ZeissImageLib

Design Specification

Release Version 1.1
for
ZEN 2 (blue edition)

Table of Contents

1. Introduction

When ZEISS started to develop the new ZEN software it gradually turned out that there is no suitable file format on the market to cover all the image data handling aspects needed. In addition to that ZEISS has a wide variety of microscopes and acquisition methods and most customers find it convenient to work with just one image data format on all systems. These were the main reasons why ZEISS decided to develop the CZI image format.

With the ZeissImageLib we want to enable registered users, third party software developers etc. to easily read the CZI file format. ZEISS delivers this API for .Net based software development on Microsoft Windows Systems as ZEN itself is developed under .Net and the API is derived from ZEN. ZEN software is developed under the current version of the .Net Framework and the same is true for the ZeissImageLib.

This design specification of the ZeissImageLib is a starter to work with the API and we intend to supply some basic ideas how to work with the CZI format. To get a deeper understanding of the CZI file format itself please refer to the DS_ZISRAW-FileFormat description.

2. Use cases

2.1 Open a file and display items (frames)

Most user will not have to deal with details of multi-dimensional data storage. They simply open an image and assign it to one of the GUI components.

```
ImageDocument imageDocument = new ImageDocument();
imageDocument.Open("sample.tif");
imageDisplay2d.Data = imageDocument;
```

2.1.1 Displaying a single slice of a Z-stack

An *ImageAccessor* enables access to subsets of data store (logical views).

```
ImageDocument imageDocument = new ImageDocument();
imageDocument.Open("sample.tif");
SubsetBounds bounds = new SubsetBounds{StartZ = 10};
imageDisplay2d.Data = imageDocument.CreateAccessor(bounds);
```

To use the standard WPF Image control, the image source must be converted to a WPF *ImageSource* (BitmapSource).

```
image.Source = imageDocument.CreateAccessor(bounds).CreateBitmapSource();
```

2.1.2 Creating an 'empty' image

The following code creates an empty image with Gray8 Pixeltype, an X/Y size of 100 * 200 and 20 Z slices. Memory allocation is deferred until the data is accessed. After allocation, the memory content is still undefined. To ensure defined data values use one of the **Fill** methods of **ImageAccessor**.

```
ImageDocument imageDocument = new ImageDocument(
                 PixelType.Gray8,
                 new SubsetBounds{SizeX=100, SizeY=200, SizeZ=20});

imageDocument.CreateAccessor().Fill(0);  // ensure zero content
```

2.1.3 Reading and writing pixels

The Methods **CopyPixelsToArray**, **CopyPixelsToBuffer**, **CopyPixelsFromArray** and **CopyPixelsFromBuffer** are provided for the most common scenarios.

Note that the term 'Source' is used to identify the pixels, not the data source of the operation. This may be confusing but ensures that From/To methods have a symmetric signature.

Pixel - Copy functions include:

- Region copy using a Source (pixels) and Target (client allocated memory) rectangle. If source and target sizes are different, a nearest neighbour zoom ensures that the destination area is filled. Pixel conversions – if applied - use predefined rules.
- Optional clipping rectangles for both source and destination
- Optional grey value translation using a **DataMapping** instance (High/low/gamma or grey translation table)

Read a rectangle of pixels into a user supplied array (in this sample, the array is composed of pixels of the proper type, anyway, you must pass the required pixel type).

```
int width = 100;
int height = 200;
Bgr24Pixel[] pixels = new Bgr24Pixel[width * height];
```

```
ImageAccessor accessor = imageDocument.CreateAccessor(new SubsetBounds(10, 10,
width, height);

// universal method with extended semantics
accessor.CopyPixelsToArray(
        SubsetBounds bounds,                    // optiional Subset, source ROI
         Int32Rect.Empty,                        // optional source clip rectangle
         pixels,                                 // target array
         0,                                      // offset of first pixel
         width * Bgr24Pixel.Size,               // stride
         PixelType.Bgr24,                        // array pixel type
         new Int32Rect(0,0,width,height));      // target rectangle
         Int32Rect.Empty,                        // optional target clip rectangle
         null,                                   // optional data mapping
         BoundsOrigin.Relative                   // 'bounds' interpretation

// for this  frequent case: copy the whole content without zooming and clipping
another override is available

accessor.CopyPixelsToArray(pixels,PixelType.Bgr24);
```

Copy methods are 2D operations in X/Y and will access the Center 2D plane of the dimensions Z,T,C etc if they are not contained in the specified bounds (the sample uses the constructor overload setting StartX/Y and SizeX/Y only).

To use another plane as the center plane in one of the dimensions, set the dimension start index explicitely.
E.g. to select the slice Z with index 10 of a Z-Stack.

```
ImageAccessor accessor = imageDocument.CreateAccessor(
   new SubsetBounds{StartX=10, StartY=0, SizeX=width, SizeX=100, StartZ=10});

accessor.CopyPixelsToArray(... );
```

Read a downsampled (1/10) rectangle

```
int sourceWidth = 10;
int sourceHeight = 20;
double zoom = 1./10;
int targetWidth = (int)sourceWidth * zoom);
int targetHeight = (int)(sourceHeight * zoom);

ImageAccessor accessor = imageDocument.CreateAccessor(
                new SubsetBounds(0, 0, sourceWidth, sourceHeight));

Bgr24Pixel[] pixels = new Bgr24Pixel[width * height];
accessor.CopyPixelsToArray(
          pixels,
          0,
          width * Bgr24Pixel.Size,
          PixelType.Bgr24,
          new Int32Rect(0,0,targetWidth,targetHeight));
```

2.1.4 Create a new ImageDocument and Initialize Data

```
SubsetBounds bounds = new SubsetBounds();
bounds.SizeX = 512;
bounds.SizeY = 512;
bounds.SizeZ = 100;
ImageDocument imageDocument = new ImageDocument(PixelType.Gray8, bounds);
```

```
SubsetBounds sliceBounds = new SubsetBounds();
sliceBounds.SizeX = 512;
sliceBounds.SizeY = 512;
for (int z = bounds.StartZ; z < bounds.SizeZ; z++)
{
     itemBounds.StartZ = z;
     ImageAccessor accessor = imageDocument.CreateImageAccessor(sliceBounds);
     ImageAccessor.Fill(z);  // sample: z=gray value
}
```

3. General Types and Enumerations

All the following types are defined in ***Zeiss.Micro.Imaging*** namespace.

3.1 PixelType

The following table shows the currently defined pixel types and the related WPF pixel formats:

Identifier	Comments	PixelFormat (WPF)	BPP
Gray8	8 bits-per-pixel grayscale channel, allowing 256 shades of gray.	Gray8	8
Bgr24	A sRGB format with 24 bits per pixel (BPP). Each color channel (blue, green, and red) is allocated 8 bits per pixel (BPP). Some file formats also may contain or handle the reverse order, i.e. RGB but we should restrict to one format.	Bgr24	24
Bgra32	A sRGB format with 32 bits per pixel (BPP). Each channel (blue, green, red, and alpha) is allocated 8 bits per pixel (BPP). This is the recommended standard format for visualization in the WPF environment.	Bgra32	32
Gray16	**Word** is 16 bits-per-pixel grayscale channel, allowing 65536 shades of gray. This format has a gamma of 1.0 16 bit unsigned. Not supported by BMP and JPG.	Gray16	16
Bgr48	A sRGB format with 48 bits per pixel (BPP). Each color channel (red, green, and blue) is allocated 16 bits per pixel (BPP). This format has a gamma of 1.0. 16 bit triples representing the color channels Blue, Green and Red. This format is provided by most acquisition devices with 12, 14 and 16 bit color resolution. Not supported by BMP and JPG.	Rgb48 (B/R swapped!)	48
Gray32Float	A 32 bits per pixel (BPP) grayscale channel, allowing over 4 billion shades of gray. This format has a gamma of 1.0. This special format not supported by standard picture formats. May be saved as specially tagged "Byte" images in formats with lossless compression. Float images are normalized to value range from 0…1 (or 0..65535 as recently discussed).	Gray32Float	32
Gray64ComplexFloat	Real / imaginary combination (2 floats)	-	64
Bgr192ComplexFloat	BGR triples of Real / Imaginary combinations	-	192

3.2 ImageDimension

The overall concept of the ZIS image is a multi-dimensional hyperspace. The number of dimensions in this hyperspace has no implicit limitation. The current set of dimensions is defined using a **ImageDimension** enumeration. This enumeration may be extended in the future.

Name	Description
X	The extent in X direction (width).
Y	The extent in Y direction (height.
Z	Slice in Z direction.
C	Channel in a Multi-Channel data set.
T	Time point in a sequentially acquired series of data.
S	Scene – for clustering items in X/Y direction (data belonging to contiguous regions of interests in a mosaic image) .
R	Rotation – used in acquisition modes where the data is recorded from various angles.
I	Illumination – represents different light sources (used e.g. im SPIM).
B	Block – used to add multiple sections into a single image container
M	Mosaic (not a real index). Use to define the order of tile within a mosaic
V	View – used for MultiView images (e.g. SPIM acquisition)

3.3 Indices and SubsetBounds

Navigation in the hyperspace uses the term **Index** to adress a logical position within the given dimension. While X and Y are something special as they refer to "real" Pixel positions and extents, the other Indices have logical meaning only, e,g. StartZ=5 means: the 6th slice (zero based) in the hyperspace but has no correspondance to a physical (e.g. focus) position.

As the number of dimensions will increase in future versions, all APIs dealing with subset definitions use the **SubsetBounds** class which encapulates the multi-dimensional subspace definition.

For convience, the most frequently used ***Indices*** in **SubsetBounds** are exposed as CLR properties.

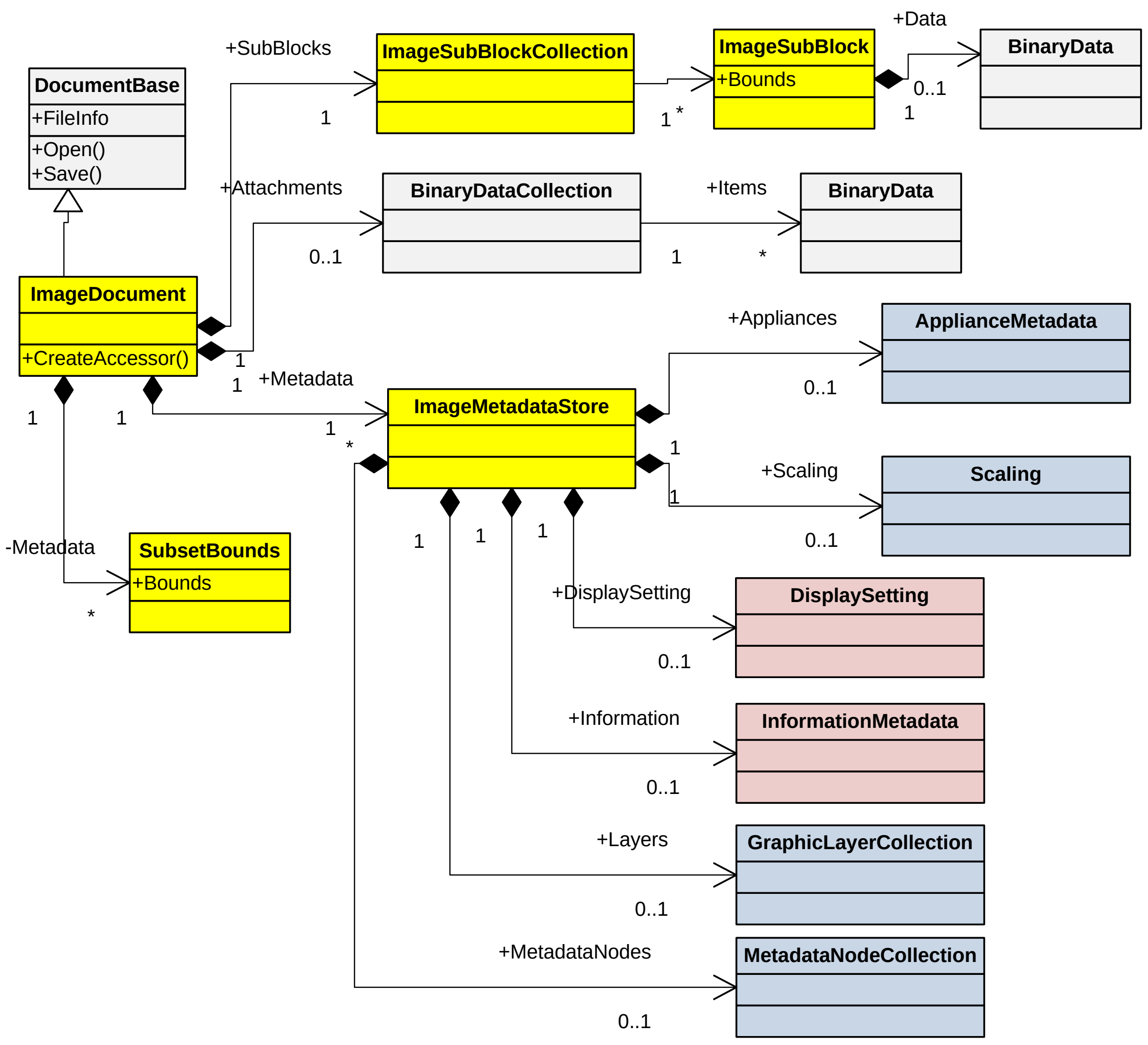
Property	Data type	Description
StartX, StartY	integer	Offsets of the left, upper pixel in the X/Y space. Mosaic tiles use those value to define their origin.
SizeX, SizeY	integer	Number of pixels in X, Y direction.
StartC	integer	Channel start index (1 st. channel contained in the subspace)
SizeC	integer	Number of channels.
StartT	integer	Time start index.
SizeT	integer	Number of time stamps.
StartZ	integer	Z-Stack start index (1 st slice contained in the subspace)
SizeZ	integer	Number of slices.
StartS	integer	Start scene index.
SizeS	integer	Number of scenes.
StartR	integer	Start rotation index.
SizeR	integer	Number of rotation angles.

StartV	integer	Start view index.
SizeV	integer	Number of views.
StartI	integer	Start ilumination index.
SizeI	Integer	Number of illumination indices.
StartB	Integer	Start block index
SizeB	Integer	Number of blocks
StartV	Integer	Start view index
SizeV	Integer	Number of views

4. The ImageDocument

4.1 Image Document model

An **ImageDocument** consists of Pixel Data and Metadata read from and witten to disks and streams (**ImageDocument**, **ImageSubBlock**, **Metadata** and **Attachments**).



4.2 Open an existing image

To get access to an existing image document on disk use the **Open** method. This will keep the document open (and the file locked) for incremental access.

```
public void Open(string fileName)

ImageDocument imageDocument = new ImageDocument();
imageDocument.Open("d:\test.jpg");
```

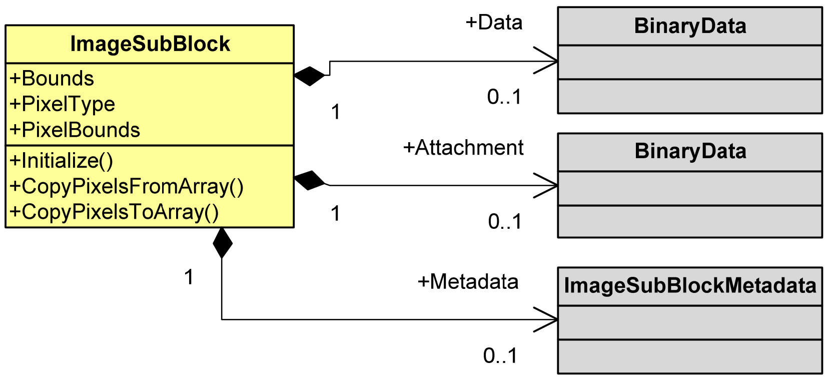
After opening or creating the image, the full storage infrastructure is created and initialized – this includes initialization of all ***SubBlocks***. The image memory allocation is deferred until the pixel data must be accessed.

In the case of an incremental access to an existing file, the memory belonging to e.g. Z=23 is allocated and filled with the pixel data only when this slice is actually accessed – by using the Z-player or something else.

4.3 The ImageSubBlock(Collection)

4.3.1 ImageSubBlock

An **ImageSubBlock** is the container for the raw pixel data and closely related metadata. By definition, this container has unlimited dimensionality but will typically contain only the 5 dimensions X,Y,C,Z,T. The contained dimensions can be obtained by examining the **Bounds** member (*Contains* method).



An **ImageSubBlock** represents a collection of 2D data items with **equal size and pixel type** and has an optiopnal attachment containing auxiliary data (e.g. data used to interpret specfic data types**).**

Pixels (Data) and Attachment use the class **BinaryData** as the holder of the raw data.

The subspace covered by an **ImageSubBlock** is filled (no gaps / holes). A multi-dimensional acquisition process will choose to either pack all data in a single *SubBlock* (e.g. fast timelapse acquisition) if the data is a homogenous stream of data with equal characteristics (a long time series of 512x512 RGB images) or create multiple *SubBlocks* for homogenous but also different data (e.g. time lapse with 5 gray channels and a single color channel).

More details – see chapter Raw pixel data storage (ImageSubBlock).

4.3.2 ImageSubBlockCollection

The collection holds the references to the individual **ImageSubBlocks**. When opening an image document, each of the contained *SubBlocks* contains a reference to its location and extent in the hyperspace (Bounds).

The method **CreateSubset** may be used to get a new **ImageSubBlockCollection** containing all the SubBlocks that intersect a specfied **Bounds** (optional: matches a given predicate).

4.4 ImageCodec

A component referenced from both the image container and the **ImageSubBlocks**. Its responsibility is to convert the raw pixel data streams and metadata into runtime data represented by Pixels, the (Overlay)Layers and Tags.

5. ImageAccessor

5.1 General information

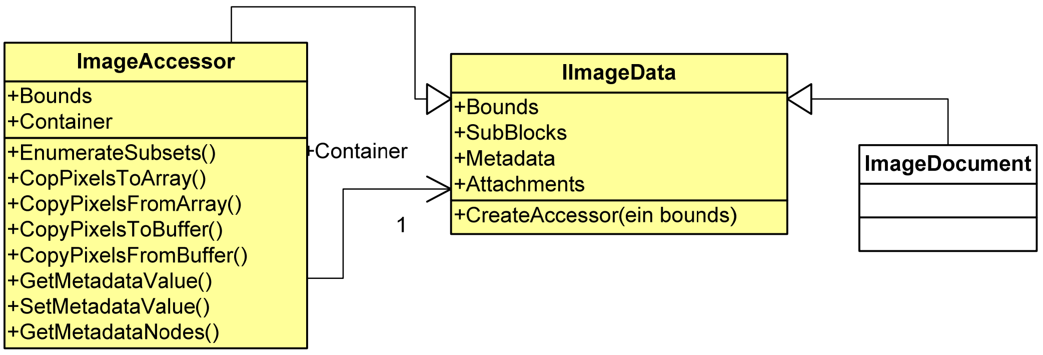
An **ImageDocument** may contain huge amount of heterogenous pixel data and metadata.

SubBlocks may have varying dimensionality and pixel type. In addition, data may be stored in sub / supersampled resolutions given by the ration *Bounds* (logical sizes) / *PixelBounds* (stored data extent).

Each single **SubBlock** may represent an individual 2D tile (or xD - cube) of an image area. In many image processing scenarios this underlying tile – structure should not be visible, i.e. an algorithm wants to read a rectangle at any position with any size – totally ignoring the physical tile situation.

This may require various steps: first get the list of ***SubBlocks*** intersecting the rectangle and then reading the individual regions into a common data buffer. More complicated scenarios will involve automatic selection of the best matching zoom stage in multi-resolution images and automatic data conversion.

Even if this could be achived using the *ImageSubBlock's* raw *CopyPixels* methods and navigating the Metadata strcuture, an **ImageAccessor** is the way for image data access.



ImageAccessor serves two main tasks

- Encapsulate a ***SubsetBounds***.
- Provide easy – cross - *SubBlock* access with automatic resolution selection when working with multi-resolution images.

IMPORTANT:

ImageAccessor treats multiple tiles (SubBlock items) belonging to the same dimensions as a single 2D area, i.e. underlying tiling is an implementation detail only when using the **CopyPixel** methods of **ImageAccessor**.

5.1.1 Typical properties of a ImageAccessor

Property	Data type	Description
Bounds	SubsetBounds	The actual bounds (subspace) accessed by this **ImageAccessor**. When creating a **ImageAccessor**, the given *Bounds* are copied to an internal member (cloned).
Container	object	The container holding the data for this **ImageAccessor**. The data store used by the **ImageAccessor** is not necessarily an **ImageDocument**. Instead, **ImageAcccessor** uses an abstraction given by the **IImageContainer** interface. *ImageDocument* implements this interface. Be aware that **SubBlocks** are shared with all descendants.

5.1.2 Typical methods of a ImageAccessor

Method	Description
Split(*SubsetBounds* bounds, *ImageDimensions[] dimensions*)	Creates a collection of ImageAccessors to all subsets within the specified dimensions.
Split2d() **Split2d**(*SubsetBounds* bounds)	Splits into 2D items. For tiled images, a 2D area is the bounding box around all tiles..
Split3d (*SubsetBounds* bounds)	Splits into 3D subsets.

CopyPixelsFromBuffer(…)
CopyPixelFromArray(…)
	Copies pixels contained in an array or buffer (IntPtr) to a rectangle within the subspace referenced by this instance. Includes zoom and data conversion

CopyPixelsToBuffer(…), **CopyPixelsToArray** (…)	Copies pixels of a rectangle (2D) within the subspace referenced by this instance to a rectangle within a user defined array or buffer. Includes zoom and data conversion.

5.2 Creating an accessor

ImageAccessors are normally created by their container using a **CreateAccessor** method. This method returns a 'Default' **ImageAccessor** to cover the most frequently operations.

Additional accessors may be provided in the future which accept an **IImageContainer** interface to work on.

```
ImageAccessor accessor = imageDocument.CreateAccessor();  // all data
ImageAccessor accessor = imageDocument.CreateAccessor(bounds);
```

5.3 SubsetBounds and effective Bounds

The **Bounds** of an **ImageAccessor** uniquely identify a subset within the multi-dimensional container. This also means that all dimensions must be present. It is the job of the container class to ensure this and this is typically done by a Union / Intersection operation before instantiating the **ImageAccessor** in the **CreateAccessor**() method.

e. g. suppose the following code

```
ImageDocument imageDocument = new ImageDocument(
           PixelType.Gray8,
           new SubsetBounds()
             { SizeX = 512, SizeY = 512, SizeC = 3, SizeZ = 40, SizeT = 4 });

ImageAccessor accessor = imageDocument.CreateAccessor(new SubsetBounds{StartZ = 10});
```

If you examine the **Bounds** property of the resulting **ImageAccessor**, it has the following 'effective' **Bounds**

```
SizeX = 512,  SizeY = 512,  SizeC = 3,  StartZ = 10, SizeT = 4
```

In general: All Size values are implicitly 1 unless otherwise specified. All Start values are implicitly 0.

Calling

```
accessora.Split2d(new SubsetBounds(10,11,12,13));
```

will result in **ImageAccessors** with the following bounds (here: total of SizeC * SizeT = 3 * 4 = 12 items), shown in reduced format for brevity.

```
StartX = 10, SizeX = 12,
StartY = 12, SizeY = 13,
StartC = [0..2], (SizeC = 1),
StartZ = 10, (SizeZ = 1)
StartT = [0..3], (SizeT = 1)
```

5.4 More on enumerated subsets

5.4.1 ImageAccessor.Split() methods

In most of the image processing scenarios, you have an algorithm that works on a defined dimensionality. To make things easy, **ImageAccessor** provides some **Split** methods that return a **ImageAccessor** instances matching certain scenarios and constraints.

```
public ImageAccessorCollection Split(SubsetBounds bounds)
```

Sample 1: Get 2-dimensional ***ImageAccessors*** for 10 slices starting from Z=10.

```
ImageDocument imageDocument = new ImageDocument(PixelType.Gray8, 10, 10, 50);
SubsetBounds bounds = new SubsetBounds {StartZ = 10, SizeZ = 10};
ImageAccessorCollection slices = imageDocument.CreateAccessor(bounds).Split2d();
```

Sample 2: Get all 2-dimensional ***ImageAccessors*** contained in the document ("flatten to 2d").

```
ImageAccessorCollection slices = imageDocument.CreateAccessor().Split2d();
```

Sample 3: Image processing sample (call 2D processing function for each item individually):

```
// open multi-dimensional image, add different constant to each item

ImageDocument input = new ImageDocument();
input.Open(@"c:\sampleZ.czi");
ImageDocument output = new ImageDocument();
output.Initialize(input, ImageInitializeModes.AllButNoContent);

int n = 3;
foreach (ImageAccessor accIn in input.CreateAccessor(null).Split2d())
{
    ImageAccessor accOut = output.CreateAccessor(accIn.Bounds);
    new AddConstant{ Input = accIn, Output = accOut, Operand = n}.Execute();
    n += 5;
}

output.Save(@"c:\sampleOut.czi", false);
output.Close();
input.Close();
```

5.4.2 CreateImageAccessors3d – create 3d stacks in any dimension

Assume a XYCZT data volume.

```
ImageDocument imageDocument = new ImageDocument(

PixelType.Gray8, new SubsetBounds(100, 100){SizeZ = 10, SizeT = 10, SizeC = 5}
);
```

You may get a series of 50 (10 * 5) Z Stacks using

```
ImageAccessorCollection acc3dZ =
     imageDocument.CreateAccessor().Split3d(ImageDimension.Z);
```

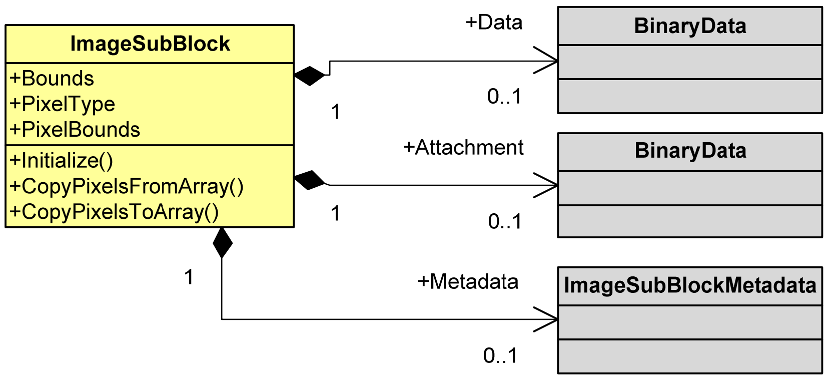
or a collection of 100 (10 * 10) C – Stacks using

```
ImageAccessorCollection acc3dC =
    imageDocument.CreateAccessor().Split3d(ImageDimension.C);
```

6. Raw pixel data storage (ImageSubBlock)

6.1 Overview

A ZIS image stores is pixels data and related metadata in **ImageSubBlocks**.



6.2 ImageSubBlock - defined

The pixel data referenced by an **ImageSubBlock** is stored in a separate memory unit.

6.2.1 Typical properties of an ImageSubBlock

Property	Data type	Description
PixelType	PixelType	The pixel type. An **ImageSubBlock** may have multiple dimensions (>2) but all data has the same pixel type..
Bounds	SubsetBounds	The actual bounds (subspace) of this **ImageSubBlock** in the hyperspace. When creating an **ImageSubBlock**, all involved ***Sizes*** of this block must be provided and this information cannot be changed after initialisation. The ***Start*** locations instead can be modified as long as the SubBlock has not been addded to the Document's SubBlock collection. To ensure this behavior, the **Initialize** method accepts a **SubsetBounds** argument and copies it to an internal member. the Sizes of the Bounds are then locked and only the Start location can be modified. When adding a SubBlock to a Collection, the Start locations are locked, too.
PixelBounds	SubsetBounds	Stored sizes in units of pixels. The ratio of a dimension's Size in logical (Bounds) and physical (PixelBounds) representation defines the implicit zoom / scale factor when reading and writing pixel data. The Start information of PixelBounds is not used. As in most cases, the subsample factor for X and Y is the same, a convenience property **Scale** is available to represent the scaling factor, e.g. Bounds.SizeX = 100 and PixelBounds.SizeX = 10 has a **Scale** of 0.1;

6.2.2 Typical methods of an ImageSubBlock

Method	Description
Initialize(*PixelType* pixelType, *SubsetBounds* bounds, *SubsetBounds* pixelBounds)	Initializes the **ImageSubBlock** but does not yet allocate memory for the pixel data. The passed-in **SubsetBounds** object is cloned internally, so you may reuse I to initialize other objects.

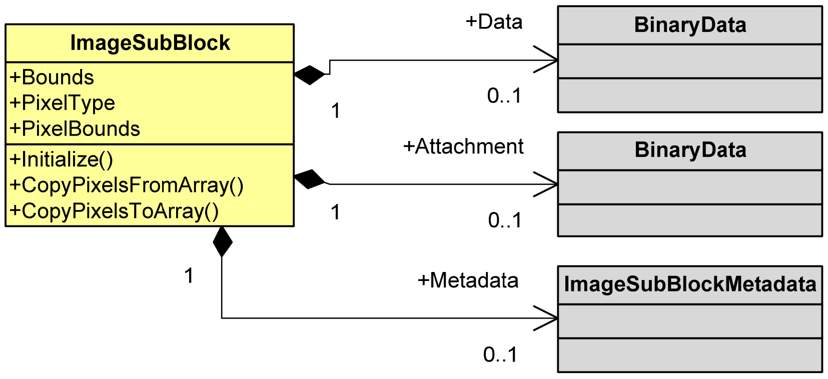
CopyPixelsFrom..(…)	Copies pixels from the specified array or buffer into the memory of this instance – with optional zoom and data conversion. Multiple overloads will be provided as required.
CopyPixelsTo..(…)	Copies pixels from the pixel memory to the specified array or buffer – with optional zoom and data conversion. Multiple overloads will be provided as required.

6.3 Image Memory

Runtime storage of the pixels uses a combination of managed / conventional (virtual) memory and memory mapped files (MMF). Each **ImageSubBlock** has an attached "***BinaryData***" instance for data storage. The algorithms used to determine the best storage schema may differ for 32 or 64 bit environments.

Image memory is provided by either

- Managed memory (new byte[])
- Unmanaged memory, heap based (Virtual Alloc)
- Shared memory (Mapped file API via Win32 Interop)



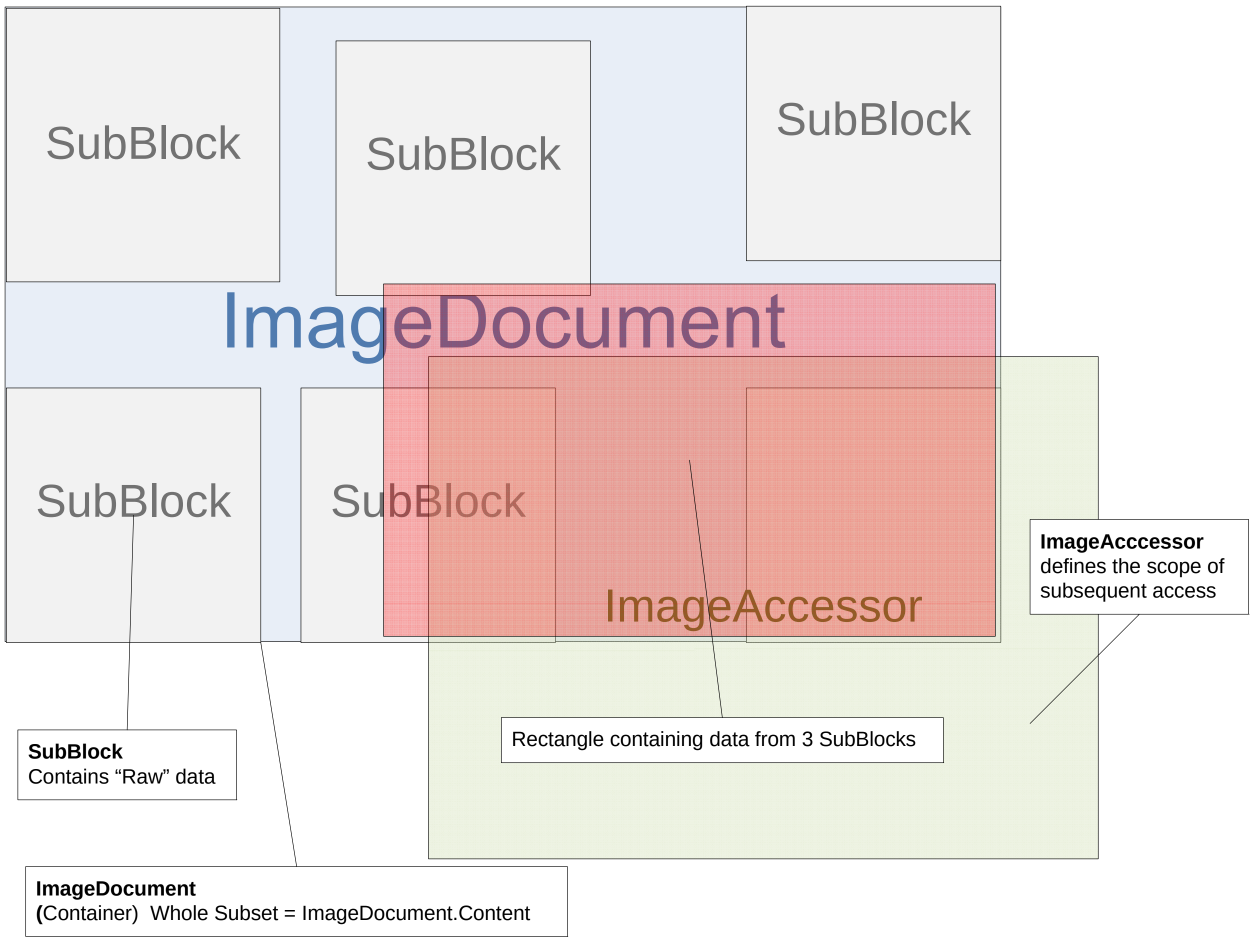
The default storage location is global unmanaged memory for best performance and easy disposal - if required. The algorithm where memory is allocated may be dynamically adjusted to optimize the application's needs. E.g. is there is enough memory and images are small, there is no need to create memory mapped files, so the unmanaged memory will be the optimal implementation.

Access to the memory is NOT done via pointers, but only using the methods **CopyPixelsTo… / CopyPixelsFrom…** The implementation of the various conversion and zoom methods is fully contained in those low levels objects to save additional memory copy operations in the upper levels.

6.4 General usage of SubsetBounds

To understand the **Bounds** parameter passed to various methods like **CreateImageAccessor** or **CopyPixelsToArray**, it is important to understand the underlying concept.

6.4.1 Overview diagram


SubBlock
SubBlock
SubBlock
ImageDocument
SubBlock
SubBlock
ImageAccessor
ImageAcccessor defines the scope of subsequent access
Rectangle containing data from 3 SubBlocks
SubBlock Contains "Raw" data
ImageDocument (Container) Whole Subset = ImageDocument.Content

6.4.2 ImageSubBlock / ImageDocument / ImageAccessor

An **ImageSubBlock** carries the "raw" data obtained from the acquisition source or data repository, the **Bounds** defines a virtual position and logical size anywhere in hyperspace. It is not necessary to align the **SubBlocks** within a container to zero.

The Container (**ImageDocument**) holds a collection of **SubBlocks** and provides a **Bounds** information which is or may be calculated as the bounding box for all of its contained **SubBlocks**. The ImageDocument's *Bounds* is linked to the *Bounds* of an attached **ImageAccessor** (**Content** property) that represents the whole data set.

A **ImageAccessor** is a data view to the underlying *ImageSubBlocks*. A *ImageAccessor's* **Bounds** use the same (absolute) indices as the *ImageSubBlocks*. It is not necessary to match the bounding area covered by the *SubBlocks* containing the pixel data, but, accessing areas with no intersecting SubBlocks data yields zero values when reading or results in data loss when writing.

6.4.3 Absolute and Relative Bounds → "Effective" Bounds

Some methods using a **bounds** parameter also require a parameter **origin** of type **BoundsOrigin**.

The following table shows the usage of the parameter **bounds** and **origin** in conjunction with the (local) Bounds of a *ImageAccessor* to calculate the (effective) Bounds for further processing,

(SubsetBounds) bounds	(BoundsOrigin) origin	Effective Bounds
Null	Absolute	localBounds
!= null	Absolute	effectiveBounds = bounds.Clone() effectiveBounds.**Union**(localBounds)

		effectiveBounds.**Intersect**(bounds)
null	Relative	localBounds
!= null	Relative	effectiveBounds = bounds.Clone() effectiveBounds.**Offset**(localBounds) effectiveBounds.**Union**(localBounds) effectiveBounds.**Intersect**(bounds)

Offset translates the specified **bounds** using the Start positions of its argument.

Union ensures that *effectiveBounds* contains all dimensions of both local and specified **bounds** and includes the index range of the specified bounds.

Intersect clips the intermediate *effectiveBounds* to not extend beyond the index range specified in the **bounds** parameter.

By default: No Clipping!

If clipping is required, an additional **sourceClipRect** parameter is available, normally this parameter will be set to Int32Rect.Empty. **SourceClipRect** must be provided in either absolute or relative units, depending on the setting of **origin**.

Absolute mode is useful to define a common bounding area for multiple items, e.g. you may define a bounding area of StartX=10, StartY=10, SizeX=100, SizeY = 100 and use the same are area for accessing a list of multiple ImageAccessors, SubBlocks etc.

In most cases, you will use **Relative** indices to access an area relative to the specified item. In **Relative** mode, the specified **bounds** are zero based, e.g. StartX=0, StartY=0, SizeX=10, SizeX=10 will copy the left / upper 10 * 10 pixel area.

6.5 Copy Pixels

The **CopyPixels** methods are used in various contexts but always have the same signature and arguments, so a common description will be given in this chapter.

CopyPixels copies data from one 2D rectangle to another. One data location are the pixels ("Source") and the other is a user defined buffer.

For implementing the most common operations, the CopyPixelsXX methods have the following features

- **Clipping and zooming**. The copy functions accept a (pixel-) source and (user data-) destination rectangle. Both rectangles can be located anywhere in "space". Proper clipping will be applied to affect only the pixels within valid data areas. Areas outside the clipping regions are simple ignored – no data is copied and no pixel values are modified. **Nearest neighbour zoom** will be used if Source and Destination rectangles have different extents.
- **Pixel Type conversions** by accepting the requested (CopyTo) or current (CopyFrom) *PixelType* of the user's data buffer. If this type is different from the accessed SubBlock's data type, data is converted.
- **Pixel value Table translation** using a DataMapping object, Mapping may be performed using the Low / High / Gamma information or a supplied transformation table.

Signature of the CopyPixel functions

```
public bool CopyPixelsToArray(
	 SubsetBounds bounds,
	Int32Rect sourceClipRect,
	Array pixels, int stride, int offset,
	PixelType pixelType, Int32Rect rect, Int32Rect clipRect,
	DataMapping dataMapping, BoundsOrigin origin)

public bool CopyPixelsFromArray(
	SubsetBounds bounds,
	Int32Rect sourceClipRect,
	Array pixels, int stride, int offset,
	PixelType pixelType, Int32Rect rect, Int32Rect clipRect, BoundsOrigin origin)
```

Both **bounds** and **sourceClipRect** are related to the **pixel source** not the source of the operation. You may use the same signature and arguments for both copy directions. See previous chapter General usage of SubsetBounds for more details on this parameters.

A second set of **Copy** methods is available accepting a buffer instead of an array.

```
public bool CopyPixelsToBuffer(
      SubsetBounds bounds,
      Int32Rect sourceClipRect,
      IntPtr buffer, int bufferSize, int stride,
      PixelType pixelType, Int32Rect rect, Int32Rect clipRect,
      DataMapping dataMapping, BoundsOrigin origin)

public bool CopyPixelsFromBuffer(
       SubsetBounds bounds,
      Int32Rect sourceClipRect,
      IntPtr buffer, int bufferSize, int stride,
      PixelType pixelType, Int32Rect rect, Int32Rect clipRect, BoundsOrigin origin)
```

The return value of all those functions is **True** if any pixels are affected by this operation and **False** if not (i.e. the given bounds is totally outside the Bounds of the pixel source.

The parameter **DataMapping** (only available in the CopyTo methods) may be used to apply a table translation of the source pixels. This parameter is useful if the result should be displayed or converted to a bitmap.

Pixel type conversions: (V = out, v = in r,g,b = components)

	Target Type				
SourceType	Bgr24	Byte	Word	Bgr48	Float
Bgr24	-	V = (vb + vb + vr) /3	V = ((vb + vb + vr) /3) << 8	Vb = vb << 8 Vg = vg << 8 Vr = vr << 8	V = (vb + vb + vr) /3
Byte	Vb = Vg = Vr = v	-	V = v << 8	Vb = Vr = Vg = v << 8	V = v
Word	Vb = Vg = Vr = v >> 8	V = v >> 8	-	Vb = Vr = Vg = v	V = v
Bgr48	Vb = vb >> 8 Vg = vg >> 8 Vr = vr >> 8	V = ((vb + vg + vr) / 3)>> 8	V = ((vb + vg + vr) / 3)	-	V = ((vb + vg + vr) / 3)
Float	Vb = Vg = Vr = (byte)v	V = (byte)v	V = (word) v	Vb = Vg = Vr = (word)v	-

7. Persistence

7.1 General

Persistence of images is a combination of saving / loading the central data (**ImageDocument**) and the **ImageSubBlocks**. The process of persisting *ImageSubBlocks* is delegated to codecs.

When opening a file and just changing some metadata (***IsMetadataModified*** is true, but ***IsDataModfied*** is false) some formats will support a fast metadata-only update.

7.2 Standard picture files (TIFF, JPG, BMP, PNG, WMP)

If an image sub-block meets some criteria, it can be stored in a standard format file. Dimensions > 2D are stored as sequence of 2D files - or 2D frames if the format supports it.

Within this sequence we use a fixed naming schema based on StartX, StartY….:

File_XxYxZxCxTx.Extension

Standard picture files and supported pixel types

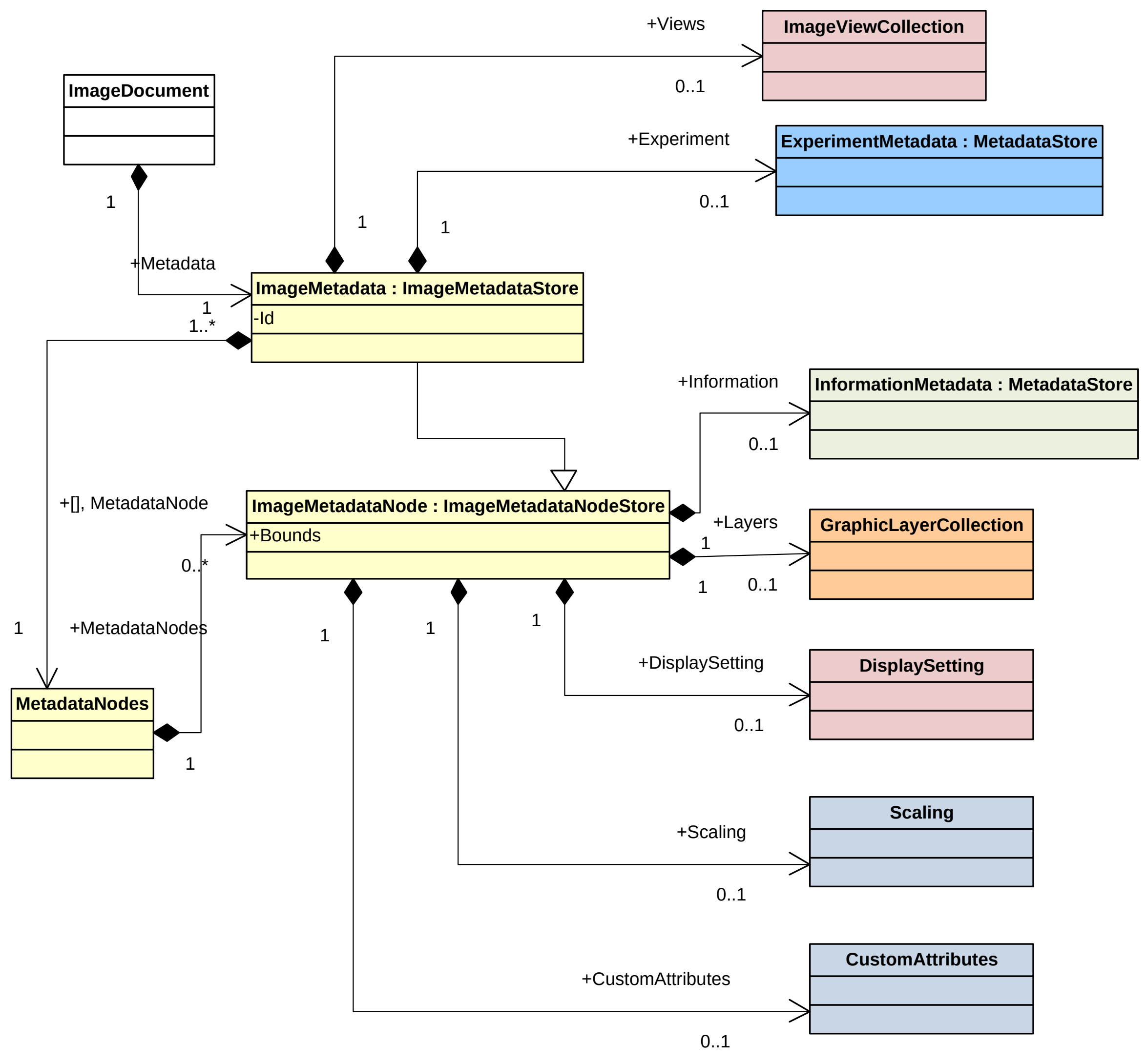
PixelType	**Format**	**Notes**
Gray8	TIFF, JPG, BMP, PNG	JPG is always compressed.
Bgr24	TIFF, JPG, BMP, PNG	
Gray16	TIFF, PNG, WMP	
Gray48	PNG	
Other	Any (Bytes, i.e. Gray8)	Raw bytes with sufficient width/height to hold the data. Can be displayed using standard tools but has no reasonable visual representation

8. Metadata

The metadata schema closely matches the XML schema defined DS_ZISRAW-FileFormat. For detailed information you may also reference this document's **Metadata** section.

8.1 Common metadata

ImageMetadata is a complex and extensible tree structure as follows



The core class in this context is **MetadataNode** which holds a number of items for specific usage areas.

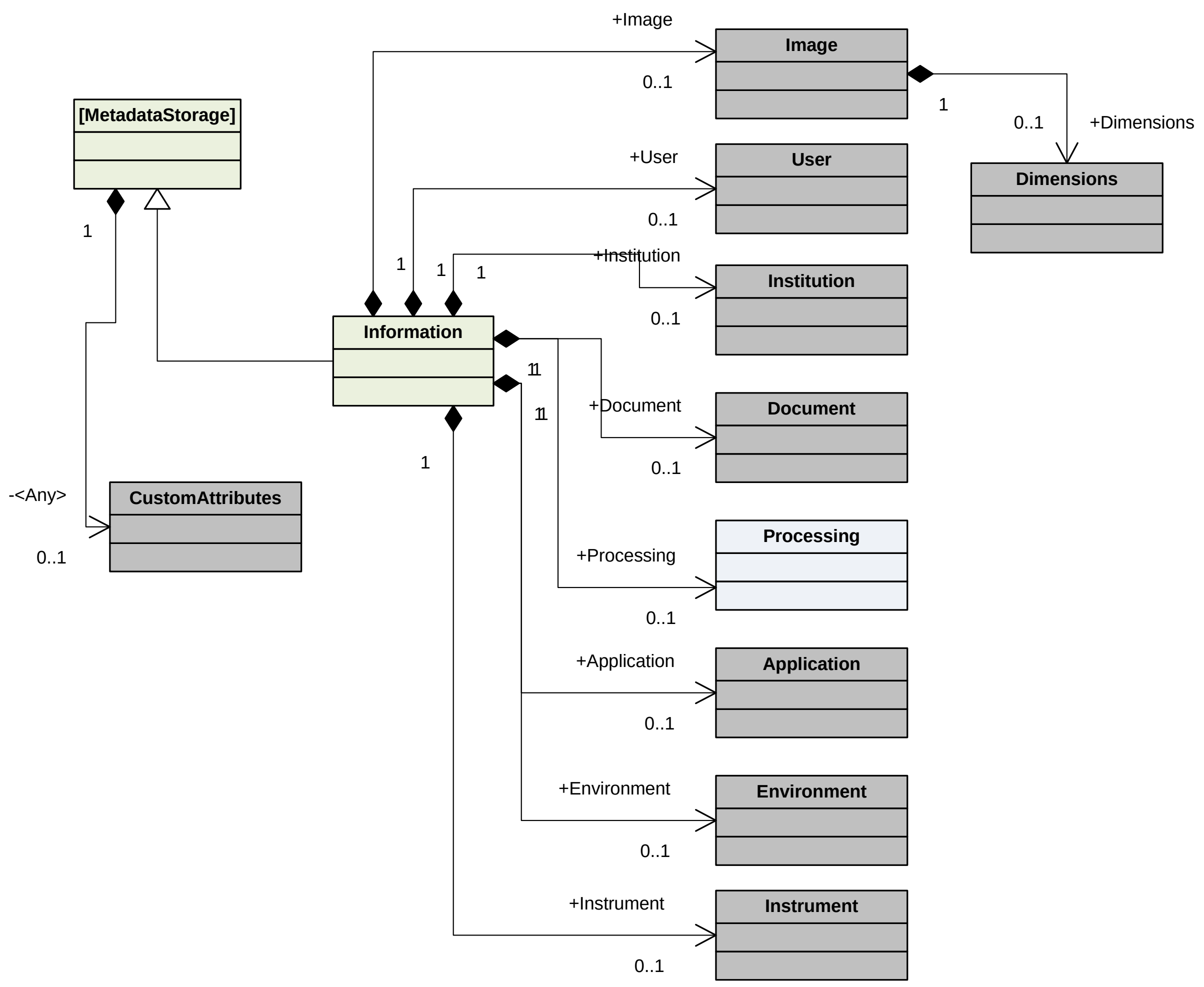
A **MetadataNode** contains the data for either the whole image or a Subset defined via a **Bounds** definition.

The root of the Metadata tree (**ImageMetadata**) derives from this node schema and adds some elements that are not intended to be specfic for a single subset.

8.1.1 Information Metadata

A special node is **Information** and contains the most common metadata to describe the image as a whole.

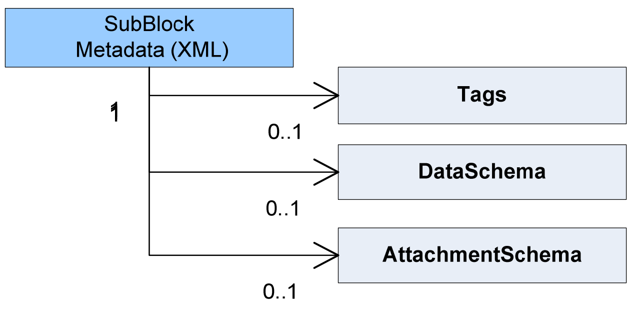
All elements with **Information** derive from the **MetadataStorage** schema (derivations of elements except the root element are not shown in the following graph) . For extensibility, the **MetadataStorage** schema uses **XStorage** elements to hold most of the data. The schema ensures that all data that is read from a source location via XML persistence is kept when saving it into another location – i.e. it also keeps XML elements that are not known for this specfic software version. This ensures that a file created with a newer software version with new elements defined does not loose data when loaded into an older version and saved afterwards.



A general extension point is **CustomAttributes** and is used to contain any kind of data – either scalar types or objects that supports XML serialisation.

8.2 SubBlock local metadata

Each ImageSubBlock may contain local metadata to hold some specific information.



www.zeiss.com/czi

Carl Zeiss Microscopy GmbH
Carl-Zeiss-Promenade 10
07745 Jena, Germany
microscopy@zeiss.com
www.zeiss.com/microscopy

Carl Zeiss Microscopy GmbH
Königsallee 9-21
37081 Göttingen
Germany