetadata class take keys as parameters.

2.3.2 PMT Key Naming Convention

PMT has determined its own key naming convention. This convention maps PmtMetadata objects to the metadata items found in image files, in a generalized (file format independent) yet straightforward fashion. For example, in Exif, the metadata items whose field name is ***ApertureValue*** (and whose Exif tag is 37378) maps to the PmtMetadata object whose key is ***CaptureConditions.Aperture***.

2.3.2.1 Hierarchical Groupings

PMT metadata keys are hierarchically arranged, in a logical fashion. For example, all the metadata items in an image file that deal with image capture conditions have been place under the key ***CaptureConditions***. The metadata item whose field name is ***BrightnessValue*** (and whose Exif tag is 37379) has the key ***CaptureConditions.Brightness***. ***Brightness***, ***Aperture***, and several other

keys dealing with image capture conditions are all found logically within the ***CaptureConditions*** object.

Several logical groupings have been used in determining PMT's key naming convention. The major groupings, all represented by keys are: CaptureCondtions, CaptureDevice, DigitalProcess, ImageContainer, IntellectualProperty, OuputOrder, and SceneContent. Each of the major groupings can be further divided into sub-groupings. For example, a sub-grouping ***Flash*** exists within ***CaptureConditions***, and ***Flash*** contains sub-sub-items. A key for referring to one of ***Flash***'s metadata objects looks like: ***CaptureConditions.Flash.Fired***, and another key within ***Flash*** is ***CaptureConditions.Flash.Energy***.

The logical groupings and sub-groupings result in a hierarchical view of metadata. As the above example shows, ***Fired*** is logically contained within ***Flash,*** and ***Flash*** is logically contained within ***CaptureConditions***. This results in a hierarchy of PmtMetadata objects, with multiple levels of objects logically contained within other objects. Object contained within other objects are referred to as **child objects**. Objects that contain other objects are called **parent objects**.

The top-most grouping of all the metadata is contained in a PmtMetadata object called the **root**. Specifically, the root is a PmtCompositeMetadata object. All PmtMetadata objects (except the root object itself) are contained either directly or indirectly within the root object. The root object is never explicitly referred to in a key. That is, we never use keys like ***Root.CaptureConditions.Aperture***, or ***Root.ImageContainer.Width***. Instead, the ***Root*** part is just implicitly understood, so we say ***CaptureConditions.Aperture*** and ***ImageContainer.Width***.

2.3.2.2 Key Segments

Notice that the period (**.**) character is the delimiter in a key. The period splits the entire key name into ***key segments***. ***CaptureConditions*** and ***Aperture*** and ***Brightness*** are all examples of key segments.

Since key segments represent the hierarchical structure of metadata objects, ***Aperture*** is considered a child of ***CaptureConditions***. And ***CaptureConditions*** is a parent of ***Aperture***.

Key segments, by themselves, do not uniquely identify a PmtMetadata object. They identify metadata objects only in their given context – the parent objects they belong to. For example, the ***Width*** object exists within both the ***ImageContainer*** and ***ImageContainer.Thumbnail*** objects.

2.3.2.3 Qualified Keys

An entire key name, specified as a path through the entire metadata hierarchy, is referred to as a **qualified key**. ***ImageContainer.Width*** and ***ImageContainer.Thumbnail.Width*** are both examples of qualified (or fully qualified) keys.

2.3.3 Key Mappings to Image File Metadata

You can find out how PMT's keys map (or correspond) to the metadata items found in traditional image files (Exif and TIFF) in the documents named "PmtMetadataKeysForExif.pdf" or "PmtMetadataKeysForTiff.pdf" that reside in the **doc** directory of the PMT distribution.

2.3.4 Wildcards

Keys can also contain wildcards. Wildcards are specified with the asterisk character (*****). For example ***CaptureConditions.**** or ***Cap**** contain wildcards. Wildcards provide a means of matching multiple keys with a single wildcard specification.

More on using keys, including wildcards, will later be seen in sections 3.3 and 3.4.

2.4 PmtMetadata Objects and Metadata Key Mappings

As mentioned above, a mapping exists in PMT of PmtMetadata objects to metadata keys. This section discusses those mappings in greater depth. Understanding these mappings is useful in using PMT, particularly the use of the getMetadatum(…) and getMetadata(…) methods on the PmtMetadata interface.

Understanding the mappings will also be useful for working with metadata values and using PMT's key wildcards. Reading the rest of this section, and later going over the code examples in section 3 Using PMT, will make the usefulness of understanding mappings apparent.

2.4.1 Key Segments Mappings

Each key segment in a metadata key maps to an instance of a PmtMetadata object. For example, in the key ***CaptureConditions.Flash.Fired***, there are three related PmtMetadata objects: one for ***CaptureConditions***, one for ***Flash***, and one for ***Fired***.

2.4.2 PmtCompositeMetadata Mappings

The beginning key segments in a key each map to a PmtCompositeMetadata object (which is a PmtMetadata object – remember that PmtCompositeMetadata inherits from PmtMetadata). In the above example, this means that ***CaptureConditions*** and ***Flash*** map to PmtCompositeMetadata objects.

Remember that PMT's keys provide a logical hierarchy to metadata. ***CaptureConditions*** contains many sub-keys, and ***Flash*** contains four sub-keys. ***CaptureConditions*** and ***Flash*** are examples of objects that can only contain other objects. There's no value associated with ***CaptureConditions*** or ***Flash***.

That's why all but the last key segments are mapped to PmtCompositeMetadata objects. PmtCompositeMetadata objects can only contain other PmtMetadata objects.

2.4.3 PmtMetadataT<TYPE> Mappings

Continuing with the ***CaptureConditions.Flash.Fired*** example, we see that the ***Fired*** key maps to an instance of a PmtMetadataT<TYPE> object. ***Fired*** represents a metadata item that has a value. That's why it maps to a PmtMetadataT<TYPE> object. PmtMetadataT<TYPE> objects can only contain values. They never contain other PmtMetadata objects.

All keys that actually represent metadata items with values end up mapping to an instance of the PmtMetadataT<TYPE> class. The **type** of value is the same as the <TYPE> of the templated class. In the case of ***Fired***, the type is a boolean. So the templated class's <TYPE> is a boolean.

2.4.4 More on Mappings

It's true that all beginning key segments in a key (all the segments except the last one) must map to PmtCompositeMetadata objects. But the last key may **or may not** map to a PmtMetadataT<TYPE> object. There are cases when it can map to a PmtCompositeMetadata object.

For example, if we examine the key ***CaptureConditions.Flash*** by itself (notice there's no sub-key under ***Flash***), that ***Flash*** does **NOT** map to a PmtMetadataT<TYPE> object. We already saw above that ***Flash*** is a key used to specify a logical hierarchy of metadata, and that ***Flash*** can contain other PmtMetadata objects. So, ***Flash*** maps to a PmtCompositeMetadata object.

To determine for sure whether or not the last key segment maps to a PmtMetadataT<TYPE> object or to a PmtCompositeMetadata object, you must determine the nature of the key segment. If the key segment represents a logical hierarchy in PMT's metadata keys (the key segment can contain sub-keys) then it maps to a PmtCompositeMetadata object. If the key segment represents a single metadata item with a value, then it maps to a PmtMetadataT<TYPE> object.

Refer to the default schema "DefaultDefinitions/PmtDefaultDefinitions.xsd", for a list of the default keys and their definitions used by PMT.

2.5 XML Schema

One or more XML Schema files are used by PMT during its initialization process. XML Schema is a W3C Recommendation for defining the layout of XML instance documents. Details on XML Schema and XML instance documents may be found at references [3][4][5][6].

However, most PMT users do not need to understand the details of XML Schema or XML instance documents. Most PMT users only need to understand how to properly initialize PMT. For traditional image metadata, such as the metadata commonly found in Exif and TIFF files, the fact that an XML Schema file is used by PMT is hidden from the user. XML Schema is an implementation detail of PMT. The user simply initializes PMT by making the appropriate API call.

For users working with non-traditional metadata, PMT is initialized slightly differently. In that case, the user only needs to understand how to get PMT to work with the appropriate XML Schema file. An example of working with non-traditional metadata is when a developer creates a unique file format for persisting metadata that is not typically found in Exif or TIFF files.

In fact, it's quite easy to create a new XML file format to hold any kind of data for PMT to work with. Since XML handling is already built into PMT, persisting PMT's metadata to XML instance documents with PMT is always available, even when the metadata is obtained from a binary file format such as Exif or TIFF.

The important points to remember are that XML Schema is an implementation detail to most users. Most users only need to know how to properly initialize PMT. In certain cases, PMT must properly refer to an appropriate XML Schema file when it's initialized. Developers who create their own metadata objects, or new file formats that work with non-traditional metadata, are the only ones required to have some understanding as to the details of XML Schema. The topic of creating XML Schema files for use with PMT is covered later in section 4.4.1 Defining Additional Metadata – Creating New Schema.

2.6 Accessors

All file I/O with PMT (with the exception of XML Instance Serialization via the IO stream operator overload on the PmtMetadata class interface) is performed via use of a PMT Accessor. An abstract base class, PmtAccessor, provides the interface for opening, closing, reading, and writing metadata objects to or from a file.

The methods on the PmtAccessor interface use PmtMetadata objects for performing the requested file I/O operations. Using PmtAccessor objects in conjunction with PmtMetadata objects provides a layer of abstraction between the file formats where metadata resides, from the metadata objects themselves. PmtMetadata objects behave the same, regardless as to if the metadata values were obtained from an Exif or TIFF (or other) file.

PMT currently provides Accessors for some file formats. Each inherits from the PmtAccessor class. Support for additional file formats should also be done through inheritance from the PmtAccessor class.

More on using Accessors will be seen later in section 3.6 Accessors.

2.7 Smart Pointers

2.7.1 Definition

Smart pointers are used throughout PMT. A smart pointer behaves in many respects like a regular C++ pointer, but is actually a separate C++ object that refers ("points") to another object. In PMT's implementation, the smart pointer object actually contains a member variable that points to the other object. The smart pointer then behaves like a regular pointer by implementing operator overloads for pointer-like operations such as the **.** and **->** operators.

In addition to just acting like a pointer, a smart pointer is easier to work with because it automatically handles memory de-allocation of the object to which it points. In other words, if you use a smart pointer instead of a regular C++ pointer, you do not need to (and should not) call 'delete' on the object to which you're pointing. When the smart pointer goes out of scope, the object it points to is automatically de-allocated.

2.7.2 PmtMetadataPtr and PmtAccessorPtr

In particular, important smart pointers used in PMT's API are PmtMetadataPtr and PmtAccessorPtr. PMT passes PmtMetadata objects to an application by passing PmtMetadataPtr smart pointers. PmtAccessor objects are passed via PmtAccessorPtr smart pointers. This means that an application using PMT never needs to deal with de-allocating the memory associated with PmtMetadata or PmtAccessor objects.

It is common when discussing a particular PmtMetadataPtr to assume that the underlying PmtMetadata object is what's being referred to (not the smart pointer object itself). A smart pointer behaves much like real C++ pointer, so think of a smart pointer in the same way you would typically refer to a real C++ pointer. Frequently, when you discuss a C++ pointer, you are referring to the object the pointer points to, not the pointer itself. The same is true for PmtMetadataPtr smart pointers. The PmtMetadata object is what's typically being referred to when a PmtMetadataPtr smart pointer is mentioned.

3. Using PMT

This section describes typical uses of PMT, and provides code examples to illustrate those uses.

For brevity throughout the following sections, when the term "getMetadata(…)" is used, it sometimes refers generally to the getMetadatum(…) or getMetadata(…) methods on the PmtMetadata class interface.

Note: The code examples throughout this document are deliberately simple. They usually do not contain the required #include files. Also, error checking is not performed, as would typically be done in a real application. Writing the code this way results in succinct and simple examples, for illustrative purposes. For any application that uses PMT, it is suggested that the appropriate error checking, such as testing return values, and implementing C++ **try/catch** statements, be included in your code.

For additional examples on how to use PMT, please consult any of the programs provided in the examples directory, or the **PmtInterpreterTest** and **AccessorTest** test programs. The test programs are covered in section 6.1.5 Testing PMT Installation.

3.1 Initializing PMT

PMT must be initialized for successful use. It is easy to initialize PMT to work with most metadata that's stored in a traditional image file, such as Exif or TIFF. It is typically performed as follows:

```
PmtLogicalDefinitionInterpreter interpreter;
interpreter.load();
```

The PmtLogicalDefinitionInterpreter class is always used to initialize PMT. The load() method prepares PMT to work with most traditional image metadata.

Other methods on the PmtLogicalDefinitionInterpreter interface allow a user to initialize PMT for use with metadata other than typical image metadata. For example, if a user creates a custom XML instance document that contains non-typical image metadata, then PMT needs to be initialized somewhat differently. Details of other ways of initializing PMT are covered in section 4.2 Initializing PMT.

Since multiple applications or threads may use PMT concurrently, PMT determines if it has already been initialized via the load() method. If load() has already been called, PMT will skip over the initialization process if it's called again. In other words, it does not hurt to call load() from multiple applications. Since an application must ensure PMT has been initialized, it should always call load() or one of the other methods discussed in section 4.2 first.

3.2 Creating Root Object

Working with PMT includes creating the appropriate PmtMetadata objects to manipulate. The first PmtMetadata object that must be created is done so via the static create() method on the PmtMetadata interface. The returned PmtMetadata object is referred to as the **root** object. (Refer to section 2.3.2.1 Hierarchical Groupings for a definition of the root object.)

Creating the root is simple. For example,

```
PmtMetadataPtr root = PmtMetadata::create();
```

The above code creates the PmtMetadata root. (The PmtMetadata object is actually stored in the returned PmtMetadataPtr smart pointer. However, remember we frequently refer to a PmtMetadataPtr and its associated PmtMetadata object to each other as if they're the same thing: the PmtMetadata object.) The PmtMetadata object represents the logical grouping of all potential metadata. The root object is an instance of a PmtCompositeMetadata object. This PmtCompositeMetadata object is initially empty. It contains no other PmtMetadata objects. However, it has the potential of containing all the metadata you will want to work with.

Calling PmtMetdata::create() is kind of like a bootstrapping process. In order to do anything of use with PMT, you need to have PmtMetadata objects to work with. You can only obtain PmtMetadata objects (with one exception – the readMetadata(void) method in the PmtAccessor class) via the PmtMetadata interface. This is why the PmtMetadata::create() method is static. It allows an application to obtain a PmtMetadata object, without having a PmtMetadata object to start with. It's how you bootstrap (start) working with metadata.

Recall that PMT metadata is logically grouped by a hierarchy, with the top most metadata groups being CaptureConditions, CaptureDevice, DigitalProcess, etc. Think of the root object as being

the object that contains all these topmost groups. The root will contain CaptureConditions, CaptureDevice, DigitalProcess, etc.

3.3 getMetadatum(…) Use

There is an important concept to understand in how PMT operates. Before you can work with a particular metadata object, the object must first be created. The getMetadata(…) methods on the PmtMetadata class performs creation of PmtMetadata objects for you. getMetadata(…) also allows you to obtain a smart pointer (PmtMetadataPtr object) to a PmtMetadata object, so you can further manipulate it.

Manipulating an object can include 1) creating sub-objects within itself (if its an object of type PmtCompositeMetadata), or 2) inspecting or changing its value (if its an object of type PmtMetadataT<TYPE>).

Let's look at an example to further understand how to use getMetadata(…). Assume a root metadata object is created as below:

```
PmtLogicalDefinitionInterpreter interp;
interp.load();  // initialize PMT

PmtMetadataPtr root = PmtMetadata::create();  // create root
```

The above code will return a root metadata object. However, no metadata objects other than the root have yet been instantiated. This means that all the other metadata objects, such as CaptureConditions, CaptureConditions.Aperture, CaptureDevice, ImageContainer, etc., have not yet been created.

Assuming we were to execute the following code in addition to the above code snippet, two new PmtMetadata objects would be created:

```
PmtMetadataPtr md;

md = root->getMetadatum("CaptureConditions.Aperture");
```

At the point in time the above code is executed, the following occurs: 1.) since no CaptureConditions object yet exists, one is created, 2.) since no Aperture object yet exists, one is created, and 3.) since the CaptureConditions.Aperture key was requested, a smart pointer to the Aperture PmtMetadata object (a PmtMetadataPtr object) is returned to the caller.

It is important to note that if this line of code were executed after the above code snippet (repeating the getMetadatum call):

```
md = root->getMetadatum("CaptureConditions.Aperture");
```

Since both CaptureConditions and Aperture already exist at the point in time this line of code is executed, that no new PmtMetadata objects would be created. However, the PmtMetadataPtr to the Aperture would still be returned.

getMetadatum(…) can take an optional second argument, called **createIfNotExists.** This argument is a boolean flag, which defaults to **true**. It tells PMT to create the requested metadata objects if they do not yet exist when set to true. When false is passed in, no object is created, and the appropriate PmtMetadataPtr is returned only if the object already exists. For example:

```
PmtMetadataPtr root = PmtMetadata::create();
PmtMetadataPtr md =
   root->getMetadatum("CaptureConditions.Aperture", false);
```

The above code results in no CaptureConditions or Aperture objects being created, since false is passed into getMetadatum(…). A NULL PmtMetadataPtr is returned because no Aperture object yet exists (and none was created).

If instead the following code were executed:

```
PmtMetadataPtr root = PmtMetadata::create();
PmtMetadataPtr md;
md = root->getMetadatum("CaptureConditions.Aperture");
md = root->getMetadatum("CaptureConditions.Aperture",false);
```

The Aperture object is returned in both getMetadatum(…) calls above. The first call to it creates both the CaptureConditions and Aperture objects. The second call to getMetadatum(…) returns a valid PmtMetadataPtr to the Aperture object, since it already exists.

If an illegitimate key is passed into getMetadatum(…), a NULL PmtMetadatPtr is returned. Consider the following example.

```
PmtMetadataPtr root = PmtMetadata::create();
PmtMetadataPtr md;
md = root->getMetadatum("CaptureConditions.Foo");
```

The above code results in the CaptureConditions object being created, since it's a legitimate object. However, Foo is not a legitimate sub-object under CaptureConditions, so no Foo object will be created. This example is interesting. Even though a CaptureConditions object is created, a NULL PmtMetadataPtr will still be returned, since the return value is based on the Foo object. Since no Foo exists, or can be created, NULL is returned.

3.3.1 Relative Calling

So far, the examples in this User's Guide have all used keys that are relative to the root PmtMetadata object. A key like CaptureConditions.Aperture is relative to the root object, because CaptureConditions is a direct child object of the root.

getMetadatum(…) calls can be made through any PmtMetadata object. Consider an object like Aperture, where its direct parent object is not the root object, but is CaptureConditions. For example:

```
PmtMetadataPtr root = PmtMetadata::create();
PmtMetadataPtr captureConditions;
PmtMetadataPtr aperture;

captureConditions = root->getMetadatum("CaptureConditions");
aperture = captureConditions->getMetadatum("Aperture");
```

The above code illustrates the concept that metadata objects can be obtained in a relative fashion. Particularly, Aperture is relative to CaptureConditions (or in other words, Aperture is a child of CaptureConditions). This means that the Aperture object can be created by calling through the CaptureConditions object, as is done in the last line of code in the above example. Being able to obtain metadata objects in a relative fashion is possible at any level in the metadata hierarchy. Wildcards can be used at any level in the metadata hierarchy too.

Obtaining parent objects, and using them as CaptureConditions is used above, is mainly a matter of preference or convenience. If a lot of metadata objects will be used at a particular level in the metadata hierarchy, then obtaining that sub-object first can be useful.

Keep in mind that keys passed to PMT should be specified relative to the PmtMetadata object being called through. This means that passing in a key that's relative to the root object, will only be legitimate when the root object is called through. For example, if the key "*CaptureConditions*.Aperture" were passed into the **getMetadatum(…)** call in the last line of code in the above example, then the key would be illegitimate (not found) and a NULL value would be returned from **getMetadatum(…)**.

3.4 getMetadata(…) Use

Multiple metadata objects can be created and/or retrieved through the overloaded getMetadata(…) method. When more than one PmtMetadata object is returned from getMetadata(…), only those objects of PmtMetadataT<TYPE> are returned. This means that only PmtMetadataT<TYPE> objects are returned, and no PmtCompositeMetadata objects are returned. However, if only one object is requested, and it is of type PmtCompositeMetadata, that composite object will be returned.

getMetadata(…) differs from getMetadatum(…) in that it returns a PmtMetadataIterator object, rather than a PmtMetadataPtr smart pointer. getMetadata(…) also differs from getMetadatum(…) in that it can take keys with **wildcards**. An example that uses a wildcard will first be considered below.

3.4.1 Wildcards

Wildcards provide a simple means of matching multiple keys with a single wildcard specification.

For particular situations, wildcards are very powerful and useful. Good examples of using wildcards are to create many PmtMetadata objects at once, or to quickly access multiple existing PmtMetadata objects in a convenient fashion.

It is important to note that wildcards may or may not be useful, depending on what it is you're trying to accomplish. In fact, in the case where all the metadata is to be read in from a file, it is suggested to not use wildcards. For more about that situation, consult section 3.6.2.1 Reading Existing Metadata.

3.4.1.1 Lone Key Segment Wildcards

Assume the following code snippet is executed:

```
PmtLogicalDefinitionInterpreter interp;
interp.load();  // initialize PMT
PmtMetadataPtr root = PmtMetadata::create();  // create root

PmtMetadataIterator mdIter =
   root->getMetadata("CaptureConditions.*");
```

Temporarily ignoring the PmtMetadataIterator, and focusing on the key ***CaptureConditions.****, notice the asterisk character (*****). When an asterisk is found as the only character of a key segment, this is referred to as a **lone key segment wildcard**. This instructs PMT to create all the sub-objects logically contained within ***CaptureConditions***. For instance, this will create ***Aperture***, ***BatteryLevel***, ***Brightness***, etc. – all the objects that begin with the ***CaptureConditions*** key segment. Notice that sub-objects are created recursively to as many levels as deep as exist. For example, the sub-sub objects contained within ***CaptureConditions.Flash*** are created too: ***CaptureCondtions.Flash.Fired***, ***CaptureConditions.Flash.Return***, etc.

Wildcards are very powerful, and can be used at various hierarchical levels. For example, executing the following code:

```
PmtMetadataIterator mdIter =
```

```
root->getMetadata("CaptureConditions.Flash.*");
```

…will only create the sub-objects contained within ***Flash***. No additional objects will be created within **CaptureConditions** other than those within ***Flash***.

3.4.1.2 Matching Wildcards

An asterisk following other characters in a key segment characterizes matching wildcards. Using a matching wildcard as follows produces different results than using a lone key segment wildcard. For example,

```
PmtMetadataIterator mdIter = root->getMetadata("Capture*");
```

This wildcard match causes PMT to create all keys that begin with the characters ***Capture***, and then create all the sub-objects under the matched keys (like single key segment wildcards do). The ***Capture**** matches and creates ***CaptureConditions*** and ***CaptureDevice***. Then all the sub-objects under those two are created.

3.4.1.3 Accessing Multiple Existing Objects

A good use of wildcards is to conveniently access multiple PmtMetadata objects. Consider using the getMetadata(…) method, and passing a 'false' for its second argument. This instructs getMetadata(…) to only return existing objects. No PmtMetadata objects will be created. For example,

```
PmtMetadataIterator mdIter = root->getMetadata(
      "CaptureConditions.*", false);
```

The above call to getMetadata(…) will return all existing PmtMetadata objects within the ***CaptureConditions*** object. This, in fact, will have the same results as the example calls to getMetadata(…) in the following section.

3.4.1.4 Empty Key

An empty key ("") passed to getMetadata(…) is a form of a wildcard. It signals PMT to retrieve all existing metadata objects. However, the createMetadataIfNotExists flag is ignored, and it's as if it were set to false. For example, the results from the following two ways of obtaining keys are identical.

```
PmtMetadataPtr captureConditions;
captureConditions = root->getMetadatum("CaptureConditions");

PmtMetadataIterator mdIter1 =
      captureConditions->getMetadata("");

PmtMetadataIterator mdIter2 =
      captureConditions->getMetadata("*",false);
```

The empty key is a convenient way of specifying a lone segment wildcard with the **createIfNotExists** argument set to false.

3.4.2 PmtMetadataIterator

A PmtMetadataIterator object is returned from a call to getMetadata(…). PmtMetadataIterator's allow traversal over multiple PmtMetadata objects. They contain smart pointers to the PmtMetadata objects (PmtMetadataPtr's). The methods mainly used on the PmtMetadataIterator interface face are **start()** and **next()**. They each return a PmtMetadataPtr.

For example:

```
PmtMetadataIterator mdIter = root->getMetadata("Capture*");
PmtMetadataPtr md;

md = mdIter.start();
while (md)
{
   md->show();

   md = mdIter.next();
}
```

The above code obtains a PmtMetadataIterator object when ***root->getMetadata("Capture*");*** is called. The returned objects are traversed with the ***mdIter.start()*** and ***mdIter.next()*** calls. They return a PmtMetadataPtr if there are any more PmtMetadataPtr objects left in the PmtMetadataIterator. An iterator may contain zero, one, or more PmtMetadataPtr's. Since it could potentially contain zero, it's a good habit to test for a NULL PmtMetadataPtr as shown in the above code example. Although we are confident that multiple objects will be returned in this case, testing for a NULL with the ***while (md)*** test on the loop ensures that no calls will be attempted on NULL PmtMetadataPtr objects under any circumstances.

In this code example, when there are no more objects to be traversed, the ***next()*** will return a NULL PmtMetadataPtr object, and the ***while (md)*** loop will exit. Keep in mind however, that ***start()*** will also return a NULL PmtMetadataPtr object, if the PmtMetadataIterator is empty.

3.4.3 getMetadata(…) Parameters

getMetadata(…) has two overloads. The first takes a simple metadata key as a parameter. The second takes an STL list<> of keys. Passing an STL list of keys is available for convenience. Instead of having to call getMetadata(…) several times to obtain different keys, passing a list is simpler. Consider the following example.

```
PmtKeyList myList;
myList.push_back("Aperture");
myList.push_back("Flash.Return");
myList.push_back("MeteringMode");
myList.push_back("Sub*");
mdIter = captureConditions->getMetadata(myList);
```

Assuming the captureConditions object were already appropriately setup before being used, the above code would obtain the metadata objects Aperture, Flash.Return, MeteringMode, and all those that match Sub* (and all Sub*'s children, if there were any). In this case, Sub* matches the keys SubjectArea, SubjectDistance, and SubjectDistanceRange. One call to getMetadata(…) obtained all those objects.

getMetadata(…) also takes a third argument, called **entryToSearch**. See section 4.1Aliases for more on using that argument.

3.5 Working With Values

Metadata values are always stored in instances of the PmtMetadataT<TYPE> class. The PmtMetadataT<TYPE> class is a templated class, with the type of metadata being stored in the template's <TYPE> parameter.

There are two basic ways of working with values with PMT. The first way is done so via the base PmtMetadata class interface.

As mentioned before, it is generally advisable to work with PmtMetadata objects through the base PmtMetadata class interface whenever possible. This is advantageous since significant portions of

an application do not care about the specific type of metadata (e.g., `PmtMetadataT<short>`, `PmtMetadataT<float>`, etc.) and do not need to cast a PmtMetadataPtr to a particular PmtMetadataT<TYPE> object.

There are times, however, when casting to a PmtMetadataT<TYPE>specialization is useful. The second way demonstrated below illustrates such casting. In context of working with values this second way, it is common to exercise such casts.

3.5.1 Values via PmtMetadata

Although the type of data is unavailable from the base class PmtMetadata interface (versus the PmtMetadataT<TYPE> interface), the PmtMetadata interface has two methods for working with values. These methods are getValueStr() and setValueStr(…). They return and take STL string arguments (or wstring arguments if using a Unicode build of PMT). For example in a regular, non-Unicode, build of PMT, the following code illustrates getting a metadata value.

```
PmtMetadataPtr md =
   root->getMetadatum("CaptureConditions.Aperture");
string value = md->getValueStr();
cout << "CaptureConditions.Aperture metadata value is: "
   << value.c_str();
```

The following example shows how to set a metadata value.

```
string value = "1";
PmtMetadataPtr md =
   root->getMetadatum("CaptureConditions.Aperture");
md->setValueStr(value);
```

(For simplicity, the above examples use STL string objects for the values. In actual production code, it is suggested that PMT's EkString typedef and EK_L macro be used as shown in section 4.5.1 Unicode Build Helpers. These mechanisms make interacting with regular or Unicode builds of PMT easier.)

Using getValueStr() and setValueStr(…) are useful, not only for avoiding casting to the appropriate PmtMetadataT<TYPE> specialization, but also for getting or setting values for display through a user interface. User interfaces typically deal with string arguments.

3.5.2 Values via PmtMetadataT<TYPE>

When needing to work with a metadata value in its exact appropriate C++ type, casting to a PmtMetadataT<TYPE> object can be performed as shown in this section. To set a value, use the value() method as follows.

```
PmtMetadataPtr md =
   root->getMetadatum("CaptureConditions.Aperture");
dynamic_cast<PmtMetadataT<float>*>(md.ptr())->value() = 4;
```

This is accomplished through the `dynamic_cast` construct, casting the ***`PmtMetadata`*** base class pointer into one of its derived classes. Note, the `dynamic_cast` must be performed on the actual base class pointer, `PmtMetadata*`, not the smart pointer ***`PmtMetadataPtr`***. Access to the base class pointer is provided through the `ptr()` method on the smart pointer class. The value() method returns a reference to the member variable, of type <TYPE>, that holds the metadata value.

Getting a value may be performed as follows.

```
float value;
PmtMetadataPtr md =
   root->getMetadatum("CaptureConditions.Aperture");
```

```
value =
   dynamic_cast<PmtMetadataT<float>*>(md.ptr())->value();
```

Keep in mind that the above examples that get metadata values assume that the values have already been placed into the appropriate PmtMetadataT<TYPE> object. In particular, the above examples are not illustrating how to obtain metadata values from files. For more on obtaining metadata values from files, see section 3.6 Accessors.

The above method of casting determines the type at run time through the `dynamic_cast` construct, which consumes execution time and requires careful planning of exception handling. As an alternative, PMT also provides support for determining metadata types at compile time. This is accomplished by using a Visitor Design Pattern. The application must supply the appropriate classes for the Visitor Design Pattern. Details are provided in Appendix B.

3.6 Accessors

As mentioned in section 2.6 Accessors, PmtAccessors control all file I/O within PMT (the exception being XML Instance Serialization via the IO stream operator overload on the PmtMetadata class interface). This section shows how to use Accessors. First, instantiating an Accessor will covered. Then, reading and writing will be discussed.

3.6.1 Instantiating an Accessor

There are three basic different ways to instantiate a PMT Accessor: using PmtAccessor::getAccessor(…), using PmtAccessor::create(…), and explicitly instantiating one in C++. We'll first look at some examples of these different ways.

3.6.1.1 PmtAccessor::getAccessor(…)

Obtaining an Accessor can be performed through the static method, getAccessor(…), on the PmtAccessor interface as follows.

```
#include "PmtAccessor.h"
#include "PmtExifAccessor.h"

const char * const myFile = "KodakDC260.jpg";
PmtAccessorPtr acc;
acc = PmtAccessor::getAccessor(myFile);
```

The call to getAccessor(…) causes PMT to instantiate a PmtAccessor of the appropriate type for the given file passed in. In this case, a PmtExifAccessor object is created (PmtExifAccessor is a sub-class of PmtAccessor) and returned.

Even though particular sub-classes of the PmtAccessor base class are returned from getAccessor(…), the base class interface is used. This is apparent from the fact that the Accessor is used by the PmtAccessorPtr smart pointer. A PmtAccessorPtr is a smart pointer to a PmtAccessor (the base class) object.

getAccessor(…) can also take an explicit second parameter as follows.

```
acc = PmtAccessor::getAccessor(myFile, PMT_READWRITE);
```

The above code passes in PMT_READWRITE, so the file can be written to, as well as read from. The second parameter to getAccessor(…) is actually a default parameter that defaults to PMT_READONLY. This means a file can only be read from unless a different parameter for the second argument is explicitly passed in. For creating new files, the PMT_CREATE parameter may be passed in also.

getAccessor(…) also automatically opens the given file. The fact that getAccessor(…) performs both functions of instantiating an appropriate Accessor, and automatically opens a file, makes it an easy way of working with Accessors in PMT.

The easiest way to guarantee a given PmtAccessor object is available is to include the file "PmtAllAccessors.h" in the application code.

3.6.1.2 PmtAccessor::create(…)

The following example shows how to use PmtAccessor::create(…) to instantiate an Accessor.

```
PmtAccessorPtr acc;
acc = PmtAccessor::create(PMT_FORMAT_EXIF);
```

The only parameter passed into PmtAccessor::create(…) is an argument of type PmtImageFileFormatName, which is an enum with the following enumerators: PMT_FORMAT_EXIF, PMT_FORMAT_TIFF, and PMT_FORMAT_XML. PmtAcceesor::create(…) returns a PmtAccessorPtr of the appropriate file type.

Note that PmtAccessor::create(…) does not open the file. It only serves to instantiate an object of type PmtAccessor. In the above example, this means that a PmtExifAccessor object is returned via the PmtAccessorPtr. PmtAccessorPtr smart pointers point to PmtAccessor objects, and the returned PmtExifAccessor is a PmtAccessor object, since the PmtExifAccessor class inherits from the PmtAccessor class.

3.6.1.3 Explicit C++ Instantiation

Accessors can be explicitly instantiated, in typical C++ fashion, with the new operator, as follows.

```
PmtAccessorPtr acc = new PmtExifAccessor;
```

The above example instantiates an Accessor for use with Exif file. For TIFF or XML, use the appropriate PmtTiffAccessor or PmtXmlAccessor classes as appropriate.

Instantiating Accessors explicitly or with PmtAccessor::create(…) can only be used when the type of file (Exif, TIFF, or XML) is known ahead of time. For processing various file types in a loop, PmtAccessor::getAccessor(…) can be used instead.

3.6.2 Reading Metadata

Requests to read metadata are fulfilled by the overloaded ***`readMetadata()`*** method. One version accepts a metadata pointer (*PmtMetadataPtr*). If the metadata is a PmtCompositeMetadata object, all metadata instances present in it will be read in from the file. A second version accepts a *PmtMetadataIterator* instance, in which all metadata objects in the iterator will be read from the file. A final version takes no arguments and returns all the existing metadata instances in the file.

It is important to note that the suggested means of reading the existing metadata in a file is to use the ***`readMetadata()`*** that takes no arguments. This is discussed further below in the next section, Reading Existing Metadata.

Using a PMT Accessor to read from a file can be performed as follows.

```
const char * const myFile = "KodakDC260.jpg";

PmtMetadataPtr md;
md = root->getMetadatum("CaptureConditions.Aperture");

PmtAccessorPtr acc;
acc = PmtAccessor::getAccessor(myFile);
```

```
acc->readMetadata(md);  // reads in value
```

Note that the ***md = root->getMetadatum("CaptureConditions.Aperture");*** code does not perform any file I/O. It's creating a couple of PmtMetadata objects (one for CaptureConditions, another for Aperture), and returning a smart pointer to the Aperture object.

The call to ***acc->readMetadata(md);*** is what actually performs the file I/O. In this case, since a PmtMetadataPtr to the CaptureConditions.Aperture object is passed into readMetadata(…), the single Aperture value is read in from the "KodakDC260.jpg" file.

Passing a PmtMetadataIterator can be performed as follows.

```
const char * const myFile =KodakDC260.jpg";

PmtMetadataIterator mdIter;
mdIter = root->getMetadata("CaptureConditions.*");

PmtAccessorPtr acc = PmtAccessor::getAccessor(myFile);
acc->readMetadata(mdIter);
```

The above example simply passes in the mdIter PmtMetadataIterator instead of a simple PmtMetadataPtr. All existing objects within ***CaptureConditions*** will be read in.

3.6.2.1 Reading Existing Metadata

It is important to point out that only those metadata items that exist within an image file will actually provide real values to corresponding PmtMetadata objects. Consider an image file that is quite sparse in metadata, and suppose it contains only two metadata values, both within CaptureConditions. Only Aperture and BatteryLevel exist in the file. In the above code example, where all the CaptureConditions objects are passed to readMetadata(...), only the two PmtMetadata objects Aperture and BatteryLevel would have their values obtained from the file. This means that all the other objects within CaptureConditions would have invalid (default) values. They would not have values from an image file in them.

Understanding this concept is particularly important when using wildcards. Being aware of which metadata objects have real values in them is important. PMT provides two mechanisms for dealing with this: the readMetdata() overload that takes no arguments, and the throwErrors() method on the PmtAccessor interface. Using throwErrors() is discussed below in section 3.6.4 Errors

The suggested means of reading only the metadata items that exist in a given file is to use the readMetadata() overload that takes no arguments. Here is a code example that illustrates the use of this method to get only the existing metadata in a file.

```
const char * const myFile = "KodakDC260.jpg";

PmtAccessorPtr acc;
PmtMetadataPtr mdRoot;
acc = PmtAccessor::getAccessor(myFile);
mdRoot = acc->readMetadata();  // reads in all metadata
```

Note that the above call to readMetadata() returns a root metadata pointer. This illustrates the fact that readMetadata() with no arguments takes care of creating the appropriate metadata objects for you. All of the metadata values encountered in the file will have corresponding PmtMetadata objects instantiated for them. Then the root PmtMetadata object of the file is returned.

This is different than using getMetadata(…) to create metadata objects for you. readMetadata() with no arguments is an exception to the general rule that getMetadata(…) is used to created metadata objects. Although readMetadata()'s use is exceptional in the sense that it automatically creates PmtMetadata objects for you, this in no way implies that using it is rare. In fact, most

applications will use readMetadata() with no arguments to obtain the metadata from files, when getting all of the existing metadata in a file is what's desired.

3.6.3 Writing Metadata

The requests to write metadata are fulfilled by the overloaded ***`writeMetadata()`*** method. This method takes a ***`PmtMetadataPtr`*** and an optional boolean parameter as arguments. If the boolean parameter is false, then only the given metadata (including any contained PmtMetadata objects, if it's a PmtCompositeMetadata) will be written; otherwise, the whole metadata tree starting from the outer-most parent will be written. The default value for this boolean parameter is false. In the case of the root metadata instance, the value of the boolean flag is irrelevant, in either case all metadata instances present will be written. Another version accepts a *PmtMetadataIterator* instance.

Using an Accessor to write to a file can be performed as follows.

```
const char * const myFile = "KodakDC260.jpg";

PmtMetadataPtr md;
md = root->getMetadatum("CaptureConditions.Aperture");
md->setValueStr("4");

PmtAccessorPtr acc;
acc = PmtAccessor::getAccessor(myFile, PMT_READWRITE);
acc->writeMetadata(md);
```

Notice the PMT_READWRITE parameter passed into getAccessor(…). As mentioned above, this opens the file for reading or writing.

3.6.4 Errors

The readMetadata(…)/writeMetadata(…) methods of the PmtAccessor interface may involve a sequence of read/write operations on a set of metadata. There are two general ways these methods approach read/write operations: 1.) If one of the read/writes fails, the entire sequence of the requested read/writes will continue, or 2.) Abort the entire sequence of read/write requests, when any request fails.

PMT by default follows approach #1 above. Approach #2 can be followed by calling the throwErrors(…) on the PmtAccessor interface as follows.

```
acc->throwErrors(true);
```

This will cause PMT to abort a read/write sequence. However, this approach has the advantage of notifying a user when a requested metadata item does not exist in an image file during a read request. If you want to use approach #1 above in error handling, but want to only read in the existing metadata from a file, it is suggested that you use the overload version of readMetadata() that takes no arguments. See section 3.6.2.1Reading Existing Metadata for more on using the readMetadata() overloaded method that takes no arguments.

3.6.5 Copying All Metadata from Source to Destination

As noted in the previous section, ***`readMetadata(void)`*** (the overload with no arguments) will read all the existing metadata instances from the file. The return value is a ***`PmtMetadataPtr`*** to a root metadata instance that contains ***`PmtMetadata`*** instances of all the metadata in the file. One valuable use of this functionality is to perform a copy of all metadata from source to destination. For example,

```
#include "PmtAllAccessors.h"

int main()
{
   PmtLogicalDefinitionInterpreter interpreter;
   interpreter.load();

   // create a accessor for the given source image file
   PmtAccessorPtr srcAcc =
      PmtAccessor::getAccessor("/path/to/srcfile");

   // create an accessor for the given dest image file
   PmtAccessorPtr destAcc =
      PmtAccessor::getAccessor("/path/to/destfile",
      PMT_READWRITE);

   if(srcAcc && destAcc)
   {
      // read all existing metadata from source
      PmtMetadataPtr srcMd = srcAcc->readMetadata( ) ;

      // write out the metadata to the destination
      destAcc->writeMetadata( srcMd ) ;
      srcAcc->close() ;
      destAcc->close() ;
   }
   return 0;
}
```

3.7 XML Instance Serialization

As an alternative to using *PmtXmlAccessor* to produce XML serializations of *PmtMetadata* instances, they can also be directly serialized to an output stream and restored from an input stream. This functionality is accessed through the overloaded "<<" and ">>" operators on the *PmtMetadata* class. For example, to serialize a *PmtMetadata* instance to a string:

```
PmtMetadataPtr rootMd = PmtMetadata::create( );
// do some thing with the metadata instance
// now serialize it to a string
ostrstream osstream ;
osstream << rootMd ;
string strMd = osstream.str() ;
```

to restore the instance, just do the opposite:

```
PmtMetadataPtr restoredMd = PmtMetadata::create( );
istrstream isstream(strMd.c_str(), strMd.size()) ;
isstream >> restoredMd ;
```

3.8 Exception Handling

PMT uses C++ exception handling mechanism to report and recover from errors. All PMT errors are reported by throwing an exception object of type *PmtError*, which is derived from the class of *EkError*. The *PmtError* class contains the information about the cause and location of the error, and would be used in a standard try/catch block like this:

```
try
{
   PmtLogicalDefinitionInterpreter interpreter;
   interpreter.load();

   PmtAccessorPtr acc =
```

```
        PmtAccessor::getAccessor("imagefile");

    if (acc)
    {
        PmtMetadataPtr md =
            PmtMetadata::create("ImageContainer.Orientation");
        acc->readMetadata(md);

        ...
    }
}
catch (PmtError& e)
{
    cout << e.getMsg();
    // code to handle the error such as cleanup, or re-throw
}

catch (...)
{
    // caught some other unknown errors
}
```

4. Additional Features

4.1 Aliases

An alias is a convenience mechanism allowing one or more metadata keys to be represented by a single, user-defined alias. Typically, an alias will be used to quickly identify multiple metadata keys. The alias can be used in place of a regular metadata key. The metadata keys defined within an alias definition are called **alias members**. In other words, alias members are the regular metadata keys associated with an alias itself.

4.1.1 Alias Definition File

An alias is defined in an XML instance file. The valid tags are:

- MetadataAliases – root level element, must contain one or more alias definitions
- MetadataAlias – element defining an alias
  - AliasKey – assigned the name of the alias – used as a regular key name in PMT
- AliasMember – element that identifies a metadata key as an alias member
  - MetadataKey – assigned the name of the alias member – must be a regular key (aliases are not allowed as alias members)

Here is an example alias definition file:

```
<?xml version="1.0" encoding="UTF-8"?>
<MetadataAliases>
    <MetadataAlias AliasKey="FirstAliasKey">
        <AliasMember MetadataKey="CaptureConditions.Aperture"/>
        <AliasMember MetadataKey="CaptureConditions.Flash.Fired"/>
        <AliasMember MetadataKey="CaptureDevice.Model"/>
        <AliasMember MetadataKey="CaptureDevice.Make"/>
    </MetadataAlias>
    <MetadataAlias AliasKey="SecondAliasKey">
        <AliasMember MetadataKey="SceneContent.Audio"/>
        <AliasMember MetadataKey="SceneContent.UserComment.UserComment"/>
```

```
            <AliasMember MetadataKey= "SceneContent.ImageCaptureDataTime.
                                        ImageCaptureDateTime"/>
        </MetadataAlias>
    </MetadataAliases>
```

The above alias definition file would result in the ability to pass the **FirstAliasKey** alias to **getMetadata(…)** to have the **CaptureConditions.Aperture, CaptureConditions.Flash.Fired, CaptureDevice.Model, and CaptureDevice.Make** objects returned. The three keys **SceneContent.Audio**, **SceneContent.UserComment.UserComment**, and **SceneContent.ImageCaptureDataTime.ImageCaptureDateTime** would be returned if the **SecondAliasKey** alias were used.

4.1.2 Loading Aliases

The aliases file must be loaded, before it can be used by PMT. For details on loading aliases, refer to section 4.2 Initializing PMT.

4.1.3 Using Aliases

Aliases may be used on the two overloaded versions of **getMetadata(…)** and the two overloaded versions of **deleteMetadata(…)** on the PmtMetadata class interface. Particularly, the **entryToSearch** argument determines how aliases may be used. **entryToSearch** is an argument of type **PmtEntryTypeEnum**. **PmtEntryTypeEnum** is a C++ enum typedef that contains three possible values: PMT_METADATA_KEYS_ONLY, PMT_ALIAS_KEYS_ONLY, and PMT_ALL_KEYS.

PMT_ALL_KEYS is the default value for **entryToSearch**. PMT_ALL_KEYS will cause PMT to compare the given key(s) passed into **getMetadata(…)** or **deleteMetadata(…)** with all the metadata keys PMT knows about, or any loaded aliases. (The metadata keys that PMT knows about is determined by how PMT was initialized with the various **load(…)**, **loadWithAliases(…)**, etc. methods on the PmtLogicalDefinitionInterpreter interface. All the traditional metadata keys associated with metadata commonly found in an Exif or TIFF file, will typically be known to PMT, since the default Schema should typically be loaded. The aliases known to PMT consist of all the aliases previous loaded into PMT during initialization time.) In other words, by default, PMT will treat all passed in keys as potential regular metadata keys, or aliases. If the passed in key is a regular key, or if it matches a loaded alias, then the request will be fulfilled.

PMT_METADATA_KEYS_ONLY can be used if the keys passed in to **getMetadata(…)** or **deleteMetadata(…)** are known be contain of regular metadata keys only, and no aliases are being used. Conversely, PMT_ALIAS_KEYS_ONLY can be used if the passed in keys are known to be aliases only. Both PMT_METADATA_KEYS_ONLY and PMT_ALIAS_KEYS_ONLY will improve the efficiency of the method called.

It is important to note that aliases must be used in an appropriate fashion, when considering which metadata object the call to **getMetadata(…)** or **deleteMetadata(…)** is being made through. Recall from section 3.3.1 Relative Calling that a key name must be relative to the PmtMetadata object called through, in order for the key to be properly found. For practical purpose, aliases should usually only contain alias members that are relative to the root PmtMetadata object. Then, C++ code that uses the aliases should always call through (relative to) the root object.

4.2 Initializing PMT

4.2.1 Loading Default Schema

Recall from section 2.5 XML Schema that PMT is initialized with the use of one or more XML Schema files. In the case of when you're working with traditional image metadata, such as the metadata commonly found in Exif and TIFF files, initializing PMT requires only an API call to

load the **default Schema**. The default Schema is hidden from the PMT user. When a call to initialize PMT is made in this fashion...

```
PmtLogicalDefinitionInterpreter interpreter;
interpreter.load();
```

…the PMT loads the default Schema -- a compiled-in version of a Schema that contains the appropriate information for initializing PMT to work with traditional image metadata. (The above way is how we've seen PMT initialized in all the previous code examples.)

4.2.2 Loading Another Schema

In the case when another XML Schema is needed to initialize PMT, then one of two general scenarios exist. Either someone else has already created a new XML Schema file. Or you, as a developer, desire to create your own new XML Schema file to initialize PMT with.

In most cases, someone else will be sharing an already-created Schema for you. In this case, another file, called the Metadata Definition Information File (MDIF) should be provided for you too. (The MDIF file format is covered in section 4.3 Metadata Definition Information File.) Find out from the appropriate person who provided you with the MIDF and Schema, what information is needed for properly installing the MDIF and XML Schema. All you need to know, is what directories are the appropriate ones for the MDIF and Schema to reside in, relative to your application.

Then, the only other thing you need to determine, is what argument needs to be passed to an overloaded **load(…)** method on the PmtLogicalDefinitionInterpreter interface. The argument passed is the relative directory path to the MDIF, from where your application is executing.

For example:

```
PmtLogicalDefinitionInterpreter interpreter;
interpreter.load("MySchemaInfoFile.xml");
```

The above example assumes the **MySchemaInfoFile.xml** MDIF file resides in the same directory as the application that's using PMT.

If you need to create you own MDIF file, refer to section 4.3 Metadata Definition Information File. If you want to create your own XML Schema file, refer to section 4.4.1 Defining Additional Metadata – Creating New Schema.

4.2.3 Loading Aliases – Default Schema

The default Schema can optionally be loaded with an aliases file. To do this, simply call one of the overloaded **loadWithAliases(…)** methods on the PmtLogicalDefinitionInterpreter class. For example:

```
PmtLogicalDefinitionInterpreter interpreter;
interpreter.loadWithAliases("MyAliasesFile.xml");
```

The argument passed in the relative directory path to the aliases file, from where your application is executing. The above example assumes the **MyAliasesFile** aliases file resides in the same directory as the application that's using PMT.

Since initializing PMT this way loads the default Schema, a call to the **load(…)** that takes on arguments would not be required in addition to this call. The above example of **loadWithAliases(…)** already deals with initializing PMT for working with traditional image metadata.

4.2.4 Loading Aliases – Another Schema

To load another Schema and aliases file, place the Schema and aliases file into the appropriate directories. Then, simply make a call to the other overloaded version of **loadWithAliases(…)**. This version takes a relative path to the MDIF file, and a relative path to the aliases file. Both paths are relative from where your application is executing. For example:

```
PmtLogicalDefinitionInterpreter interpreter;
interpreter.load("MySchemaInfoFile.xml",
                 "MyAliasesFile.xml");
```

The above example assumes the **MySchemaInfoFile.xml** MDIF file and **MyAliasesFile.xml** file reside in the same directory as the application that's using PMT.

4.2.5 Loading In-Memory Schema

It is possible for another application to compile-in a Schema and/or aliases file, and to initialize PMT with the compiled-in copies. The reader is referred to the PmtInterpreterTest.cpp file, that comes in any of the PMT distributions, for example code on how this is done. Particularly, the function **testLoadMemory()** is what performs this type of PMT initialization. The method **loadMemory()** is the method called on the PmtLogicalDefinitionInterpreter interface.

4.3 Metadata Definition Information File

This section covers the format of a MDIF file. Understanding the format of a MDIF is necessary, when you create your own new XML Schema file. You should always create a new MDIF, and provide it along with your new Schema, so others can properly initialize PMT. (Refer to section 4.2 Initializing PMT, for specifics on initializing PMT with a MDIF file.)

The Metadata Definition Information File (MDIF) is used to convey additional information associated with an XML Schema file to PMT. Primarily, the MDIF associates a Schema's root element, and an Accessor Translation Table together.

The root element requires identification because of a quark in XML Schema. XML requires one and only one element appear at the root level of an instance file. For example, the following is a well-formed XML instance file:

```
<?xml version="1.0" encoding="UTF-8"?>
<rootLevelElement>
    <item1> … </item1>
    <item2> … </item2>
</rootLevelElement>
```

while the following XML instance is not well-formed since it contains a second element at the root level (the element with the tag *<rootElementIllegal>*):

```
<?xml version="1.0" encoding="UTF-8"?>
<rootLevelElement>
    <item1> … </item1>
    <item2> … </item2>
</rootLevelElement>
<rootElementIllegal>
    <item1> … </item1>
    <item2> … </item2>
</rootElementIllegal>
```

The quark in XML Schema is that there is no restriction on the number of elements that can be declared at the root level nor is there a means to identify which of the root element declarations is intended to be the "true" root.

The following is an example of the MDIF:

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

    <MetadataDefinitionBinding>
        <MetadataDefnFileURI> Another.xsd </MetadataDefnFileURI>
        <RootElementName> AnotherRoot </RootElementName>
        <AssociatedTranslationTableURI>
            AnotherTranslationTable.xml
        </AssociatedTranslationTableURI>
    </MetadataDefinitionBinding>

    <MetadataDefinitionBinding>
        <MetadataDefnFileURI> Local.xsd </MetadataDefnFileURI>
        <RootElementName> LocalRoot </RootElementName>
    </MetadataDefinitionBinding>

</MetadataDefinitionBindings>
```

The MDIF allows for one or more metadata definition bindings to be identified within the root element ***<MetadataDefinitionBindings>***. Each individual binding is enclosed within the tag ***<MetadataDefinitionBinding>***. For each binding, a URI to the MDF must be specified through the content of the ***<MetadataDefnFileURI>*** tag. The content of the ***<RootElementName>*** tag identifies the name attribute's value of the element in the MDF that declares the root element. The associated accessor translation table file is identified through the content of the optional tag ***<AssociatedTranslationTableURI>.***

4.4 Extending PMT

There are at least two general ways in which the functionality of PMT may be extended. The first is to define unique (or additional) metadata objects for PMT's use. The second general way that PMT can be extended it to support one or more additional file formats - either XML instance documents or binary file formats. This section further discusses each of these two general ways.

4.4.1 Defining Additional Metadata – Creating New Schema

If the metadata you would like to use is not in PMT's default Schema (the default Schema is discussed in section 4.2.1 Loading Default Schema above), or in another Schema that already exists, then it is necessary to create a new Schema file containing the new metadata items of interest. This allows the initialization of PMT, with the appropriate Schema(s), so PMT is enabled to work with the additional metadata. Remember: defining additional metadata for PMT means creating a new Schema file.

4.4.1.1 XML Schema

XML Schema, a W3C Recommendation, is used by PMT to specify the logical layout of metadata (as discussed in section 2.3.2.1 Hierarchical Groupings). PMT has adopted the use of XML Schema since it provides a rich syntax for hierarchical structure, is an open standard, and a W3C Recommendation.

Creating a new Schema for PMT requires basic knowledge of XML and XML Schema. Please refer to [3][4][5][6] for details on XML and XML Schema. For many situations, understanding XML Schema in great detail is not necessary. The more basic syntax of XML instance documents and XML Schema is often all that's a prerequisite to making a new Schema for PMT. Also, starting with an existing Schema, and modifying it to create a new one, can often be useful.

PMT parses a Schema file when it's initialized. Initialization of PMT includes any of the **load(…)**, **loadWithAliases(…)**, or **loadMemory(…)** methods being called on the **PmtLogicalDefinitionInterpreter** class interface. (For traditional image metadata, the XML Schema is actually provided in a compiled-in form, and is used when the **load()** that takes no parameters is called.)

A Schema does two primary things: it defines the logical layout of metadata (as discussed in section 2.3.2.1Hierarchical Groupings), and it defines the types of values that can be held in metadata objects.

4.4.1.2 XML Schema Example & Mappings

Understanding how PMT parses a Schema file is germane to creating a new Schema file. Consider the following short sample Schema file, used only for illustrative purposes (It bears repeating at this time that the following assumes basic knowledge of XML and XML Schema. Please refer to [3][4][5][6] if you need details.)

```
1     <?xml version="1.0" encoding="UTF-8"?>
2     <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3         <xsd:complexType name="PmtRootElementT">
4             <xsd:sequence>
5                 <xsd:element name="CaptureConditions" type="CaptureConditionsT"/>
6                 <xsd:element name="ImageContainer" type="ImageContainerT"/>
7             </xsd:sequence>
8         </xsd:complexType>
9
10        <xsd:complexType name="CaptureConditionsT">
11            <xsd:sequence>
12                <xsd:element name="Aperture" type="xsd:float"/>
13                <xsd:element name="Contrast" type="xsd:unsignedByte"/>
14            </xsd:sequence>
15        </xsd:complexType>
16
17        <xsd:complexType name="ImageContainerT">
18            <xsd:sequence>
19                <xsd:element name="Height" type="xsd:unsignedInt"/>
20                <xsd:element name="Thumbnail" type="ThumbnailT"/>
21                <xsd:element name="Width" type="xsd:unsignedInt"/>
22            </xsd:sequence>
23        </xsd:complexType>
24
25        <xsd:complexType name="ThumbnailT">
26            <xsd:sequence>
27                <xsd:element name="Height" type="xsd:unsignedInt"/>
28                <xsd:element name="Width" type="xsd:unsignedInt"/>
29            </xsd:sequence>
30        </xsd:complexType>
31
32        <xsd:element name="PmtRootElement" type="PmtRootElementT"/>
33    </xsd:schema>
```

The above example contains four **complexType** definitions: **PmtRootElementT**, **CaptureConditionsT**, **ImageContainerT**, and **ThumbnailT**. The complexType declarations begin on lines 3, 10, 17, and 25 respectively. The example also contains ten **element** declarations, including CaptureConditions, ImageContainer, Aperture, Contrast, Height, etc. The element declarations are on lines 5, 6, 12, 13, 19, etc.

When PMT parses the above Schema, each of the **complexType** declarations **map** to a PmtCompositeMetadata object. The term **map** means that PMT is capable of creating the appropriate PmtCompositeMetadata object(s) when a subsequent request for metadata is made via the **getMetadatum(…)** or **getMetadata(…)** methods on the PmtMetadata interface. That is, during initialization time, each complexType declaration in the Schema results in PMT being setup properly to allow subsequent creation of the corresponding PmtCompositeMetadata objects.

(In other Schemas, it is possible that one or more particular complexType declarations are unused – they're declared but never used elsewhere in the Schema. In that case, the complexType is ignored by PMT, and no mapping occurs.)

This same concept of mapping applies to the other things specified in a Schema. Each of the **element** declarations in a Schema result in PMT being setup to allow subsequent PmtMetadata object creations.

Each **element (**whose type is a simpleType for purposes of this discussion) maps to a PmtMetadataT object. Another way to say this is: if an element declaration results in a single Schema object, then a PmtMetadataT object is mapped. If the element's type is a complexType, then a PmtCompositeMetadata object is mapped. (There is an unusual exception to this case, but for purposes of this discussion, it can be ignored.)

A separate document located in the **doc** directory of the PMT distribution, entitled PmtTypeTable.pdf, lists the supported simple types and their mapping to C++ types. (XML Schema allows for a simple type to be defined as a list of some simple type. This list construct is mapped into an STL vector of the type.)

Each complexType declaration also corresponds to the beginning key segments in a key. For example, in the full key **CaptureConditions.Aperture**, **CaptureConditions** is a complexType in the Schema, and a PmtCompositeMetadata object in PMT. (It may be helpful to review section 2.4 PmtMetadata Objects and Metadata Key Mappings.)

Each element declaration also corresponds to the end-most key segments in a full key, when that object contains a value. **Aperture** is an element in the Schema, and a PmtMetadataT object in PMT.

The complexType declarations are used to logically group similar metadata objects together. Since they map to PmtCompositeMetadata objects, they do not contain values. The element declarations that have a primitive type, and thus represent a single value, are used to declare metadata objects that contain values.

To further illustrate the logical hierarchy in the above Schema example, notice that within the **ImageContainer** object, that it contains a **Height** and a **Width**. It also contains a **Thumbnail** object, which in turn also contains a **Height** and a **Width**. This means that some of the legitimate keys are **ImageContainer.Height**, **ImageContainer.Width**, **ImageContainer.Thumbnail.Height**, and **ImageContainer.Thumbnail.Width**. The **Height** and **Width** in the **Thumbnail** are different from the **Height** and **Width** contained directly within **ImageContainer**. They contain different values.

To emphasize an important principle between Schema files and PmtMetadata objects: element declarations determine the metadata keys that are used in working with PmtMetadata objects. Specifically, the value associated with the name attribute in a XML Schema <element…> declaration determines exactly the key name used in PMT. For example, the declaration

```
<xsd:element name="CaptureConditions" type="CaptureConditionsT"/>
```

means that the key **CaptureConditions** (from **name="CaptureConditions"**) will be the key name for the PmtMetadata object that happens to be a direct child underneath the root PmtMetadata object.

4.4.1.3 Creating New XML Schema

When creating a new Schema, it is suggested to use **complexType** declarations as appropriate, to logically group your metadata items together. Then, use **element** declarations with simple types,

to define the metadata items that actually contain values. Keep in mind that the complexType declarations will map to PmtCompositeMetadata objects in the code, and that element declarations with simple types will map to PmtMetadataT<TYPE> objects.

In many situations, understanding the information covered already in this document, and looking at some of the Schemas in PMT's test programs and how the test programs' code manipulates the corresponding PmtMetadata objects, will be all you need to know to create your own new Schema. For many Schemas, primarily using complexType and element declarations will suffice in creating a new Schema layout. Some of the additional features used by PMT in a Schema become apparent quickly when looking at the examples provided with the test programs. (Refer to section 6.1.5 Testing PMT Installation, for more on the test programs.)

The additional features of Schema syntax supported by PMT are not explicitly covered in this document. Analyzing PMT's source code, looking further into PMT's test programs, and learning more about XML Schemas (starting at references [3][4][5][6]), is left to the responsibility of the developer.

Any Schema file to be used by PMT must be a valid Schema, as defined by the W3C. A tool such as XML Spy can quickly determine if a new Schema you've created is valid or not. (A trial version of XML Spy can be downloaded at http://www.xmlspy.com.) Other validation tools are available. Refer to http://www.w3.org for more.

Currently, a sub-set of XML Schema syntax is supported by PMT. Appendix A provides two tables outlining the XML Schema constructs that are supported. Also, PMT supports Schema syntax as outlined in the MetadataDefinitionsBestPractice-v1_0.pdf file, available in the **doc** directory of a PMT distribution. This document discusses suggested practices in creating one or more new Schema files. PMT is aimed at working with Schema that follow the practices discussed in this document. Schema files that do not conform to these practices are not supported by PMT.

4.4.2 Support Additional File Formats

If you only need to persist your metadata to XML instance documents (and not to a binary file format), then keep in mind that XML persistence is already built into PMT. If you have a Schema with the appropriate metadata available, then simply initialize PMT with it. Then, use the persistence mechanisms available in either the PmtAccessor interface (covered in section 2.6 Accessors) or the overloaded IO stream operators (covered in section 3.7 XML Instance Serialization).

If the new file format you would like to support is a binary file format, then implementing a new PmtAccessor specialization is necessary. Examples of current PmtAccessor specializations in PMT can be found in the **src/PmtAccessor** directory. Implementation of a new PmtAccessor is not covered in this document. Analyzing that task through inspection of the PMT source code is left to the responsibility of the developer.

4.5 Unicode Builds

PMT can be built into Unicode versions on the Windows platform. The Visual Studio project files that come in the PMT distribution have Unicode configurations built into them. These configurations have the **_UNICODE** value defined in them, by passing the **/D "_UNICODE"** option to the compiler.

Please note that although PMT may be built into various Unicode configurations, that the supported status of what data can be Unicode data in PMT has not yet been fully determined. The details of what data should be Unicode enabled needs to be defined.

4.5.1 Unicode Build Helpers

There are a couple of convenience mechanisms for writing code that works easier with a Unicode build of PMT. This section covers those mechanisms.

4.5.1.1 EK_L Macro

An **EK_L** #define'd macro exists as a convenience for specifying string values that can be either a Unicode string, or a regular ASCII string. For example:

```
mdHandle->setValueStr(EK_L("a string"));
```

The above code example, taken from the AccessorTest.cpp file, illustrates the use of EK_L. EK_L takes one parameter, which is a quoted string. If you are building a Unicode version of your code, make sure the _UNICODE #define value is set in your Visual Studio project file. This is usually done by passing in **/D "_UNICODE"** setting to the compiler. This causes the EK_L macro to prefix the quoted string with the letter L. That is, **"EKC"** becomes **L"EKC"**.

On regular non-Unicode builds, the EK_L has no effect. So **"EKC"** simply stays **"EKC"**.

4.5.1.2 EkString

EkString is a typedef that allows you to conveniently declare STL string objects that are appropriate for Unicode or non-Unicode builds. If your build is for Unicode, EkString is a **wstring.** If your build is for non-Unicode, EkString is a regular STL **string**.

5. Implementation Details

5.1 Accessor Implementation

5.1.1 Accessor Translation Table

The Accessor Translation Table (ATT) contains the information necessary for Pmt's accessor facility to know how to translate a common metadata instance into an image file format specific instance and vice versa. The ATT is captured in XML, as the following example illustrates:

```
<?xml version="1.0"?>
<!DOCTYPE TRANSLATION []>
<TRANSLATION CreatedTime="Mon Aug  7 14:53:36 2000">
<ENTRY Key="CaptureConditions.Aperture" >
    <EXIF Tag="37378" Type="urational" Location="APP1_IFD0.EXIF_IFD"
                Translator="builtin2float"/>
    <TIFF Tag="37378" Type="urational" Location="IFD_MAIN"
                Translator="builtin2float">
        <TIFF Tag="37378" Type="urational" Location="IFD_EXIF"
                Translator="builtin2float"/>
    </TIFF>
</ENTRY>
<ENTRY Key="CaptureConditions.BatteryLevel" >
    <TIFF Tag="33423" Type="urational" Location="IFD_MAIN"
                Translator="builtin2float"/>
</ENTRY>
<ENTRY Key="CaptureConditions.Brightness" >
    <EXIF Tag="37379" Type="rational" Location="APP1_IFD0.EXIF_IFD"
                Translator="builtin2float"/>
    <TIFF Tag="37379" Type="rational" Location="IFD_MAIN"
                Translator="builtin2float">
        <TIFF Tag="37379" Type="rational" Location="IFD_EXIF"
                Translator="builtin2float"/>
    </TIFF>
</ENTRY>
</TRANSLATION>
```

The ***<TRANSLATION>*** tag is the root level element for the translation table and has an optional attribute ***CreatedTime*** to capture the time of the creation of the table. Each element of the table begins with the ***<ENTRY>*** tag. The ***Key*** attribute of the ***<ENTRY>*** tag identifies the pertinent metadata item via its qualified key. The content of the ***<ENTRY>*** tag identifies how to access the metadata in various file formats and how to perform the necessary type translation.

The attributes of the file format tags capture the access and translation information. The attributes are: Tag, Type, Location, and Translator. The appropriate accessor defines their use. With respect to the Exif and TIFF accessors, the **tag** attribute is the numerical identifier for the metadata. The **type** attribute specifies the metadata's type as stored in the file. The valid values for the Type attribute are:

Type	**Attribute Value**	
	signed	**Unsigned**
character (8 bit integer)	byte	Ubyte
short (16 bit integer)	short	Ushort
long (32 bit integer)	long	Ulong
rational	rational	Urational
float (single precision real)	Float	
double (double precision real)	Double	
Ascii text	Ascii	
Undefined (not a simple type)	Undefined	
Audio stream	Audio	

The **`location`** attribute specifies the location of the metadata within the image file format's structure. In the case of Exif, the valid **`location`** values are: ***APP1_IFD0, APP1_IDD0.EXIF_IFD, APP1_IFD1, APP3_IFD0,*** and ***APP4_IFD0***. The valid Location values for Tiff are: ***IFD_MAIN, IFD_SUB, IFD_EXIF, IFD_SOUND, IFD_GPS,*** and ***IFD_EXIF_INTER***. The last attribute is Translator, which specifies the type translator to use. Pmt provides a set of translators for the built in types of the C++ compiler. In general, the name of these translators is *"builtin2?formatType?"*, where *?formatType?* is replaced with format type attribute value from the table above. For example, if the format type is "*long"* then the translator is *"builtin2long"*. In addition, users may implement their own translators and register them with the translator factory. See the usage of the EK_PROVIDE and EK_REQUIRE macros in the top of the **PmtBuiltInTranslators.cpp** file for how the translator factory is used.

The ***<TIFF>*** tag may also have contain another ***<TIFF>*** tag. This contained ***<TIFF>*** tag indicates that the particular metadata may be found in another location in the Tiff file.

6. Compiling & Linking with PMT

This section is organized into sub-sections that give the appropriate steps needed to compile & link with PMT on its supported platforms. Windows users should refer to section 6.1 Windows Platform. Linux/UNIX users should refer to section 6.2 Linux/UNIX Platforms.

6.1 Windows Platform

6.1.1.1 To work with the Windows version of PMT, you should first follow all the instructions in section 6.1.2 First Steps. Then, depending on if you are working with a Binary or Source distribution of PMT, follow the instructions in one of the appropriate sections: 6.1.2.7 Configuration File

When building PMT on Windows, most configuration options are done via a configuration file. This file is located in the **src/PmtCore** directory and is called **"PmtConfig.h".** For example, this would be where you decide if you want an XML parser, and if so, which one. There are numerous options that can be set, and more are added all the time. The file is well documented so please refer to it if you desire any configuration that may be out of the norm.

Binary Distribution or 6.1.4 Source Distribution. Lastly, test your installation by following the instructions in section 6.1.5 Testing PMT Installation.

6.1.2 First Steps

6.1.2.1 Visual C++ 6.0, 7.0, or 7.1

- On Windows, there is support for building with either VC6 with SP5, VC7 (.Net 2002) and VC7.1 (.Net 2003). If you are using VC6 and are not certain that SP5 is installed, you can use the information found at the following URL to determine if it is: http://support.microsoft.com/default.aspx?scid=kb;en-us;Q194295

If you need to download the SP5 service pack, go to the following URL: http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/sp5_dwnld.asp

6.1.2.2 Download & Unzip PMT

- If you have not already done so, download the latest released version of PMT from http://sourceforge.net/projects/picturemetadata. For Windows, there are binary distributions for VC7.1. If you want to modify the PMT source code, or would prefer to build the source code yourself, then download the source distribution. If you want to use the PMT libraries, but do not want to work with the source code, then you should download the binary distribution.

- Unzip the PMT distribution into the directory of your choice on your computer. The directory you unzip PMT to is referred to as the **PMT distribution** directory throughout this document.

6.1.2.3 XML Parser

- There is support for building with MSXML or Xerces-C version 2.2.0. You can also build without an XML parser, but there will be limited functionality (you will only be able to use the default metadata schema and translation table).

- For Xerces, download the binary or source version 2.2.0 of the Xerces parser from: http://xml.apache.org/dist/xerces-c/stable/archives/. If using the source distribution, follow Xerces' instructions to build it.

6.1.2.4 OpenTiff

The current version of PMT supports OpenTiff version 1.2.

These instructions only apply when downloading and building a source distribution. OpenTiff binaries are included in the binary distribution's Toolkits\ directory.

Follow these steps to download and test OpenTiff. OpenTiff is a library used by PMT to access the metadata in TIFF image files.

- Download the Open Source toolkit **OpenTiff**. OpenTiff is located on the SourceForge web site at: http://sourceforge.net/projects/opentiff. You can download a source or a binary distribution.

- Unzip the OpenTiff distribution into the directory of your choice on your computer. The directory you unzip OpenTiff to is referred to as the **OpenTiff distribution** directory throughout this document.

- If you've downloaded the source distribution, follow the instructions in OpenTiff's readme.html file to build it.

- Build and run the test program in the test\ directory and compare your output to TestOut.txt.

6.1.2.5 OpenExif

The current version of PMT supports OpenExif version 1.5

These instructions only apply when downloading and building a source distribution. OpenExif binaries are included in the binary distribution's Toolkits\ directory.

Follow these steps to download and test OpenExif. OpenExif is a library used by PMT to access the metadata in Exif formatted jpeg image files.

- Download the Open Source toolkit **OpenExif**. OpenExif is located on the SourceForge web site at: http://sourceforge.net/projects/openexif. You can download a source or a binary distribution.

- Unzip the OpenExif distribution into the directory of your choice on your computer. The directory you unzip OpenExif to is referred to as the **OpenExif distribution** directory throughout this document.

- If you've downloaded the source distribution, follow the instructions in OpenExif's readme.html file to build it. Note that you will also need to download and build the Independent Jpeg Group (IJG) toolkit from http://www.ijg.org.

- Build and run the test program in the test\ directory and compare your output to GroundTrue.txt.

6.1.2.6 Environment Variables

Setup the following environment variables on your computer.

- Point your **Path** system environment variable to the **lib** directory in the PMT distribution. For example, if you unzipped the PMT distribution into a directory called C:\Pmt, then you would point your Path to: C:\Pmt\lib

- EXIFROOT - This points to the OpenExif toolkit directory.
  For example: C:\CvsNow\Pmt\Toolkits\openexif

- JPEGROOT - This points to the IJG jpeg toolkit directory.
  For example: C:\CvsNow\Pmt\Toolkits\jpeg

- TIFFROOT - This points to your OpenTiff toolkit directory.
  For example: C:\CvsNow\Pmt\Toolkits\opentiff

If you are building with Xerces, then you also need to:

- Setup your computer's **Path** environment variable to point to the directory that contains the **xerces-c_2_2_0.dll** file that is in your Xerces distribution. For example, if you unzipped Xerces into a directory called C:\Xerces, you would point your Path to: C:\Xerces\xerces-c_2_2_0-win32\bin (version 2.2)

- XERCESCROOT - This points to the Xerces directory.
  For example: C:\Xerces\xerces-c_2_2_0-win32

6.1.2.7 Configuration File

When building PMT on Windows, most configuration options are done via a configuration file. This file is located in the **src/PmtCore** directory and is called **"PmtConfig.h".** For example, this would be where you decide if you want an XML parser, and if so, which one. There are numerous options that can be set, and more are added all the time. The file is well documented so please refer to it if you desire any configuration that may be out of the norm.

6.1.3 Binary Distribution for the latest Windows compiler

If you have correctly followed the instruction in section 6.1.2 First Steps, then PMT should be properly installed and ready to test. Although you are using a binary distribution of PMT, building the test programs from their source code is part of testing PMT. (The source code for the test programs comes in the binary distribution.) Follow the instructions in section 6.1.5 Testing PMT Installation to build the test programs and test your PMT installation.

6.1.4 Source Distribution

6.1.4.1 Build Instructions

- In Visual C++, open the **BuildAllStatic.dsw**, or **BuildAllStatic.sln** for VC7 and VC7.1, workspace in the **projects\Win32\Vc6.0**, **projects\Win32\VC7.0** for VC7 and **projects\Win32\VC7.1** for VC7.1, directory of the PMT distribution. This will allow you to build static libraries for PMT. (For building dynamic libraries, you would open **BuildAllDynamic.dsw**, **BuildAllDynamic.sln** for VC7 and VC7.1.)

  PMT libraries come in: debug, release, static, dynamic, non-unicode and unicode builds. The **BuildAllStatic** workspace or solution contains the necessary configurations for all of the static library builds. The **BuildAllDynamic** workspace or solution contains all of the configurations associated with dynamic library (.dll) versions of PMT.

- From the Microsoft's Visual C++ menu, select **Build, Batch Build…**. Make sure all the project configurations have their check boxes checked. Select the **Rebuild All** button.

Note: There is an apparent bug in Microsoft Visual C++ 6.0 that causes some of the PMT libraries to build with the error **LNK1561: entry point must be defined**. This error happens only the first time PMT is built. Simply select the **Build** button again from the **Build, Batch Build…** dialog. This will correctly build each library to completion.

Using the test programs that come with PMT can verify that things built correctly.

6.1.5 Testing PMT Installation

There are two test programs that should be built and executed to verify the correct functioning of the PMT libraries. Their names are **PmtInterpreterTest** and **AccessorTest**. They both reside in sub-directories under the **test** directory in your PMT distribution. Regardless as to whether you're working with a binary or source distribution of PMT, make sure you build and run both test programs.

Although the test programs should already be built if you've built all of PMT in a source distribution (with one of the workspaces in the **projects** directory) it is suggested that you complete the following steps for building the test programs below, before executing them. This will ensure that the proper code is built, regardless as to whether you're working with a binary or source distribution of PMT.

6.1.5.1 PmtInterpreterTest Program

Build and test the PmtInterpreterTest program first. The PmtInterpreterTest program relies on two of the three major libraries built by PMT: the libraries **PmtCore.lib** and **Ek.lib** (these are the static, release version names). The PmtInterpreterTest program exercises the PmtLogicalDefinitionInterpreter and PmtMetadata interfaces in detail. No image files are read from or written to in the PmtInterpreterTest program.

- Open up the PmtInterpreterTest workspace or solution file in Microsoft Visual C++. It resides in the **projects\Win32\VC*.*** directory.

  To work with the dynamic (.dll) versions of the PMT libraries instead of the static libraries, you would need to use the **PmtInterpreterTesti** workspace or solution.

- Select an appropriate configuration. For example, from the Microsoft's Visual C++ menu, select **Build, Set Active Configuration…** (**Configuration Manager…** in VC7); then select the **PmtIntepreterTest – Win32Release** configuration. This will build all the appropriate code in the test program (and if you're working with a source distribution of PMT, the workspace will build PMT itself, if needed) for the selected configuration. In this example, the static, release code will be built.

- From the menu, select **Build, Build PmtInterpreterTest.exe**. This will build the test program.

- Inspect the build results. There should be no build errors. If there are errors, then make sure you have followed all the required steps discussed in this document before building PMT. If you are using VC6, also make sure you have the required service pack installed as discussed in section 6.1.2.1 Visual C++.

- Bring up an MS-DOS Command Prompt, making the current directory the **test\PmtInterpreterTest** directory.

- From that directory execute the command: **release\PmtInterpreterTest > look.txt**

Note that the name of the directory **release** above would be changed to **debug** or some other directory name, depending on which configuration of the test program you've built and are testing. Executing the test programs from one directory above where its executable resides is necessary, since various files used by the test program are searched for in the current directory. Those files used by the test program in the above example reside in **test\PmtInterpreterTest** (not **test\PmtInterpreterTest\release**).

Notice that output from the test program is redirected into the file named **look.txt**. This file can then be compared to the file named **GroundTrue.txt** in a diff utility like Microsoft's WinDiff. The output of the two files should be identical (except for the date at the top and time at the bottom). If the output matches correctly, the test has been executed successfully.

NOTE: If you are building without an XML parser, then the test will not run successfully.

6.1.5.2 AccessorTest Program

The Accessor test program relies on more libraries than PmtInterpreterTest does, since the AccessorTest uses PMT's third major library too, named **PmtAccessor.lib** (release version name) and uses a few image file toolkit libraries (the ones pointed to by some of the environment variables previously set).

Since the environment variables you've set up on your computer for PMT are accessed during the building of the AccessorTest program, incorrect settings for these variables may manifest during the building process. If you encounter build errors, one of the first things to check is the proper setting of your PMT environment variables.

The AccessorTest program exercises reading from and writing to image files, via use of the PmtAccessor interface. Since using the PmtLogicalDefinitionInterpreter and PmtMetadata interfaces is always necessary in using PMT, those things happen somewhat in the AccessorTest program too. However, the testing of those things in detail is done primarily in the PmtInterpreterTest program.

- Open up the workspace **AccessorTest** workspace or solution file in Microsoft Visual C++. It resides in the **projects\Win32\VC*.*** directory.

- Select an appropriate configuration, as you did with PmtInterpreterTest. For example, select the **AccessorTest – Win32 Release** configuration.

- From the menu, select **Build, Build PmtInterpreterTest.exe**. This will build the test program.

- Inspect the build results. There should be no build errors. If there are errors, then make sure you have followed all the required steps discussed in this document before building PMT. If you are using VC6, make sure you have the required service pack installed as discussed in section 6.1.2.1 Visual C++. Also ensure your PMT environment variables have been properly set. Resolve any build errors, if any, before continuing.

- Bring up an MS-DOS Command Prompt, making the current directory the **test\AccessorTest** directory.

- From that directory execute the command: **release\AccessorTest > look.txt**

As with the PmtInterpreterTest, note that the name of the directory **release** above would be changed to **debug** or some other directory name, depending on which configuration of the test program you've built and are testing.

Also similar to the PmtInterpreterTest, the output is redirected to a file named **look.txt**. If the output matches the **GroundTrue.txt** file, the test has been executed successfully.

6.2 Linux/UNIX Platforms

To work with the Linux/UNIX version of PMT, you should follow all the instructions in this section (this includes sub-sections 6.2.1 First Steps and 6.2.2 Building Source). The Linux/UNIX distribution comes only in source code form - there's no binary distribution.

6.2.1 First Steps

6.2.1.1 Unix Tools

The Linux/UNIX build environment uses the GNU auto configuration facility. The `configure' shell script that comes in the PMT distribution attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile' in each directory of the package that contains a Makefile.in (and Makefile.am). It also creates a shell script `config.status' that you can run in the future to recreate the current configuration, a file `config.cache' that saves the results of its tests to speed up reconfiguring, and a file `config.log' containing compiler output (useful mainly for debugging `configure').

We support the following compilers:

- GCC version 2.95.2, or greater

6.2.1.2 OpenTiff

Follow these steps to download, build, and test OpenTiff. OpenTiff is another library used by PMT to access the metadata in TIFF image files.

- Download the Open Source program **OpenTiff**. OpenTiff is located on the SourceForge web site at: http://sourceforge.net/projects/opentiff.

- Unzip the OpenTiff distribution into the directory of your choice on your computer. The directory you unzip OpenTiff to is referred to as the **OpenTiff distribution** directory throughout this document.

OpenTiff uses the GNU auto configuration facility. More about this facility is discussed in the context of building PMT. For now, to build OpenTiff, follow these steps:

- Bring up a shell prompt, making the current directory the OpenTiff distribution directory.
- Configure OpenTiff for building by executing the **configure** script. The configure script optionally takes a **--prefix** option that specifies where the OpenTiff package will be subsequently installed. By default, the package would be installed in the **/usr/local** directory. So if you do not have sufficient privileges for **/usr/local**, you can specify another directory location (such as one in your home directory) where you do have sufficient privileges.

  For example, from your shell's command prompt execute: **./configure**
  or, to specify another directory where OpenTiff will subsequently be installed:
  ./configure --prefix=/path/to/install
- After successful configuration, make OpenTiff. From your prompt, execute: **make**

- Make the current directory the **test** directory. From your prompt, execute: **cd test**
- Execute the test program. From that directory execute the command: **./tiffTest**

  Successful completion of the test program indicates a good build of OpenTiff.

- Install the OpenTiff software. From your prompt, execute: **make install**

  This installs the software into either the **/usr/local** directory, or the directory you specified above with the **--prefix** option passed to **./configure**.

6.2.1.3 OpenExif

Follow these steps to download, build, and test OpenExif. OpenExif is another library used by PMT to access the metadata in Exif formatted jpeg image files.

- Download the Open Source toolkit **OpenExif**. OpenExif is located on the SourceForge web site at: http://sourceforge.net/projects/openexif.

- Unzip the OpenExif distribution into the directory of your choice on your computer. The directory you unzip OpenExif to is referred to as the **OpenExif distribution** directory throughout this document.

OpenExif uses the GNU auto configuration facility. More about this facility is discussed in the context of building PMT. For now, to build OpenExif, follow these steps:

- Bring up a shell prompt, making the current directory the OpenExif distribution directory.
- Configure OpenExif for building by executing the **configure** script. The configure script optionally takes a **--prefix** option that specifies where the OpenExif package will be subsequently installed. By default, the package would be installed in the **/usr/local** directory. So if you do not have sufficient privileges for **/usr/local**, you can specify another directory location (such as one in your home directory) where you do have sufficient privileges.

  For example, from your shell's command prompt execute: **./configure**
  or, to specify another directory where OpenTiff will subsequently be installed:
  ./configure --prefix=/path/to/install
- After successful configuration, make OpenExif. From your prompt, execute: **make**

- Make the current directory the **test** directory. From your prompt, execute: **cd test**

- Execute the test program. From that directory execute the command: **./exifTest**

- Successful completion of the test program indicates a good build of OpenExif.

6.2.1.4 XML Parser

- Download the binary version 2.2.0 of the Xerces parser (or download and build the source) from: http://xml.apache.org/dist/xerces-c/stable/archives/

- You can also build without any XML parser using the `–with-xml=none` configuration option. This will limit you to using the default metadata schema and translation tables, which is the case for typical applications, including our example programs.

6.2.1.5 Environment Variables

Setup the following environment variables on your computer.

- TIFFROOT - This points to the directory where OpenTiff was installed (not unzipped).
  For example, if **./configure** was run with no **--prefix** parameter, then it could be set to:
  /usr/local
  Or, if it was run with a --**prefix=/path/to/install** option:
  /path/to/install

- EXIFROOT - This points to the directory where OpenExif was installed (not unzipped).
  For example, if **./configure** was run with no **--prefix** parameter, then it could be set to:
  /usr/local
  Or, if it was run with a --**prefix=/path/to/install** option:
  /path/to/install

- JPEGROOT - This points to the directory where the Jpeg toolkit is installed. Some Linux distributions have it installed in /usr/bin. Otherwise, it is where you installed it when building OpenExif.

- XERCESCROOT - This points to the Xerces distribution directory. This is not necessary if you're building without an XML parser.
  For example: /home/rupe/xerces-c_2_2_0-linux

- LD_LIBRARY_PATH =
  $XERCESCROOT/lib:$TIFFROOT/lib:$EXIFROOT/lib:$LD_LIBRARY_PATH

6.2.2 Building Source

- If you have not already done so, download the latest released version of PMT from http://sourceforge.net/projects/picturemetadata.

- Unzip the PMT distribution into the directory of your choice on your computer. The directory you unzip PMT to is referred to as the **PMT distribution** directory throughout this document.

The steps for building PMT are similar to what they are for building OpenTiff and OpenExif. They are as follows:

- Bring up a shell prompt, making the current directory the PMT distribution directory.
- Configure PMT for building by executing the **configure** script. The configure script optionally takes a **--prefix** option that specifies where the PMT package will be subsequently installed. By default, the package would be installed in the **/usr/local** directory. So if you do not have sufficient privileges for **/usr/local**, you can specify another directory location (such as one in your home directory) where you do have sufficient privileges.

  For example, from your shell's command prompt execute: **./configure**
  or, to specify another directory where PMT will subsequently be installed:
  ./configure --prefix=/path/to/install

  There is another notable configure option, that can be used to disable the building of shared libraries. For example:
  ./configure --disable-shared

  Example of a typical configure:
  ./configure --prefix=/path/to/install --disable-shared

For a list of all the configure options, type:
./configure --help

- After successful configuration, make PMT. From your prompt, execute: **make**

- Make the current directory the **test/PmtInterpreterTest** directory. From your prompt, execute: **cd test/PmtInterpreterTest**

- Execute the PmtInterpreterTest program. From that directory execute the command:
  ./interpTest > look.txt
  diff –a --ignore-space-change look.txt GroundTrue.txt

  Successful completion of the test program should show no differences (except for the date and times.)

- Make the current directory the **test/AccessorTest** directory. From your prompt, execute:
  cd ../test/AccessorTest

- Execute the Accessor test program. From that directory execute the command:
  ./accrTest > look.txt
  diff –a --ignore-space-change look.txt GroundTrue.txt

  Successful completion of the test program should show no differences (except for the date and times.)

Note: You can read more about the purposes of these two test programs in the section on building the Windows version of PMT in section 6.1.5 Testing PMT Installation. This section also discusses how to further verify the output of the test programs.

- You may optionally install the PMT software on your system. From your prompt, execute:
  make install

This installs the software into either the **/usr/local** directory, or the directory you specified above with the **--prefix** option passed to **./configure**.

Appendix A. XML Schema Constructs Supported

This Appendix provides two tables to indicate what XML Schema constructs are supported by PMT, and how they are interpreted. Items in **bold** font are currently supported, those in normal font are planned for eventual support, and those in *italicized* font have no support plans.

For information pertaining to the Schema types and how they are associated to the types used in PMT, such as the C++ types, please refer to the document entitled **PmtTypeTable.pdf** that's located in the **doc** directory of the PMT distribution.

Table 1. Metadata type definitions and Metadata declarations

XML Schema	PMT Interpretation

Construct	Attributes	Content	
attribute	form, id, **name**, ref, type, use, value	annotation, simpleType	Attribute associated with a PmtMetadata instance
attributeGroup	id, name, ref	annotation, attribute, attributeGroup, anyAttribute	
complexType	abstract, block, final, id, *mixed*, **name**	annotation, simpleContent, complexContent, *group*, *all*, choice, **sequence**, attribute	Definition the type for a PmtCompositeMetadata.
simpleContent	id	annotation, restriction**,** extension	
restriction	base, id	annotation, simpleType, minExclusive, minInclusive, maxExclusive, maxInclusive, totalDigits, fractionalDigits, length, minLength, maxLength, enumeration, whiteSpace, pattern, attribute, attributeGroup, anyAttribute	
extension	**base**, id	annotation, attribute, attributeGroup	
complexContent	id	annotation, restriction, extension	
extension	**base**, id	annotation, *group*, *all*, choice, sequence, attribute, attributeGroup, anyAttribute	
restriction	base, id	annotation, *group*, *all*, choice, sequence, attribute, attributeGroup, anyAttribute	
element	abstract, block, default, final, form, id, **maxOccurs**, **minOccurs**, **name**, nillable, **ref**, **type**	**complexType**, **simpleType**, key, keyref, unique	Declaration of a metadata instance. The name and type attributes of element become the key and type of a metadata instance, respectively. An element with a complexType is mapped to a PmtCompositeMetadata, an element with a simpleType is mapped to a PmtMetadataT<TYPE>

XML Schema			PMT Interpretation
Construct	**Attributes**	**Content**	
simpleType	id, final, **name**	annotation, restriction, list, union	The type for a leaf metadatum, i.e., a metadatum with a PMT type PmtMetadataT<TYPE>.
restriction	**base**, id	annotation, **simpleType**, minExclusive, **minInclusive**, maxExclusive, **maxInclusive**, totalDigits, fractionalDigits, length, minLength, maxLength, **enumeration**, pattern	
list	id, **itemType**	annotation, **simpleType**	
union	id, memberTypes	annotation, simpleType	
include	**schemaLocation**		Schema at the given location is loaded as part of the current schema
import	**schemaLocation**		Schema at the given location is loaded as part of the current schema. (Note namespaces are not yet supported in PMT, so this is the same as **include**)

Table 2. Facets for simpleTypes

XML Schema			PMT Interpretation
SimpleType Facet	**Attributes**	**Content**	
enumeration	id, **value**	annotation	Facets on PmtMetadataT<TYPE>'s value.
length	id, value, fixed	annotation	
minLength	id, value, fixed	annotation	
maxLength	id, value, fixed	annotation	
minExclusive	id, value, fixed	annotation	
minInclusive	id, **value**, fixed	annotation	
maxExclusive	id, value, fixed	annotation	
maxInclusive	id, **value**, fixed	annotation	
totalDigits	id, value, fixed	annotation	
fractionDigits	id, value, fixed	annotation	
whiteSpace	id, value, fixed	annotation	
pattern	id, value	annotation	

Appendix B. Visitor Design Pattern

As discussed in section 3.5.2 Values via PmtMetadataT<TYPE>, the type specific metadata classes, i.e., *`PmtMetadataT<TYPE>`*, provide methods to get and set the metadata's value. In general, an application will pass the *`PmtMetadata`* base class pointer (*`PmtMetadataPtr`*) around as the handle to an instance. Most methods associated with classes in PMT that deal with metadata communicate the metadata instance in this way. This is advantageous since significant portions of an application that handle metadata actually do not care about the specific type of metadata, i.e., *`PmtMetadataT<short>`*, *`PmtMetadataT<float>`*, etc. all appear to be the same type, *`PmtMetadataPtr`*. Eventually, the application will need to set or get the value of a metadata and, as stated above, the metadata's type (i.e., the type of its value) must be known. Section 3.5.2 presented a run-time method to accomplish this. There are inherent risks in this approach such as performance degradation due to the dynamic cast and complex exception handling. With strongly typed languages such as C++, it is much safer to determine the exact types at compile time. The Visitor Design Pattern is a uniquely suited solution.

Gamma, et al., [7] presents many benefits of the Visitor design pattern. However, Gautier [8] outlines some significant problems with their implementation. He provides an implementation that is much more robust in the face of a changing type hierarchy. Therefore, the implementation presented in [8] has been adopted as the method to represent and implement algorithms to manipulate metadata.

The Visitor design pattern of [8] requires the definition of an abstract base class and parameterized adapter class for each class hierarchy the concrete Visitors are to access. In this case, we are only concerned with the metadata class hierarchy. PMT supplies the abstract base class ***PmtMetadataVisitor*** and abstract class for the implementation, ***PmtMdVisitorImpl***.

```
class PmtMetadataVisitor
{
  public:
    virtual ~PmtMetadataVisitor() {}

    virtual void visit( PmtMetadata& md ) = 0 ;

  protected:
    PmtMetadataVisitor(){}

};

class PmtMdVisitorImpl
{
  public:
    virtual ~ PmtMdVisitorImpl () {}

    virtual void visit( PmtMetadata& md ) = 0 ;
    virtual void visit( PmtCompositeMetadata& md ) = 0 ;
    virtual void visit( PmtMetadataT<int8>& md ) = 0 ;
    virtual void visit( PmtMetadataT<vint8>& md ) = 0 ;
    virtual void visit( PmtMetadataT<uint8>& md ) = 0;
    virtual void visit( PmtMetadataT<vuint8>& md ) = 0 ;
   …
    virtual void visit( PmtMetadataT<float>& md ) = 0 ;
   …
  protected:
    PmtMdVisitorImpl(){}

};
```

The ***PmtMetadataVisitor*** defines a root class that allows for application extension to metadata types beyond the ones provided by PMT. The ***PmtMdVisitorImpl*** class defines the visitor interface for the metadata types provided by PMT. It defines a pure virtual ***visit()*** method for each and every type defined in the metadata class hierarchy. The ***visit()*** method's signature identifies the concrete metadata class that sent the visit request to the visitor. Therefore, within the ***visit()*** method, full access to the specialized metadata interface is available.

The ***PmtSingleMetadataVisitorT*** class is the adapter class for single dispatching, i.e., visits of a single metadata item at a time. Adapting the algorithms through the ***PmtSingleMetadataVisitorT*** isolates the algorithms from the ***PmtMdVisitorImpl*** class, eliminating the need for the algorithms to declare and implement a ***visit()*** method for each specialization of the ***PmtMetadata*** class. Instead, the algorithm only implements ***visit()*** methods for the metadata types in which it is interested.

```
template < class V >
class PmtSingleMetadataVisitorT : public PmtMdVisitorImpl
{
```

```
  public:
    PmtSingleMetadataVisitorT( V& u ) : userVisitor( u ) {}
    virtual ~PmtSingleMetadataVisitorT( void ) {}
    virtual void visit( PmtMetadata& md )
      { userVisitor.visit( md ) ; }
    virtual void visit( PmtCompositeMetadata& md )
      { userVisitor.visit( md ) ; }
    virtual void visit( PmtMetadataT<int8>& md )
      { userVisitor.visit( md ) ; }
    virtual void visit( PmtMetadataT<vint8>& md )
      { userVisitor.visit( md ) ; }
    virtual void visit( PmtMetadataT<uint8>& md )
      { userVisitor.visit( md ) ; }
    virtual void visit( PmtMetadataT<vuint8>& md )
      { userVisitor.visit( md ) ; }
…
     virtual void visit( PmtMetadataT<float>& md )
       { userVisitor.visit( md ) ; }
…
  private:
    V& userVisitor ;
};
```

An additional adapter class, ***PmtDoubleMetadataVisitorT***, allows for double dispatching, i.e., visits of two metadata items at a time.

```
template < class V, class M >
class PmtDoubleMetadataVisitorT : public PmtMdVisitorImpl
{
  public:
    PmtDoubleMetadataVisitorT( V& u, M& fmd )
      : userVisitor( u ), firstMd( fmd ) {}

    virtual ~PmtDoubleMetadataVisitorT( void ) {}

    virtual void visit( PmtMetadata& md )
      { userVisitor.visit( firstMd, md ) ; }

    virtual void visit( PmtCompositeMetadata& md )
      { userVisitor.visit( firstMd, md ) ; }

    virtual void visit( PmtMetadataT<int8>& md )
      { userVisitor.visit( firstMd, md ) ; }

    virtual void visit( PmtMetadataT<vint8>& md )
      { userVisitor.visit( firstMd, md ) ; }

    virtual void visit( PmtMetadataT<uint8>& md )
      { userVisitor.visit( firstMd, md ) ; }

     virtual void visit( PmtMetadataT<vuint8>& md )
       { userVisitor.visit( firstMd, md ) ; }
…
     virtual void visit( PmtMetadataT<float>& md )
       { userVisitor.visit( firstMd, md ) ; }
…
  private:
    V& userVisitor ;
    M& firstMd ;
};
```

```
#include "PmtSingleMdVisitorT.h"

class AlgorithmX
{
  public:
    AlgorithmX( const NLRoleTypes& theRole ) ;
    ~AlogorithmX( void ) { delete visitor; }

    void execute( PmtMetadataPtr md ) ;
    void visit( PmtMetadataT<uint8>& md ) ;
    inline void visit( PmtMetadata& /* md */ ) { return ; }

  private:
    PmtMdVisitorImpl* visitor ;
};

AlgorithmX::AlgorithmX( )
{
  visitor = new PmtMetadataVisitorT<AlgorithmX>( *this ) ;
}

AlgorithmX::execute( PmtMetadataPtr md )
{
  md->accept( visitor ) ;
}

AlgorithmX::visit( PmtMetadataT<uint8>& md )
{
  uint8 mdVal = md.value() ;
  // do something with it
}
```

How does a ***visit()*** method get called without the algorithm needing to know the specific instance of a metadata object? This is addressed by requiring each metadata specialization to have an ***accept(PmtMdVisitorImpl* theVisitor)*** method. Within the ***accept()*** method, the metadata calls the visitor's ***visit()*** method with itself as an argument, i.e., ***theVisitor->visit(*this)***, at this point the exact type of the metadata is known. Further detail is graphically shown in the sequence diagram shown below. The request for AlogorithmX to run is initiated by calling its ***execute()*** method with the argument being the metadata upon wich to operate. The ***execute()*** method in turn calls the ***accept()*** method on the ***PmtMetadata***'s interface with the argument being ***AlogithmX*** itself adapted to an ***PmtMdVisitorImpl*** through the ***PmtSingleMetadataVisitorT*** class. The ***PmtMetadata*** instance in turn calls the ***PmtSingleMetadataVisitorT***'s visit method with the argument being a reference to itself (again, at this point the actual type of the metadata instance is known since the call is being made from within the metadata instance). The call by ***PmtSingleMetadataVisitorT'***s ***visit()*** method to ***AlgorithmX'***s ***visit()*** method will resolve to ***visit(PmtMetadataT<uint8>&)*** if the metadata instance is of the type ***PmtMetadataT<uint8>*** otherwise it will resolve to the default ***visit(PmtMetadata&)***.

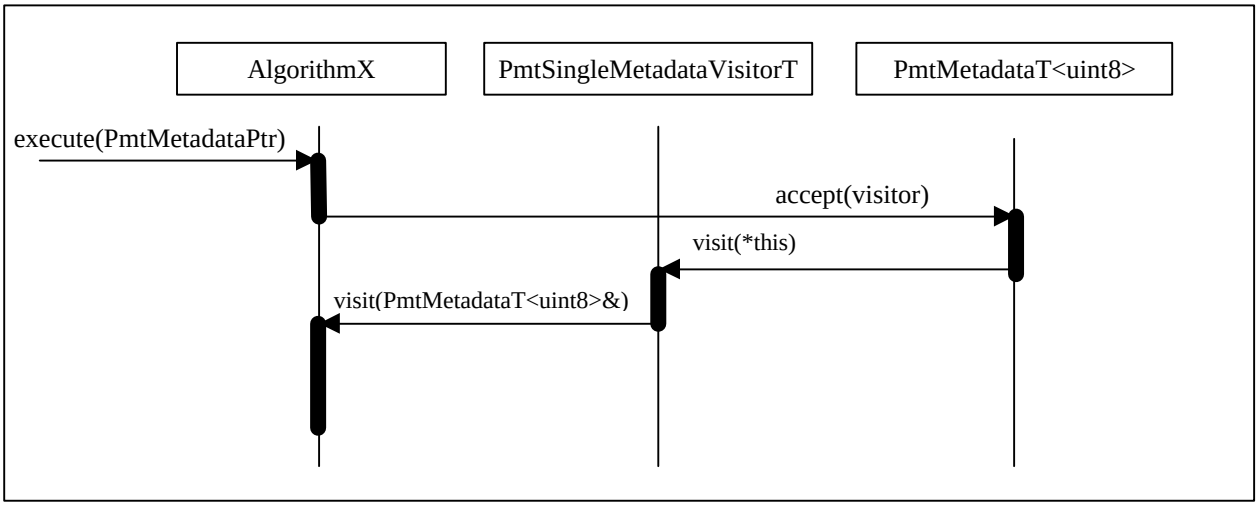

Figure 1. Sequence diagram illustrating how a PmtMetadata's type is resolved with the Visitor Pattern.

Appendix C. Creating a New Default Schema

Creating a new default schema consists of the following steps, which must be performed in Linux:

1 Create a simplified schema.

   a. The default schema cannot have imports/includes, nor can it have facets or types not listed in the **PmtTypeTable.pdf** document.

2 Place the new schema in **DefaultDefinitions/PmtDefaultDefinitions.xsd**.

3 Change the working directory to **scripts.**

4 Execute the script "**BuildNewSchema.sh**".

5 Change the working directory to the top of the PMT source tree.

6 Configure and build PMT

   a. The new default pre-parsed schema located in the **src/PmtInterpreter/** directory and called **PmtDefaultPreparsedSchema.h** may contain text line breaks that need to be repaired.

7 The new default schema is now ready for use.

Appendix D. References

[1] Musser, D.R., Saini, A, "STL Tutorial and Reference Guide," Addison-Wesley, Reading, MA (1996).

[2] TIFF v6.0, http://partners.adobe.com/PDFS/TN/Tiff6.pdf

[3] eXtensible Markup Language, http://www.w3.org/XML/

[4] Walsh, N., "A Technical Introduction to XML," http://www.xml.com/pub/98/10/guide0.html

[5] "The Annotated XML 1.0 Specification," http://www.xml.com/pub/axml/axmlintro.html

[6] *XML Schema Primer, Structures and Datatypes Recommendation*, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[7] Gamma, E., et al., "Design Patterns – Elements of Reusable Object-Oriented Software," Addison-Wesley, Reading, MA, pp. 163 (1995).

[8] Gautier, P., "Visitors Revisited," C++ Report, September 1996, pp. 37 – 45.

[9] "Japan Electronic Industry Development Association Standard for Digital Still Camera Image File Format Standard (Exchangeable image file format for Digital Still Cameras: Exif)," June 1998, V2.1, http://www.jeida.or.jp/guide/book/index-e.html

[10] Exif Specifications: http://www.exif.org/