

# **Medv4D**

## **project documentation**

**Medv4D: project documentation**

# Table of Contents

<b>Preface .....</b>	<b>vi</b>
<b>1. Library Common .....</b>	<b>1</b>
<b>2. Library Imaging.....</b>	<b>2</b>
2.1. Compilation.....	2
2.1.1. Dependencies.....	2
2.2. Architecture.....	2
2.2.1. Datasets.....	3
2.2.2. Filters .....	3
2.2.3. Connections .....	3
2.2.4. Ports.....	3
2.3. Most important classes.....	4
2.3.1. Dataset hierarchy .....	4
2.3.2. Filter hierarchy .....	4
2.4. Main design concepts.....	6
2.4.1. Locking image parts .....	6
2.5. Usage.....	7
2.5.1. How to build a pipeline.....	7
2.5.2. Creating new filter .....	9
2.5.3. Defining new dataset type.....	13
<b>3. DICOM Client library.....</b>	<b>14</b>
3.1. DCMTK .....	14
3.2. Compilation.....	15
3.2.1. Dependencies.....	15
3.3. Architecture.....	15
3.3.1. DcmProvider.....	15
3.3.2. DicomObj .....	15
3.3.3. DicomAssociation .....	15
3.3.4. AbstractService.....	16
3.3.5. FindService.....	16
3.3.6. MoveService.....	16
3.3.7. StoreService.....	17
3.3.8. LocalService .....	17
3.4. Usage.....	17
<b>4. Data modifier.....</b>	<b>19</b>
4.1. Compilation.....	19
4.1.1. Dependencies.....	19
4.2. Architecture.....	19
4.3. Usage.....	20
<b>5. Remote computing (CellBE library) .....</b>	<b>21</b>
5.1. Compilation.....	21
5.1.1. Dependencies.....	21
5.2. Architecture.....	21
5.2.1. Job.....	21
5.2.2. Serializers .....	22

5.2.3. CellClient class .....	24
5.2.4. Remote Filter .....	24
5.2.5. Server .....	24
5.3. Usage .....	26
5.3.1. FilterSerializer .....	26
5.3.2. DataSetSerializer .....	29
5.3.3. Remote filter .....	32
<b>6. User Manual .....</b>	<b>35</b>
6.1. Getting Started .....	35
6.1.1. System requirements .....	35
6.1.2. Installing the application .....	35
6.2. Using the application .....	36
6.2.1. Application window .....	36
6.2.2. Using the toolbar .....	37
6.2.3. Using tools .....	38
6.3. Retrieving and Viewing Images .....	42
6.3.1. Using the Study Manager .....	42
6.3.2. Viewing studies .....	46
6.4. Manipulating Images .....	48
6.4.1. Adjusting window/level settings .....	49
6.4.2. Changing image/slice orientation .....	50
6.4.3. Adjusting image viewing options .....	50
6.4.4. Clearing measurements .....	50
6.5. Measuring Images .....	50
6.5.1. Overlaying text .....	50
6.5.2. Making measurements .....	50
6.5.3. Probing images .....	50
6.5.4. Clearing measurements .....	51

# List of Tables

6-1. Main tools.....	38
6-2. Common tools .....	39
6-3. Measurement tools .....	40
6-4. Image Manipulation tools.....	41
6-5. Volume tools.....	42

# Preface

aaaa

# Chapter 1. Library Common

aaaaaaa

# Chapter 2. Library Imaging

Main goal of this library is effective implementation of pipeline computation on input datasets. Whole computation should be as parallel as possible in order to utilize resources available in modern processors, etc.

Design of interfaces and class hierarchies is aimed to extensibility and code reusability.

## 2.1. Compilation

### 2.1.1. Dependencies

This library should be linked together with libCommon. So all its dependencies come with it.

## 2.2. Architecture

All declarations are in namespace **M4D::Imaging**.

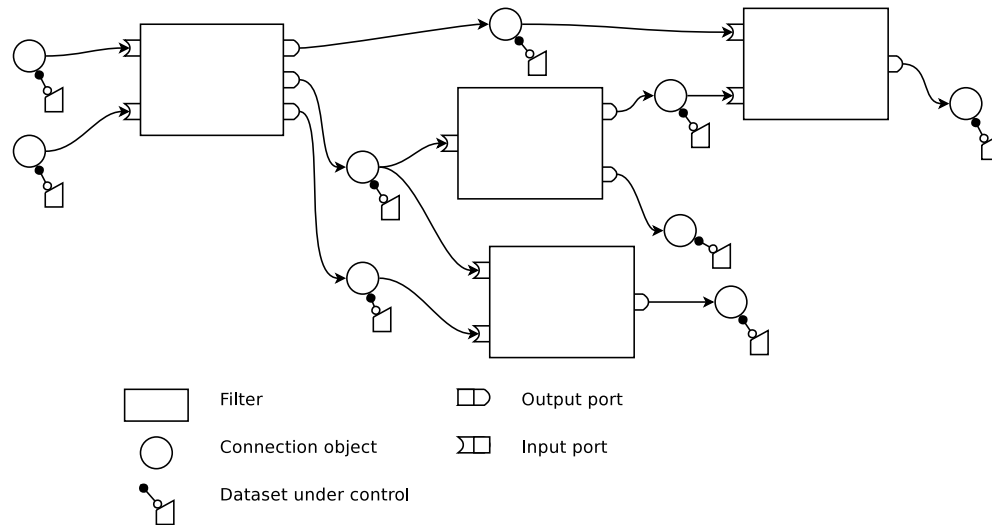
Data structures in library are designed to make possible construction of pipeline computation system as shown in scheme Figure 2-1. We now present all types of these data structures and their tasks.

- **Datasets.** Centre of whole design - hierarchy of classes prepared for storing data, which provide methods for access synchronization over data.
- **Filters.** Are objects planned as processors working on datasets. Only implemented branch of filters are pipeline filters. They get access to input datasets through input ports, do the computation and write results to output datasets obtained from output ports.
- **Ports.** Communication channels - their job is exactly as the name says. Filters can get access to input/output datasets and receive messages through ports.
- **Connections.** Connection objects are meant as bridge between objects with ports (filter-filter, filter-viewer). They own datasets and grant access to them through ports. Input ports have only read access and output ports have write access. Connection object oversee synchronization of readers and writer. Model is one-to-many, so there can be multiple readers, but only one writer.
- **Messages and message receivers.** Communications between objects in pipeline is assured by sending messages - successors of `PipelineMessage`. Objects which can receive messages have to implement interface `MessageReceiver` (ports, filters, etc.), there are also other interfaces - but this one is most important. If you make object with this interface, you can add message hook to connection object by



calling `AddMessageHook()`. Till this moment your object will obtain every message going through this particular connection object - handy for implementation of notifiers, progress watchers, etc.

**Figure 2-1. Pipeline scheme**



## 2.2.1. Datasets

Actual data are stored in proper descendants of class `AbstractDataSet`. Its hierarchy is shown in . Purpose of these classes is to provide access methods for reading and writing informations of certain type and optional synchronization.

By optional synchronization is meant set of synchronization methods, which are not called by access methods in class. And user should use these methods only in situations requiring synchronization. This less comfortable though, but more effective.

## 2.2.2. Filters

## 2.2.3. Connections

aaaaaaa

## 2.2.4. Ports

aaaaaaa

## 2.3. Most important classes

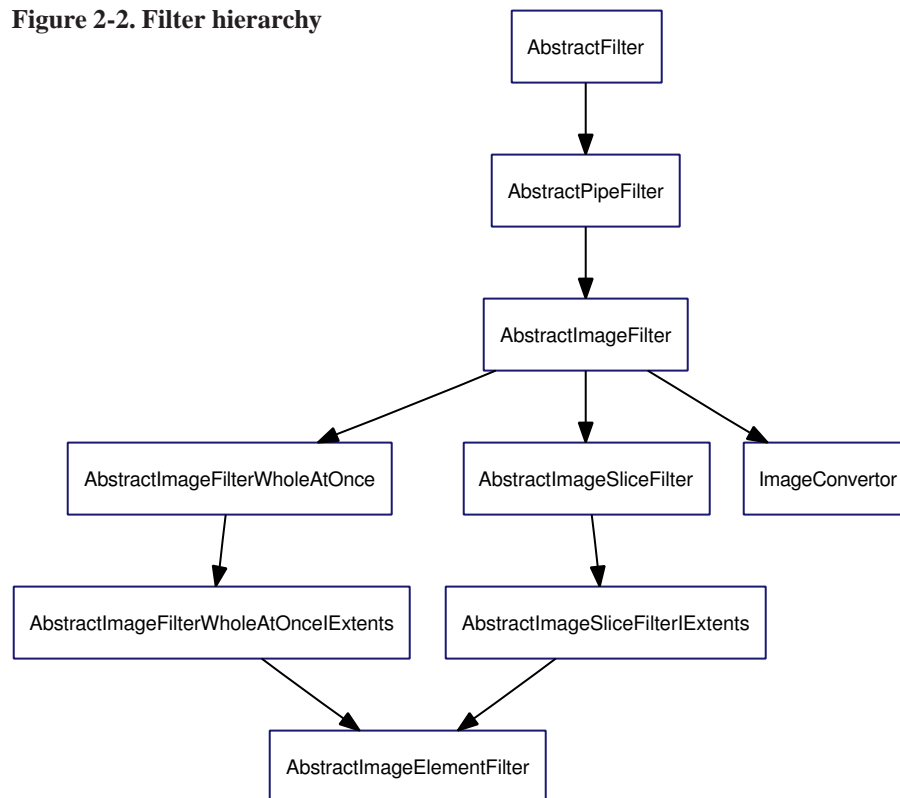
### 2.3.1. Dataset hierarchy

**AbstractDataSet.** Is predecessor of all datastructures containing data. Only concept implemented by this class is read/write locking system. This locking system ensure synchronization only on dataset structure (its extents, allocated buffers, etc.), not on data contained inside. Read/write lock let multiple readers to obtain access and writers have to wait. If there is at least one writer waiting, no other reader is allowed to obtain lock. And when all readers finish their work, first writer get exclusive access.

Synchronization on data should be implemented in successors, because it differ in each type of dataset. Changes in internal structure can be detected by comparing timestamps. When some change in internal structure of dataset has happened - timestamp is increased. So if you store value from previous access you can easily detect changes.

## 2.3.2. Filter hierarchy

Figure 2-2. Filter hierarchy



**AbstractPipeFilter.** Ancestor of pipeline filters. Public interface is extended with few methods modifying behaviour (ie. setting invocation style) and access methods to input ports and output ports. These ports are communication channels - can send and receive messages, get access to datasets, etc.

In nonpublic interface there are declared pure virtual and virtual methods with special purpose - they are called in predefined situations or in right order during computation. If somebody wants to create new pipeline filter, he must at least inherit its implementation from this class and override these methods : `ExecutionThreadMethod()`, `PrepareOutputDatasets()`, `BeforeComputation()`, `AfterComputation()`.

**AbstractImageFilter.** This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

**AbstractImageSliceFilter.** This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

**AbstractImage2DFilter.** This template is planned to be used as predecessor for filters procesing on two dimensional data. By that are meant 2D images and 3D images processed in slices. Output dataset proportions are set to the same values as input dataset, so only method to be overrided is `Process2D()`.

**AbstractImageElementFilter.** This template is prepared to ease design of image filters, which work on zero neighbourhood of element - use only value of the element. These filters work with output dataset with same extents as input.

Because calling virtual method consumes time - this template uses different way of implementation of actual computation - third parameter of template is functor which has implemented operator(), which takes two parameters - constant reference to input value, and reference to output value. This method is best to be inline and effective - its called on every element of input dataset.

**AbstractImage2DFilter.** This template is planned to be used as predecessor for filters processing on two dimensional data. By that are meant 2D images and 3D images processed in slices. Output dataset proportions are set to the same values as input dataset, so only method to be overridden is `Process2D()`.

**AbstractImageFilterWholeAtOnce.** This template is prepared for creation of image filters which need to access whole input dataset or for experimental implementation - without progressive computing.

Before call of `ProcessImage()` filter waits on read bounding box with same proportion as image and after that write bounding box containing output image is marked as dirty. After finished computation is this bounding box marked as modified or cancelled if computation did not finished successfully - `ProcessImage()` returned false.

In classes inheriting from this one you must override methods `ProcessImage()` and `PrepareOutputDatasets()`.

**AbstractImageFilterWholeAtOnceIExtents.** Same usage as template `AbstractImageFilterWholeAtOnce`, but only when input and output image are the same dimension and proportions.

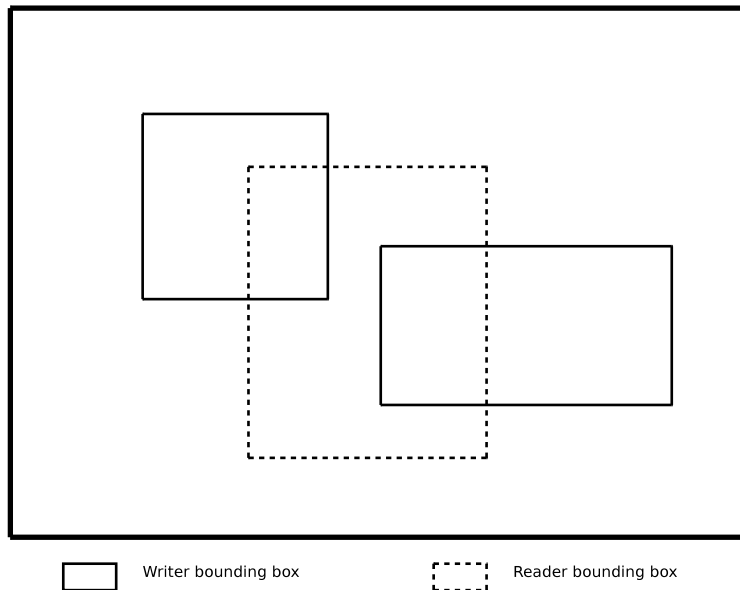
So only method you must override is `ProcessImage()`.

## 2.4. Main design concepts

## 2.4.1. Locking image parts

Figure 2-3

Figure 2-3. Synchronization over image data - 2D example



## 2.5. Usage

### 2.5.1. How to build a pipeline

First thing we should know, is how to create filtering pipeline with supposed behaviour using prepared filters. This section will be brief tutorial to pipeline construction. We show possible usage on example class that will have all desired abilities normally distributed across application.

Before we start construction, we must decide if we want handle storing and deallocation of all objects. There is prepared container for all pipeline objects, which can handle deallocation of stored objects and semiautomatically connects filters, and other objects with ports used as communication channel. For our example we use this `PipelineContainer`. For manual control and construction see its source code.

```
#include "Imaging/Image.h"
#include "Imaging/PipelineContainer.h"

const unsigned Dim = 3
```

```

typedef uint16 ElementType;

typedef M4D::Imaging::Image< ElementType, Dim > ImageType;

class ExamplePipelineHandler
{
public:
    //Default constructor - pipeline construction
    ExamplePipelineHandler();

    //Method which pass image to pipeline and start computation.
    void
    FilterImage( ImageType::Ptr image );

    //This method will be called, when filters finish.
    //Parameter tells, if computation passed withou problems.
    //Implementation of this method depends on its purpose (notify user, ...).
    void
    FinishNotification( bool succesfully )
    { /*Put your code here.*/ }

protected:
    M4D::Imaging::PipelineContainer    _pipeline;
    M4D::Imaging::AbstractPipeFilter  *_firstFilter;
    M4D::Imaging::AbstractImageConnectionInterface *_inConnection;
    M4D::Imaging::AbstractImageConnectionInterface *_outConnection;
};

```

Actual pipeline construction should proceed in three basic steps:

- Allocation of filters
- Establishing connections
- Setting message hooks

Now we discuss these steps in detail. With example implementation.

**Allocation of filters.** This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

**Establishing connections.** This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

**Passing input datasets and execution.** This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

```

#include "ExamplePipelineHandler.h"

ExamplePipelineHandler
::ExamplePipelineHandler()
{
    _firstFilter = new
    M4D::Imaging::AbstractPipeFilter *secondFilter = new

```

```

_pipeline.AddFilter( _firstFilter );
_pipeline.AddFilter( secondFilter );

_pipeline.MakeConnection( _firstFilter, 0, secondFilter, 0, true );

_inConnection =
    dynamic_cast< M4D::Imaging::AbstractImageConnectionInterface * >(
        &(_pipeline.MakeInputConnection( _firstFilter, 0, false )) );

_outConnection =
    dynamic_cast< M4D::Imaging::AbstractImageConnectionInterface * >(
        &(_pipeline.MakeOutputConnection( secondFilter, 0, true )) );
}

void
ExamplePipelineHandler
::FilterImage( ImageType::Ptr image )
{
    _inConnection->PutImage( image );

    _firstFilter->Execute();
}

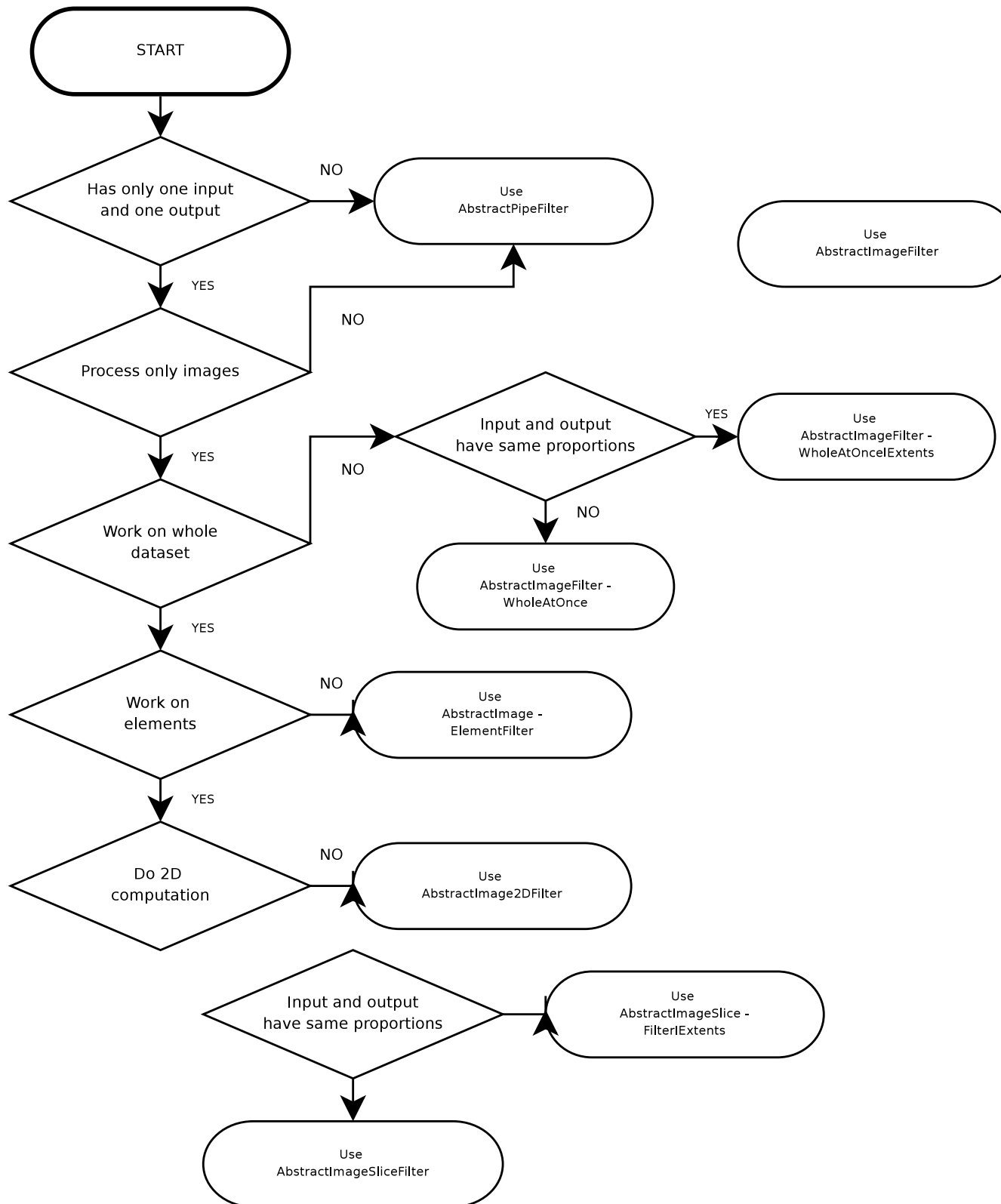
```

### 2.5.2. Creating new filter

One of the main advantage of this library is its extensibility. The biggest potential is easy way to add new filters with all features promised by pipeline design (synchronization, parallel execution, etc.). This is achieved through set of filter abstract classes, each designed for special purpose. Author of new filter have to consider few aspects (dimensionality, type of input/output data, way the filter computes, etc.) and choose right ancestor class for his filter.

To make decision easier you can use prepared flowchart Figure 2-4 and find most suitable ancestor class.

Figure 2-4. Filter decisive flowchart





Now if you have chosen right ancestor class, you can start with actual implementation of your filter. You should keep few rules and concepts, which can not only help you with writing, but even some other parts of library will just work without handling extra issues. This is achieved by generic design of whole library, templates are used almost everywhere.

We now introduce these rules and concepts, and show practical examples from library sources. First of all try design your filters as generic as possible. So try maximally use dataset traits (now only available `ImageTraits`), template specializations, etc. All ancestor classes are templated (with exception of `AbstractPipeFilter`), so it makes that easier.

Try to keep these rules, when designing filter interface:

- Filter class has public typedef to predecessor class with name `PredecessorType`.
- All properties of filter are in one public nested `struct Properties` deriving from `PredecessorType::Properties`. In case, that set of properties is empty, make at least public typedef to `PredecessorType::Properties` with name `Properties`.
- Prepare default constructor, and constructor with pointer to `Properties`. Default constructor creates default set of properties. In both constructors pointer to instance of `Properties` is passed to predecessor constructor in list of initializers. Using constructor with parameter will completely initialize filter with passed properties, no other method is needed to call.

If you follow mentioned rules you can use few preprocessor macros, which can simplify implementation of your class.

- Pointer to `Properties` structure is stored as protected member `_properties`, but its type is `AbstractFilter::Properties`. So if you want access members of your properties structure you must either cast to right type every time, or put preprocessor macro `GET_PROPERTIES_DEFINITION_MACRO` to private section of your class declaration. Now method `GetProperties()` returning reference to `Properties` is available for usage.
- To make declaration of Get/Set methods easier three macros are prepared:  
`GET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )`,  
`SET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )` and  
`GET_SET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )`. These macros will be unwinded into inline declarations of `TYPE Get'NAME'()const, void Set'NAME'( TYPE value )` and both. Parameter `NAME` is used in name of Get/Set method and parameter `MEMBER_NAME` is name of `Properties` member accessed by these two methods.

```
template< typename ElementType >
class ThresholdingFunctor
{
public:
    void
    operator()( const ElementType&    input, ElementType& output )
    {
        if( input < bottom || input > top ) {
```

```

        output = outValue;
    } else {
        output = input;
    }
}

ElementType    bottom;
ElementType    top;

ElementType    outValue;
};

template< typename ImageType >
class ThresholdingFilter
: public AbstractImageElementFilter<
    ImageType,
    ImageType,
    ThresholdingFunctor< typename ImageTraits< ImageType >::ElementType >
>
{
public:
    typedef ThresholdingFunctor
        < typename ImageTraits< ImageType >::ElementType >      Functor;

    typedef Imaging::AbstractImageElementFilter
        < ImageType, ImageType, Functor >                        PredecessorType;

    typedef typename ImageTraits< ImageType >::ElementType      InputElementType;

    struct Properties : public PredecessorType::Properties
    {
        Properties(): bottom( 0 ), top( 0 ), outValue( 0 ) {}

        InputElementType    bottom;
        InputElementType    top;

        InputElementType    outValue;

        void
        CheckProperties() {
            _functor->bottom = bottom;
            _functor->top = top;
            _functor->outValue = outValue;
        }

        Functor    *_functor;
    };

    ThresholdingFilter( Properties * prop );
    ThresholdingFilter();

    GET_SET_PROPERTY_METHOD_MACRO( InputElementType, Bottom, bottom );
    GET_SET_PROPERTY_METHOD_MACRO( InputElementType, Top, top );

```

```

        GET_SET_PROPERTY_METHOD_MACRO( InputElementType, OutValue, outValue );
protected:

private:
    GET_PROPERTIES_DEFINITION_MACRO;

};

template< typename ImageType >
ThresholdingFilter< ImageType >
::ThresholdingFilter()
    : PredecessorType( new Properties() )
{
    GetProperties()._functor = &(this->_elementFilter);
}

template< typename ImageType >
ThresholdingFilter< ImageType >
::ThresholdingFilter( typename ThresholdingFilter< ImageType >::Properties *prop )
    : PredecessorType( prop )
{
    GetProperties()._functor = &(this->_elementFilter);
}

```

### 2.5.3. Defining new dataset type

- Designing dataset interfaces and synchronization system
- Implementing connection objects and input/output ports
- Creating filter base classes working on this dataset type

## Chapter 3. DICOM Client library

This class is gate to DICOM world. DICOM is a standard describing creating, storing, manipulating and many more actions taken on medical data. For full description informations see (<http://www.dclunie.com/dicom-status/status.html>). Actual standard is 'little' bigger because it describes all possible scenarios of all possible types. So when you want to get inside the matter you can read only some parts of it. There are those parts referred in the code of particular parts.

Data are stored in files (\*.dcm). The files are divided into pairs of key and value. There is dictionary of keys. The keys changes with every new specification of the standard (yearly). There are set of commonly used types of files like Images, Waves of sound, CardioGrams, ... Each type has its own set of mandatory or optionally tags to be filled when created on a device. Most important tag is the one that contains actual data of the image. Although every DICOM file has set of common tags. These are used to identify the file. There are 4 levels of entities that identifies the files. These are: patient, study, series, image. So to fully identify one image 4 IDs must be provided.

On these tags DICOM server querying is performed, where data are normally stored. There are many implementations of that server. Most common is PACS server. Since there is no universal language SQL like system, actual querying is performed through dicom files. Filled tags means 'SQL WHERE clause', tags provided with no value defines 'wanted columns'. Although DICOM provides variety filetypes, we have focused on filetypes that provides images. Most used are: CT (Computer Tomography), MR (Magnetic Resonator), RT (radiation therapy). DICOM server implements searching (C-FIND: query file is send and resulting files are received), moving (C-MOVE: query file is send like in searching but with already known ID params that identify specific image or set of image. Then particular image files are received), storing (C-STORE: new image with unique generated IDs is send, not yet implemented) functionalities.

Library also provides local filesystem searching functionality. This is used when actual data are stored on filesystem.

The library uses 3rd party toolkit DCMTK by DICOM@Office ([dicom.offis.de](http://dicom.offis.de)) for manipulation with DICOM files (reading, saving, server querying).

### 3.1. DCMTK

DCMTK (DICOM ToolKit) is public domain library for handling DICOM files. It is written in C++. That is the main reason we chose it to use. Another reason was that it implements some example programs that you can compose working DICOM server. We use them to have a DICOM server and be able to test with it. Library also contains some programs that serves as DICOM client. From that programs our DICOM client code was derived.

That was definitely pros but there are some cons. For instance there are networking absolutely hidden to library user. So we had to use in manner the DCMTK authors wanted. But we want some scatter-gather

like behaviour to be able to make the incoming data be written to directly to one big memory array that is essential for image-processing algorithms operating upon the data. It was impossible to do that due to the networking hiding and impossibility knowing DICOM file sizes before actual data are received that would be needed to scatter-gather ops. So we agreed to following scenario: Whole file is received. Then type of the elements as well as image sizes are read from dataSet (see next) to set up the places in memory where the data should go. Then actual data is copied to the place from dataSet. And then are deleted from it. So whole dataSet with all attributes about the image but actual data remains ('empty bottle').

DataSet is map-like container for attributes (key-value pairs) that the DICOM file is composed from. When a file is being retrieved from server or from filesystem, the dataSet is built automatically by the DCMTK library. The building is atomical so no possibility of knowing some attributes before others (discussed above).

## 3.2. Compilation

### 3.2.1. Dependencies

Common library

## 3.3. Architecture

Public declarations are in namespace **M4D::Dicom** while private (not visible from outer) are in **M4D::DicomInternal**. Whole design of class hierarchies is Figure 2-1

### 3.3.1. DcmProvider

DcmProvider is the class all DICOM functionality is provided from. It has methods for communication with DICOM server and files or fileSets manipulation (C-FIND, C-MOVE, ...) as well as searching local filesystem folders. It also contains basic class that represents one DICOM file, DicomObj and some structures representing found results such as TableRow for view found information in table.

### 3.3.2. DicomObj

Represents one DICOM file. When retrieving whole series, then the DicomObjs are stored in vector (DicomObjSet). It has methods for Saving and Loading to filesystem. As well as method for copying data to defined place when 'overspilling the bottle'. It also contains methods to retrieve basic information from dataSet like width height, element's data type, ...

### 3.3.3. DicomAssociation

This is base class for DICOM association. Association is something like connection. It is defined by IP address, port and 'Application Entity' (AE). AE is like name. Both sides (client and server) has its own AE. This class contains pointers to DCMTK library objects that contains actual association and his properties. As well as some action members that establish (request), aborts and terminate the association. Next item contained in this class is address container that holds necessary properties for different association. Association has different properties when different services are called. So the container is a map indexed by string each called service. The container is filled from config file. There are some supporting methods taking care of it. The container is shared between all instances (static).

### 3.3.4. AbstractService

This is base class for all services that is requested to the side of DICOM server. There is pointer to DCMTK Network object which need network subsystem on windows system initialized at the beginning of usage and unloaded when is no more needed. So there is reference counting.

Each service is divided into 2 parts. SCP (Service Class Producer = server) and SCU (Service Class User = Client). Both sides of an service has to agree while establishing association what role to play. Normally DICOM server plays server role (SCP) for C-FIND, C-MOVE, C-STORE but for C-MOVE subassociations when image are transferred to client plays SCU role (it requests sending of data). Each scenario is described later in doc of successors. Another class member is query dataSet that is used as a query to server, similarly like SQL query string. Each query can be done on one of 4 levels: Patient, Study, Series, Images. For example: for Study level are all matched studies returned, for Series Level all matched series, ... On each level are relevant only specified set of matchable attributes so its quite hard to send robust query. Some other filtering has sometimes to be done on returned records.

Common scenario of all services is to prepare query dataSet that selects wanted data files. Then proceed the query to main TCMTK performing function and then retrieve resulting data through callbacks to final data structures. Ancesting classes implementing the services contain supporting callback definitions that cooperated with DCMTK functions and definitions of structures that are then passed to appropriate callbacks.

### 3.3.5. FindService

Implements C-FIND service to DICOM server. Process description in a nutshell: client (SCU) establish association to server (SCP) and sends query dataSet. Server process query dataSet and sends back matched results. For more details see DICOM doc ([ver]\_08.pdf chapter 9.1.2) and corresponding annexes).

### 3.3.6. MoveService

Implements C-MOVE service to DICOM server. Its purpose is to move data files (records) from DICOM server. There are two main functions that retrieve data files. One retrieve one SINGLE image. The image is specified by unique IDs on all levels (patient, study, serie, image). The other retrieve all images from specified serie (SET). That means specification of IDs on all levels but image. Process description in a nutshell: Client (SCU) establish association to server (SCP), send query dataSet, server find matching image files, then establish another subassociation (as SCU) with calling client (that plays SCP role) and transmit data files over the subassociation. For more details see DICOM doc ([ver]\_08.pdf chapter 9.1.4) and corresponding annexes).

### 3.3.7. StoreService

Implements service that performs C-STORE operation to DICOM server. Its purpose is to generate unique IDs and send new data to server. Behavior in a nutshell: Client (SCU) generates unique ID for sent (new) data, establish association with a server (SCP) and sends the data to server. For more informations see DICOM doc ([ver]\_08.pdf chapter 9.1.1) and corresponding annexes). Generation of unique IDs is based on prefix and the rest is delegated to DCMTK functions. More informations about unique IDs generation see DICOM doc.

### 3.3.8. LocalService

Implements searching and getting functions to local FS dicom files. It sequentially loads data files in specified folder (and subfolders through queue), read ID info, based on that info and given filter inserts or not inserts (if matching found) record into result. Each search run writes the folder that is performed on, build structure of information that is used when additional informations concerning data from the same run are required. One run is quite expensive while loading each file is needed (there is no other way how to read required IDs). So it is sensitive how wide and deep is the subtree of the given folder. Maybe some timeouts will be required. All functions are private because are all called from friend class DcmProvider.

## 3.4. Usage

Usage is quite simple. First we have to construct DcmProvider instance. Then we only create objects needed for holding data (ResultSet) and issue them along with some other parameters to member functions of DcmProvider class instance created at the beginning. Example follows:

```
#include "dicomConn/DICOMServiceProvider.h"

M4D::Dicom::DcmProvider dcmProvider; // DICOMProvider instance
// ....
```

```
// ....
// somewhere when finding record based on some filter form values

// create resultSet container
M4D::Dicom::DcmProvider::ResultSet resultSet;

// issue filter values and resultSet to provider method
dcmProvider.Find(
    &resultSet,
    firstName,
    lastName,
    patientID,
    fromDate,
    toDate,
    referringMD,
    description
);

// now we have in resultSet (vector) all found items ...
```



# Chapter 4. Data modifier

Is tool that scans given folder (recursively subfolders) and modify the information tag values within found DICOM files. This is because all data that we dispose has anonymized these values nad so when query is performed its results are 'nothing saying' crap. So thats why we developed this tool that generate new values based on dictionary from config file.

It scans files, builds tree according IDs within the files to assign same generated values to files within the same study, and changes the values generated from dictionary.

Values that are (currently) chaged are:

- Patient sex
- Patient name (based on sex)
- Patient birth date
- Study date (from given interval)

There are another mode of this tool. When --info param is given, it only scans the files and builds the tree. At the end is the tree written and you can see what data is within given directory.

## 4.1. Compilation

### 4.1.1. Dependencies

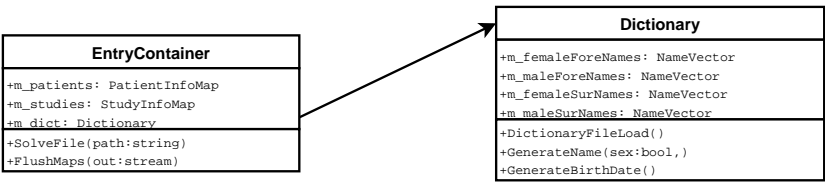
Common library

DCMTK library

## 4.2. Architecture

Everything is in namespace **M4D::DataModifier**. Whole design of class hierarchies is Figure 4-1

Figure 4-1. Module scheme



### 4.3. Usage

You can edit the dictionary files foreNames.cfg, sureNames.cfg which should be placed in working directory of the dataModifier executable.

Informations about parameters run the tool without params.

# Chapter 5. Remote computing (CellBE library)

Library used to send some parts of pipeline to remote machines to be executed and result sent back. The name is from Cell Broadband Engine architecture name coming from IBM. This architecture contains also supercomputers like Blade servers as well as not so huge system like the one in PlayStation3 console. CellBE should originally be the one that have be target of remote computing because there is some machines available for testing on faculty. But This library is not bound to specific architecture. It was one of primary request to be platform independent. It has to eliminate such scenarios that is usual in hospitals: 'Here you have the CT machine with software that can work and communicate ONLY with our supercalculating server that is bundled with. All for such small price of few millions'.

## 5.1. Compilation

### 5.1.1. Dependencies

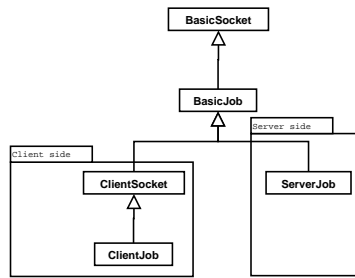
Imaging lib

## 5.2. Architecture

All declarations are in namespace **M4D::CellBE**. System is classic Client-Server architecture. The client parts is in the library while the server part in another project. Purpose of the library is to be linked with main program that has to have ability of remote computing. The both parts have some parts in common that reflects class hierarchy. Figure 5-1

### 5.2.1. Job

Job is entity that represents remote computation. Job has 2 main parts: Defining Container and input&output dataSets. Defining Container contains information about filters that the remote pipeline represented by this job will consist of and their settings. Each filter has Properties inner class. Instance of that class give us all necessary information of that filter because it is templated as well (it's inner class of templated class) and has all properties of filter that represents. So we don't need to instantiate any filter when defining remote pipeline. Filter::Properties instances are satisfied. According these information is actual pipeline created on their server side. Input&output dataSets are actual dataContainers that are dataRead from and send to server (input) and received and written to (output). When you look at the class hierarchy, you can see there are common parts for job on client and server side. But both 'types' of job has different behaviour when on client side and server side. So 2 branch in hierarchy reflects that. Each hierarchy member will be described now in more details:

**Figure 5-1. Hierarchy of classes composing job**

### 5.2.1.1. BasicSocket

Base class containing network functionality. Used BOOST::Asio for asynchronous networking and scatter gather. It contains pointer to socket that all the communication is performed through.

### 5.2.1.2. ClientSocket

Client has to be able to establish connection to server. This class can do it.

### 5.2.1.3. ClientJob

Now we focus on particular branches of the hierarchy. Lets start with client one.

This class advances functionality for special for client job behaviour. That is ID generation, sending definition container and his content. It has also supporting members special for client async operations as well as members that do them.

### 5.2.1.4. ServerJob

The last class in Job hierarchy remains. This class form server side brach.

Is opposite to ClientJob. It's purpouse is to handle operation that the client job performs (definition container unpacking and building pipeline and creating some supporting structures that are used in re-recieving definition container content. Will be dicused later on)

## 5.2.2. Serializers

Because this system (or library) is closely related to Imaging library that we wanted to let independent to any other library we have to solve how to perform serialization on the objects from the Imaging library that are mostly templated. Serializers object was developed.

Serializer is object that can perform serialization of its content. We have 2 types of serializers. These are `FilterSerializers` perform Serialization of `Filter::Properties` classes. Used in sending definition container. And `DataSetSerializers`. These performs `DataSet` serialization. Each Serializer performs serialization of 2 main types. First is class info serialization (CIS). In this stage is template parameters along with class type serialized. These information should be enough for other side for creating appropriate templated class instance. The second serialization tier is for actual content serialization (ACS) of entities (properties for `filterProperties`, `DataSet` attributes for `DataSet`).

The whole library is closed system that should not be changed. Only these Serializers are subjects to edit. Serializers reflects hierarchy of objects in Imaging library. So when some new item in Imaging library appears and its author wants it to be able to use it remotely so new serializer has to be written and registered. Registering is done in according type of `GeneralSerializer`, that serves as recognizer of particular serializers. Details of creating new Serializer will follow later on. Now we will discuss each type of serializer in more details.

### 5.2.2.1. FilterSerializer

Each `filterSerializer` (FS) should be derived from `AbstractFilterSerializer` class which is the base class for every FS and defines interface of its behaviour. CIS is performed by `SerializeClassInfo` & `DeSerializeClassInfo` pair of member functions. ACS by `SerializeProperties` & `DeSerializeProperties` pair.

### 5.2.2.2. DataSetSerializer

`DataSetSerializer` has more work to do than `FilterSerializer`. It performs CIS (within `GeneralDataSetSerializer` and appropriate switch cases, described later) and ACS in `SerializeProperties` & `DeSerializeProperties` pair of functions that are in interface defined by `AbstractDataSetSerializer` that each new `DataSet` should inherit.

New is Actual Data Serialization (ADS). This is performed through another part of `AbstractDataSetSerializer` interface parts: `Serialize`, `OnDataPieceReadRequest`, `OnDataSetEndRead` functions. For more detail of these function purpose see appropriate headers.

Main idea of `DataSet` serialization is to divide the whole `DataSet` into smaller parts, that can be transported through network separately (`dataPieces`, DP) and alternatively the calculation can be started on independently from other DP that have not yet arrived. Each DP has its header that says how long it is.

So on receiving side when such header is received it's passed to `DataSetSerializer's OnDataPieceReadRequest` method that will decide where the data that the DP refers to are going to be received. This can tell through `DataBufs` vector by putting `DataBuf` struct into it. `DataBuf` is struct of `void *` pointer pointing somewhere into memory and size. So the `DataBufs` vector defines place where data of DP should be transmitted (in general manner).

Another mandatory function is `OnDataSetEndRead`. This is called when whole `DataSet` is received. No more DP are going to be received. This is recognized by receiving of special `DBHeader`. Function can perform some clean up or something.

`Serialize` is another mandatory function. It has to perform actual whole ADS. It gets pointer to `iPublicJob` interface and through its function `PutDataPiece` should perform serialization. `PutDataPiece` is the same as in deserialization but it writes data into network in scatter manner. In this function the programmer should define meaning of DP and reflect the meaning in deserialization functions. Perfect example is serialization of 3d image. DP is one slice. So whole `DataSet` is sent with slice granularity. On the other side DP (slice) is received directly into the right place in big array for 3d image and calculation can be started if another slices not needed. This is ideal case. Current implementation does not allow it due to `DataSet` locking and lack of universal `DataSet` handling. So `DataSet` is currently transferred whole and whole received.

Because this serialization is stateful the inner state has to be in serializer instance. `Reset` mandatory method should reset the state for serializer reuse.

### 5.2.3. CellClient class

`CellClient` represents the gate to remote computing. Each job on client side is created through this class by `CreateJob` function. This class also contains container of available servers that can be used for remote computing. It can also load it from config file.

### 5.2.4. Remote Filter

Remote filter is filter that represents remote pipeline. It shall create definition container that is needed to create Job, that represents remote computation. It shall contain `filterProperties`, that corresponds to the filter in remote pipeline which compose. Because everything is templated there is not yet no other way of creating remote pipeline than define it through this filter properties, that is members of remote filter (they are some kind hardcoded). That means if you want to create another remote filter with different pipeline within, you have to write new successor of `RemoteFilter`. So every remote filter MUST inherit from `RemoteFilter`.

To have hierarchy complete there is `RemoteFilterBase`. This class is ancestor of `RemoteFilter` and its purpose is to make `CellClient` to the successors to make job creating possible.

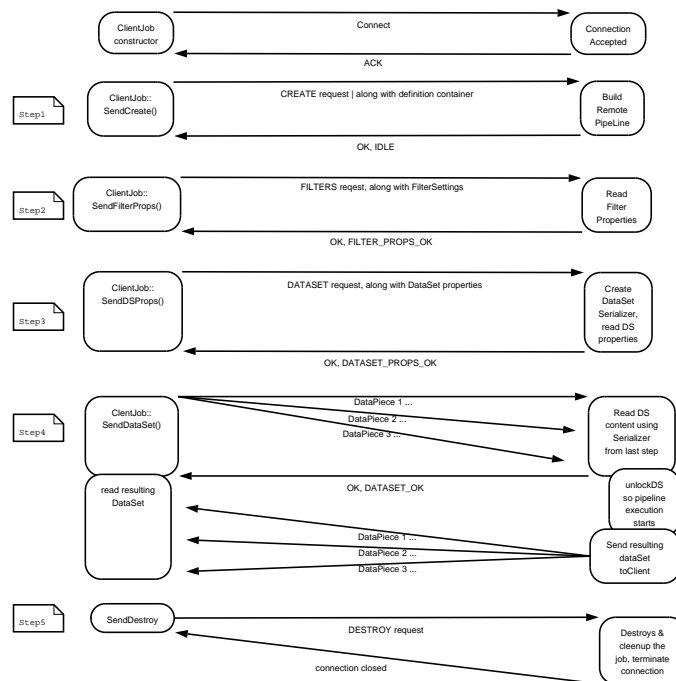
## 5.2.5. Server

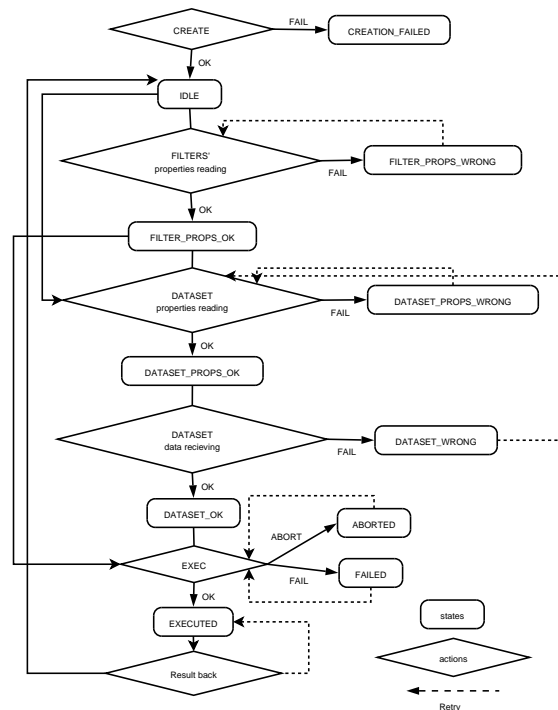
Server is whole implemented in asynchronous manner. Base consists `Server`. Main part is `JobManager` that stores all jobs that are currently on the server. It is associative container with `JobID` key. Only thing that server can do accept connections and read primary header.

When accepts connection, reads `PrimaryHeader` and based on `PrimaryHeader::action` it decides what to do. For action definitions see `BasicJob`. Primary action performed by `Server` is job creation. It creates new `ServerJob` instance, adds it to the `JobManager` and then all work is delegate to this job. Then the job receive other commands (the socket that is created in accept is given to job) and performs appropriate actions.

Server can also understand other actions. This is done for case when connection to client is broken. After reconnect server has to react to all types of action and associate received request to existing job, give new socket to it and delegate reaction to the existing job.

**Figure 5-2. Job life cycle**



**Figure 5-3. Job states**

## 5.3. Usage

### 5.3.1. FilterSerializer

Next is the list of necessary steps for adding new FilterSerializer:

- Add enum FilterID item in cellBE/filterIDEnums.h header
- Write actual FilterSerializer derived from AbstractFilterSerializer and implementing its interface. Right place where the code to place is cellBE/FilterSerializers folder
- Register new Serializer by adding instance into array in FilterSerializerArray class constructor and appropriate edit size of the array

Example of creation of a new FilterSerializer (taken from code. For details see ThresholdingSerializer.h & .tcc). It uses supporting function, that is called right after CIS.



```

// supportig templated function, that creates actual instances
template< typename ElementType, unsigned Dim >
void
CreateThresholdingFilter(
    M4D::Imaging::AbstractPipeFilter **resultingFilter
    , AbstractFilterSerializer **serializer
    , const uint16 id
    , M4D::CellBE::NetStream &s )
{
typedef typename M4D::Imaging::Image< ElementType, Dim > ImageType;
typedef typename M4D::Imaging::ThresholdingFilter< ImageType > Filter;
    typedef typename FilterSerializer< Filter > FilterSerializer;

Filter::Properties *prop = new Filter::Properties();

*resultingFilter = new Filter( prop ); // filter creation
*serializer = new FilterSerializer( prop, id); // appropri. serializer
}

////////////////////////////////////

// actual FilterSerializer class

template< typename InputImageType >
class FilterSerializer< M4D::Imaging::ThresholdingFilter< InputImageType > >
: public AbstractFilterSerializer
// inheritance from AbstractFilterSerializer
{
public:
    // typedef to Properties of filter this serializer is for.
    // Just for simpler usage
typedef typename M4D::Imaging::ThresholdingFilter< InputImageType >
    ::Properties Properties;

// ctor - subject of customization, must call typed ancestor ctor
FilterSerializer( Properties * props, uint16 id)
: AbstractFilterSerializer( FID_Thresholding, id )
, _properties( props )
{}

// member performing CIS
void SerializeClassInfo( M4D::CellBE::NetStream &s)
{
    s << (uint8) ImageTraits< InputImageType >::Dimension;
s << (uint8) GetNumericTypeID< ImageTraits< InputImageType >
    ::ElementType >();
}

// member performing CIS deserialization
void
DeSerializeClassInfo(
    M4D::Imaging::AbstractPipeFilter **resultingFilter

```

```

        , AbstractFilterSerializer **serializer
        , const uint16 id
        , M4D::CellBE::NetStream &s
    )
    {
        uint8 dim;          // read from netStream
uint8 typeID;
s >> dim;
s >> typeID;

// build appropriate classes based read info with assistance of
// CreateThresholdingFilter templated function defined above
NUMERIC_TYPE_TEMPLATE_SWITCH_MACRO( typeID,
DIMENSION_TEMPLATE_SWITCH_MACRO(
    dim, CreateThresholdingFilter<TTYPE, DIM >(
        resultingFilter, serializer, id, s ) )
);
    }

    // pair of members performing ACS
void
SerializeProperties( M4D::CellBE::NetStream &s)
{
s << _properties->bottom << _properties->top << _properties->outValue;
}
void
DeSerializeProperties( M4D::CellBE::NetStream &s )
{
s >> _properties->bottom >> _properties->top >> _properties->outValue;
}

protected:
Properties *_properties;    // pointer to properties (actual content)
};

```

Here is snippet from FilterSerializerArray.cpp constructor where actual registration of a new FilterSerializers should be performed. FilterSerializerArray is based on array of filterSerializers. Each serializer on position defined by it ID in FilterID enum.

```

// within FilterSerializerArray constructor ....
// creation of buddy instance of our new serializer class on place within
// the array defined by our new enum member (step 1)
m_serializerArray[ (uint32) FID_Thresholding] =
    new FilterSerializer< typename ThresholdingFilter< Image<uint8, 3> > >(
        NULL, 0 );

```

```
// .....
```

### 5.3.2. DataSetSerializer

Next is the list of necessary steps for adding new DataSetSerializer:

- Add enum DataSetType item in Imaging/dataSetClassEnum.h header
- Write actual DataSetSerializer derived from AbstractDataSetSerializer and implementing its interface. Right place where the code to place is cellBE/DataSetSerializers folder
- Register new DataSetSerializer by adding new switch case of new enum member (step 1) into GeneralDataSetSerializer::GetDataSetSerializer method and writing the content of switch case that instantiates DataSetSerializer based on inner properties (like TypeID, Element type, dimension. Properties of image). And another switch case in GeneralDataSetSerializer::DeSerializeDataSetProperties method that on the other side can instantiate DataSetSerializer class (CIS).

Here is snippet from ImageSerializer. Example of SerializeProperties & DeSerializeProperties functions. Here used to write information about dimensionality, element type, sizes, etc. relevant for images.

```
template< typename ElementType, uint8 dim>
void
ImageSerializerBase<ElementType, dim>
    ::SerializeProperties(M4D::CellBE::NetStream &s)
{
    AbstractImage *im = (AbstractImage *) m_dataSet; // cast to sucessor

    // serialize common properties
    s << (uint16) im->GetDimension() << (uint16) im->GetElementTypeID();

    for( unsigned i = 0; i < im->GetDimension(); ++i ) {
        const DimensionExtents &dimExtents = im->GetDimensionExtents( i );

        s << (int32)dimExtents.minimum;
        s << (int32)dimExtents.maximum;
        s << (float32)dimExtents.elementExtent;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```

template< typename ElementType, uint8 dim>
M4D::Imaging::AbstractDataSet::ADatasetPtr
ImageSerializerBase<ElementType, dim>
    ::DeSerializeProperties(M4D::CellBE::NetStream &s)
{
    int32 minimums[ dim ];
    int32 maximums[ dim ];
    float32 elExtents[ dim ];

    for( unsigned i = 0; i < dim; ++i ) {

        s >> minimums[ i ];
        s >> maximums[ i ];
        s >> elExtents[ i ];
    }

    NUMERIC_TYPE_TEMPLATE_SWITCH_MACRO( GetNumericTypeID< ElementType >(),
    return M4D::Imaging::ImageFactory::CreateEmptyImageFromExtents< TTYPE >( dim, minimums, maxi
    );

    // unreachable but for syntactic corectness
    return M4D::Imaging::AbstractDataSet::ADatasetPtr();

}

```

Here is a nother snippet from ImageSerializer (3D case). Example of Serialize & OnDataPieceReadRequest functions. That performs ACS.

Serialize goes through image slice by slice and send each as dataPiece through iPublicJob::PutDataPiece.

OnDataPieceReadRequest should decide where to put data piece that is going to be recieved and corresponds to dataPieceHeader given to this method. Memory places where the data piece should be recieved to is specified by DataBuff strucutres that is put into bufs container that comes as well within param. Note state variable m\_currSlice that says which slice is going to be recieved. Such variables are subject of Reset.

```

template< typename ElementType>
void
ImageSerializer< typename ElementType, 3>
    ::Serialize( M4D::CellBE::iPublicJob *job)

```

```

{
Image<ElementType, 3> *im = (Image<ElementType, 3> *) m_dataSet;

uint32 width;
uint32 height;
uint32 depth;
int32 xStride;
int32 yStride;
int32 zStride;
ElementType *pointer = im->GetPointer(
    width, height, depth, xStride, yStride, zStride);

    // put slices as dataPieces. Suppose whole DS is serialized. Not only window part
    DataBuff buff;

    size_t sliceSize = width * height;

for( uint32 k = 0; k < depth; ++k ) {
    buff.data = (void*) pointer;
    buff.len = sliceSize * sizeof( ElementType);
    job->PutDataPiece( buff);

    pointer += sliceSize; // move on next slice
}
}

////////////////////////////////////

template< typename ElementType>
void
ImageSerializer< typename ElementType, 3>
::OnDataPieceReadRequest( DataPieceHeader *header, DataBuffs &bufs)
{
    Image<ElementType, 3> *im = (Image<ElementType, 3> *) m_dataSet;

    uint32 width;
uint32 height;
uint32 depth;
int32 xStride;
int32 yStride;
int32 zStride;
ElementType *pointer = im->GetPointer( width, height, depth, xStride, yStride, zStride );

    size_t sliceSize = width * height;

    DataBuff buf;
    buf.data = pointer + ( sliceSize * m_currSlice);
    buf.len = sliceSize * sizeof( ElementType);

    bufs.push_back( buf);

    m_currSlice++; // important
}

```

Here is example of new dataSet registration in GeneralDataSetSerializer. Actual registration is performed by adding another switch case of dataSet type which serializer is being registered.

The same way should be switch case added to DeSerializeDataSetProperties.

```
AbstractDataSetSerializer *
GeneralDataSetSerializer::GetDataSetSerializer( AbstractDataSet *dataSet)
{
    switch( dataSet->GetDatasetType() )
    {
        case DATASET_IMAGE:
            NUMERIC_TYPE_TEMPLATE_SWITCH_MACRO(
                ((AbstractImage *)dataSet)->GetElementTypeID(),
                DIMENSION_TEMPLATE_SWITCH_MACRO(
                    ((AbstractImage *)dataSet)->GetDimension(),
                    return new ImageSerializer< TTYPE, DIM >(dataSet) )
                );
            break;

        case DATASET_TRIANGLE_MESH:
            return NULL;
            // TOBEDONE WHEN triagleMesh created
            break;
        ...
        ...
        ...
    }
}
```

### 5.3.3. Remote filter

Next is the list of neccessary steps for writing new remote filter:

- Write new class that inherits from RemoteFilter templated class and implements all neccessary methodes (PrepareOutputDatasets, constructor)
- In that new class should be all Filter options classes that compose the remote pipeline added as member variables (hardcoded).

Here is BoneSegmentationRemote filter. It contains thresholding and median filter. (See m\_thresholdingOptions and m\_medianOptions members). This filter properties offers to the outer world by 'Get' methodes to let them to be changed as well as definition the two mandatory necessary methodes. There are some typedef that make the template madness easier.

```
template< typename ImageType >
class BoneSegmentationRemote
    : public RemoteFilter<ImageType, ImageType>
{
public:
    typedef typename RemoteFilter<ImageType, ImageType> PredecessorType;
    typedef PredecessorType::Properties Properties;

    BoneSegmentationRemote();

    //////////////// To customize ////////////////
    // putting options available to outer world to be able to specify it ....

    // thresholding filter issues
    typedef ThresholdingFilter<ImageType> Thresholding;
    typedef typename Thresholding::Properties ThresholdingOptsType;

    ThresholdingOptsType *GetThreshholdingOptions( void)
    {
        return &m_thresholdingOptions;
    }

    // median filter issues
    typedef MedianFilter2D<ImageType> Median;
    typedef typename Median::Properties MedianOptsType;

    MedianOptsType *GetMedianOptions( void)
    {
        return &m_medianOptions;
    }

protected:
    void PrepareOutputDatasets();

private:

    /**
     * Here should be added members of filter options type that will
     * define the remote pipeline this filter represents. Each member
     * for single filter in remote pipeline. As a next step is defining
     * retrieving public members, that will provide ability to change
     * the filter options from outer world.
     */
    ThresholdingOptsType m_thresholdingOptions;
```

```

    MedianOptsType m_medianOptions;
// ...

};

```

Here is implementation of BoneSegmentationRemote constructor. This way will be implemented all remote filters constructors. There are two phases. The first: creation of defining container (vector of FilterSerializers) and putting Serializers that correspond to filter properties member variables into it. The second stage is creating job while passing that defining container.

```

template< typename ImageType>
BoneSegmentationRemote<ImageType>::BoneSegmentationRemote()
: PredecessorType( new Properties() )
{
    AbstractFilterSerializer *ser;

    // definig vector that will define actual remote pipeline
    FilterSerializerVector m_filterSerializers;

    uint16 filterID = 1;

    // put into the vector serializers instances in order that is in remote pipe
    {
        // insert thresholding serializer
        ser = GeneralFilterSerializer::GetFilterSerializer<Thresholding>(
            &m_thresholdingOptions, filterID++);
        m_filterSerializers.push_back( ser);

        // insert median serializer
        ser = GeneralFilterSerializer::GetFilterSerializer<Median>(
            &m_medianOptions, filterID++);
        m_filterSerializers.push_back( ser);

        // ... for other possible members definig remote pipe filters
    }

    // create job
    m_job = s_cellClient.CreateJob( m_filterSerializers);
}

```



# Chapter 6. User Manual

## 6.1. Getting Started

...an application used for viewing and manipulating medical images. Digital images and data from various sources (including CT, MR, US units, computed and digital radiographic devices, secondary capture devices...

### 6.1.1. System requirements

This section describes the hardware and software required to run the application.

#### 6.1.1.1. Required hardware

**The application must be run on a computer that meets the following hardware requirements:**

- Pentium II CPU
- 256 MB RAM
- 10 MB free hard drive space for application + for image storage
- 32 MB of video RAM
- Minimum display resolution 1024 x 768

#### 6.1.1.2. Recommended hardware

**To get the most out of the application, it's recommended that your computer meet the following specifications:**

- Pentium 4 CPU
- 1 GB RAM
- 128 MB of video RAM
- Display resolution 1280 x 1024

#### 6.1.1.3. Supported operating systems

- Windows 2000, Windows XP or Windows Vista
- \*nix

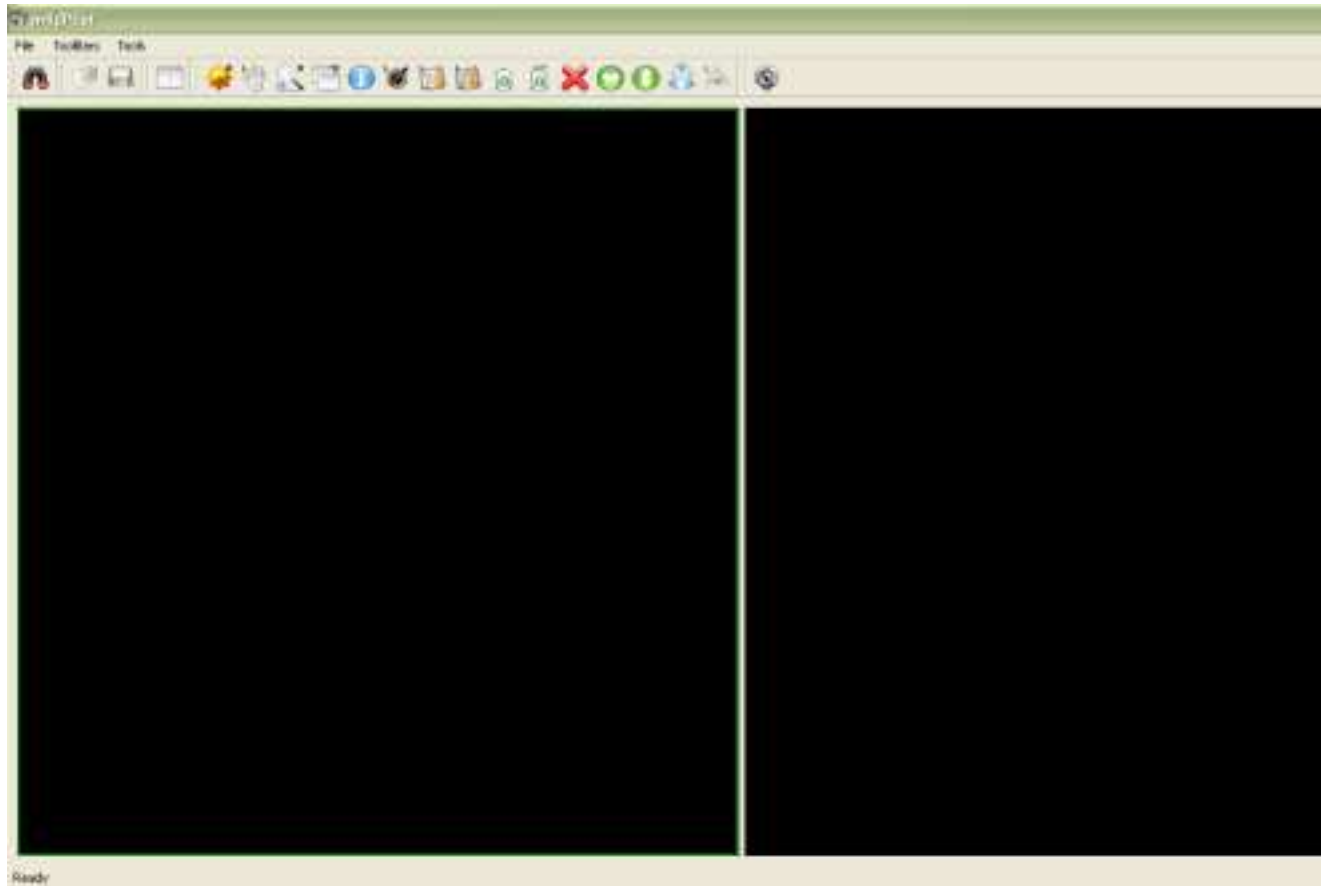
## 6.1.2. Installing the application

It's no need to install the application. Just copy the application and the configuration file to PC hard drive and then simply start it.

## 6.2. Using the application

### 6.2.1. Application window

The main application window is a workspace where you can view and work on retrieved images.



This window can contain more than one image at a time, each in a separate pane arranged in a grid. The menu bar appears at the top of the window, and the status bar will appear at the bottom (if enabled). By default, the tool bar appears at the top of the window directly beneath the menu bar, but you can move it to the bottom, left, or right of the window in order to accommodate your preferences - by dragging it to a

new location.

## 6.2.2. Using the toolbar

Tools in the toolbar are displayed based on the toolbar settings and activated according to the selected viewer type.

To view the description of a tool, hold the cursor over its icon. Full descriptions of the tools can be found in Section 6.2.3.

### 6.2.2.1. Moving the toolbar

The toolbar does not have to remain at the top of the window. It (its components) can be moved to the bottom, left, or right of the window in order to accommodate your preferences - by dragging it to a new location.

### 6.2.2.2. Customizing the toolbar

If you do not wish to view all of the tools in the toolbar, you can customize which tools will be displayed in the toolbar by right-clicking on the toolbar region and selecting wanted groups of tools.

You can enlarge the toolbar icons by clicking **Medium** or **Large** on the **ToolBars** menu.

#### **To customize the toolbar (viewer tools):**

- Click **ToolBars**→**Customize (Ctrl-M)**.

The Viewer Tool Property window appears (row ~ tool).



- Add/remove tools to the customized toolbar by clicking the visibility icon ( ✓ / ✗ ).
- Click on tools shortcut to edit them (doubleclick - edit, click - rewrite) - if the new value is not suitable for use as a shortcut, it will be reset to its previous value. You can now activate the tool using the assigned shortcut and modifier key combination (if you save your settings).
- Click OK to save your changes, or click Cancel to exit without saving any changes.

## 6.2.3. Using tools

### 6.2.3.1. Main tools

Table 6-1. Main tools

**Search**



Search for patient studies available for viewing

**Open**



Open an existing document from disk or network file system

**6.2.3.2. Common tools**

**Table 6-2. Common tools**

**Screen Layout**



Redisplay series and images in various layouts

**Window/Level**






Adjust the brightness and/or contrast of the image

**Pan**





Reposition the images in the window

<b>Zoom</b> 	Increase or decrease the image's field of view
<b>Stack</b> 	Scroll through images within a series
<b>Toggle Overlay</b> 	Hide or display the study information

### 6.2.3.3. Measurement tools

Table 6-3. Measurement tools

<b>Probe Tool</b> 	Give a pixel value for a given point
<b>New Point</b> 	Create a new point

**New Shape**

Start a new shape



**Clear Point**

Clear last created point



**Clear Shape**

Clear last created shape



**Clear All Points/Shapes** Clear all selections (points, shapes)



### 6.2.3.4. Image Manipulation tools

Table 6-4. Image Manipulation tools

**Flip Horizontal**

Flip the selected image from left to right about the vertical axis



**Flip Vertical**

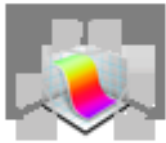
Flip the selected image from top to bottom about the horizontal axis

**Slice Orientation**

Switch the viewing axis orientation ( $x \sim y$ ,  $y \sim z$ ,  $z \sim x$ )

**6.2.3.5. Volume tools**

Table 6-5. Volume tools

**Rotate Volume**

Rotates the volume about the three axes


## 6.3. Retrieving and Viewing Images

A study is a set of related images which can be displayed and manipulated in the application. You retrieve images from both local and remote exam studies.

### 6.3.1. Using the Study Manager

The Study Manager window allows you to search for a study to view and manipulate with the application.

**To access the Study Manager window:**

- File → Search (Ctrl-S)
- Click 



The Study Manager window appears.

The screenshot shows the Study Manager window with the following sections:

- Filter:**
  - Buttons: Search, Today, Yesterday, Clear Filter, Options.
  - Fields: Patient ID, Last Name, First Name (each with a dropdown arrow).
  - Fields: From: 23..8..2008, To: 23..8..2008 (each with a dropdown arrow).
  - Fields: Accession#, Study Desc., Referring MD (each with a dropdown arrow).
- Modality:**
  - Checkboxes: All, CR, ES, NM, R, CT, MG, OT, R, DX, MR, PT, S.
- Study List:**
  - Buttons: View, Recent Exams, Remote Exams, DICOMDIR.
  - Table headers: Patient ID, Name, Modality, Description, Date, Time, Study ID, Sex, BI.

The **Recent Exams** tab lists the studies that were recently viewed - it has two modes: Recent Remote Exams (recently retrieved and viewed from available DICOM server) and Recent DICOMDIR (recently viewed studies that are stored in DICOMDIR format) (see Section 6.3.1.1).

The **Remote Exams** tab lists the studies stored on available DICOM server. If you want to view one of these exams, you can select it and it will be retrieved for you (see Section 6.3.1.2).



The **DICOMDIR** tab lists the studies that are stored in DICOMDIR format on either a CD, your workstation's hard drive, or a mapped network drive (see Section 6.3.1.3).

All the tabs can be customized to suit your preferences by re-sorting the columns in the exam lists. Click a header to sort the list according to that heading. For example, click Patient Name to sort the list alphabetically. Clicking the header field again will sort the list in the reverse order.

### 6.3.1.1. Searching for Recent Exams

Recent Exams are studies that were recently viewed - it has two modes: Recent Remote Exams (recently retrieved and viewed from available DICOM server) and Recent DicomDIR (recently viewed studies that are stored in DicomDIR format).

#### To search for a Recent Exam:

1. Click the **Recent Exams** tab.
2. Click the  or the  icon to select the Recent Exams mode - Recent Remote Exams (recently retrieved and viewed from available DICOM server) and Recent DicomDIR (recently viewed studies that are stored in DicomDIR format).
3. Optionally, filter the search by entering the search criteria. Enter either a single criterion or a combination of: Patient ID, Last Name, First Name, Accession Number, Study Description, or Referring M.D.
4. Optionally, enter a range of dates in which to search. Select the From: and To: check boxes to activate them, and then enter the date parameters either by hand or by using the calendar window by clicking on the date field drop-down list.



If you know that the study was performed today, click **Today**. Today's date appears in the date boxes. If you know that the study was performed yesterday, then click **Yesterday**, and that date will appear in the date boxes.

5. Optionally, filter the search by Modality type. Select the All check box to include all modality types in the search, or clear it to filter by specific modality types, which can be selected by clicking each modality type's corresponding check box.
6. Click **Search**. A study list appears in the bottom half of the Study Manager window.

To view studies without filtering, clear all of the filters by clicking **Clear Filter**, and click **Search**.

#### To view a Recent Exam (choose one of the following options):

- Select a study from the list and double-click it to view it automatically
- Select a study from the list and click View

### 6.3.1.2. Searching for Remote Exams

Recent Exams are studies that are stored on available DICOM server. If you want to view one of these exams, you can select it and it will be retrieved for you.

#### To search for a Remote Exam:

1. Click the **Remote Exams** tab.
2. Follow steps 3 through 6 of Section 6.3.1.2 to complete the search.

#### To retrieve and view a Remote Exam (choose one of the following options):

- Select a study from the list and double-click it to retrieve and view it automatically
- Select a study and click View

These studies will be available in the Recent Remote Exams mode.

### 6.3.1.3. Searching for DICOMDIR Exams

DICOMDIR studies are stored in DICOMDIR format on any folder accessible via file systems such as CDs, removable file systems such as memory sticks, your workstation's hard drive, or a mapped network drive.

#### To search for a DICOMDIR Exam:

1. Click the **DICOMDIR** tab.
2. Click Path. The directory tree and input field (with history) appears to the lower right of the window.



3. Browse to the DICOMDIR file in the directory tree or write its path directly to the input field. These two elements are updating each other - browsing in the directory tree causes input field value change, the currently selected path appears there which can be edited. Writing the path to the input field causes directory tree expand to the selected path (if it's valid). Input field has higher priority.
4. Follow steps 3 through 6 of Section 6.3.1.2 to complete the search.

**To view a DICOMDIR Exam (choose one of the following options):**

- Select a study from the list and double-click it to view it automatically
- Select a study from the list and click View

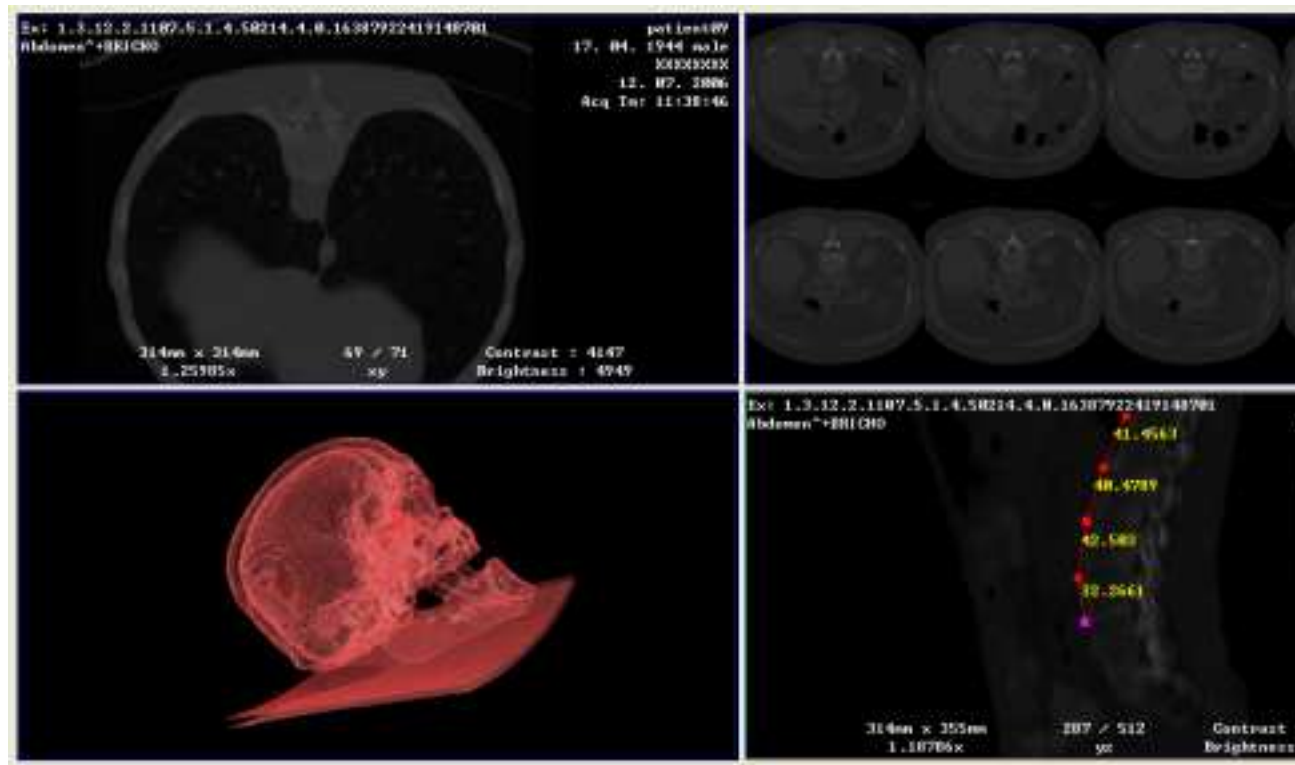
These studies will be available in the Recent DICOMDIR mode.

## 6.3.2. Viewing studies

Studies can be viewed using the procedures for the four exam tabs outlined in previous sections. This section provide a general reiteration of those procedures. In addition to learning how to view a study, this section will also show you how to adjust the screen layout and compare multiple studies.

**To view a study:**

- Select a study from the list and double-click it to view it automatically, or select a study from the list and click View. The study appears in the main window (in the previously selected viewer), and the toolbar is updated according to the viewer you are using.



Images appear side-by-side in a grid (default setting = 1x2), like the films mounted beside each other on a light box. This grid configuration can be adjusted by following the procedure outlined in Section 6.3.2.1.


### 6.3.2.1. Adjusting the screen layout

Images that appear on the screen are laid out in a side-by-side (1x2) grid configuration by default. This configuration can be adjusted to suit your preferences.

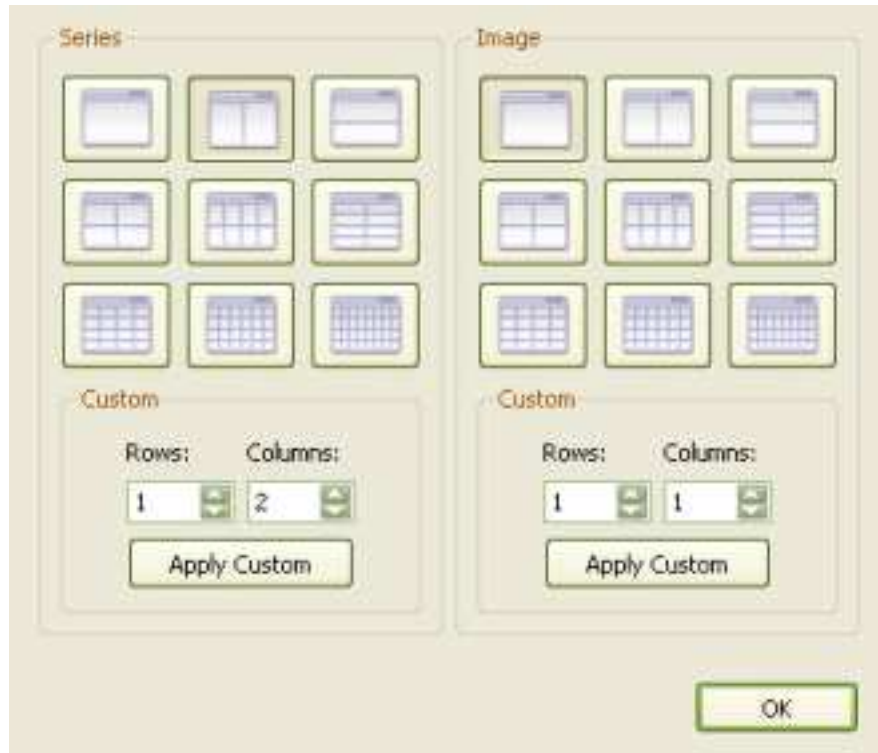
**To adjust the screen layout:**

1.

**Choose one of the following options to access the Screen Layout dialog box:**

- Tools → Screen Layout (Ctrl-C)
- Click 

The Screen Layout dialog box appears.




The Series layout determines the format of the panes in the window. Each pane can contain one series. The Image layout determines the format of the images within the active series.

2. Select a layout for the series/image, or define the values for rows and columns, and click **Apply**.
3. Click **OK** to close the Screen Layout dialog box.

### 6.3.2.2. Comparing multiple studies

Multiple studies can be compared by selecting additional studies while viewing a study.

**To select an additional study for comparison with the current study being viewed:**

1. Select a viewer where you want to appear the study (border around the selected viewer turns green).
2. While viewing the current study, search for the additional study by clicking .
3. Conduct the search (see Section 6.3.1.2).
4. Select the required study from the list and click **View**.
5. Change the screen layout as required (see Section 6.3.2.1).

## 6.4. Manipulating Images

This section covers manipulating image display functionality, such as orientation, magnification, field of view, etc.

### 6.4.1. Adjusting window/level settings

Window leveling allows you to adjust the brightness and contrast of images.


#### 6.4.1.1. Adjusting brightness

The level setting controls the brightness of an image.

**To adjust the brightness of an image:**

1.

**Choose one of the following options:**

- Tools → Window/Level (**Ctrl-W**)
- Click 

From here, you can adjust the brightness of the selected image.

2. Position the cursor over the image to be adjusted, and right-click and drag the cursor up or down over the image.
3. Release the mouse button to apply the new values to all images within the series. These values are displayed on the lower right corner of each image.


#### 6.4.1.2. Adjusting contrast

The window setting controls the contrast of an image.

**To adjust the contrast of an image:**

1.

**Choose one of the following options:**

- Tools → Window/Level (**Ctrl-W**)
- Click 

From here, you can adjust the contrast of the selected image.

2. Position the cursor over the image to be adjusted, and right-click and drag the cursor left or right over the image.
3. Release the mouse button to apply the new values to all images within the series. These values are displayed on the lower right corner of each image.

## **6.4.2. Changing image/slice orientation**

The following two procedures can be used to change image orientation and one for slice orientation.

## **6.4.3. Adjusting image viewing options**

.....

## **6.4.4. Clearing measurements**

.....

# **6.5. Measuring Images**

.....

## **6.5.1. Overlaying text**

.....

## **6.5.2. Making measurements**

.....



### **6.5.3. Probing images**

.....

### **6.5.4. Clearing measurements**

.....