MITK Developer Guide

Bounias Dimitrios, Ashish Singh
CBICA • University of Pennsylvania • April 2020

Overview

This guide assumes basic C++, CMake, and ITK knowledge.

- Introduction
- MITK concepts
- Building MITK
- Modules
- Command line programs
- Plugins
- Working with MITK
- Customization

Introduction

Introduction

Current stable version

2018.04.2

Resources

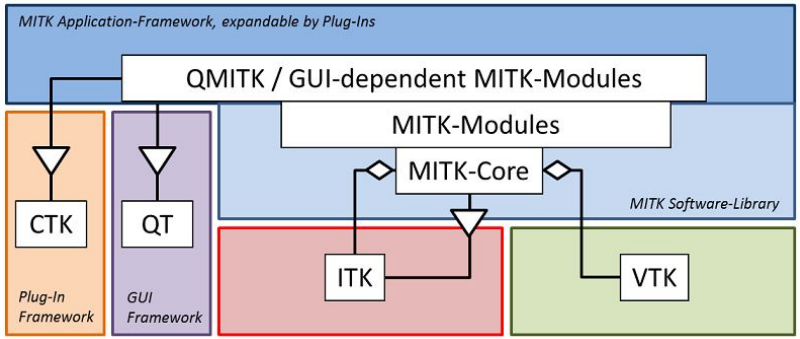
[Developer Manual]

[Developer Tutorials]

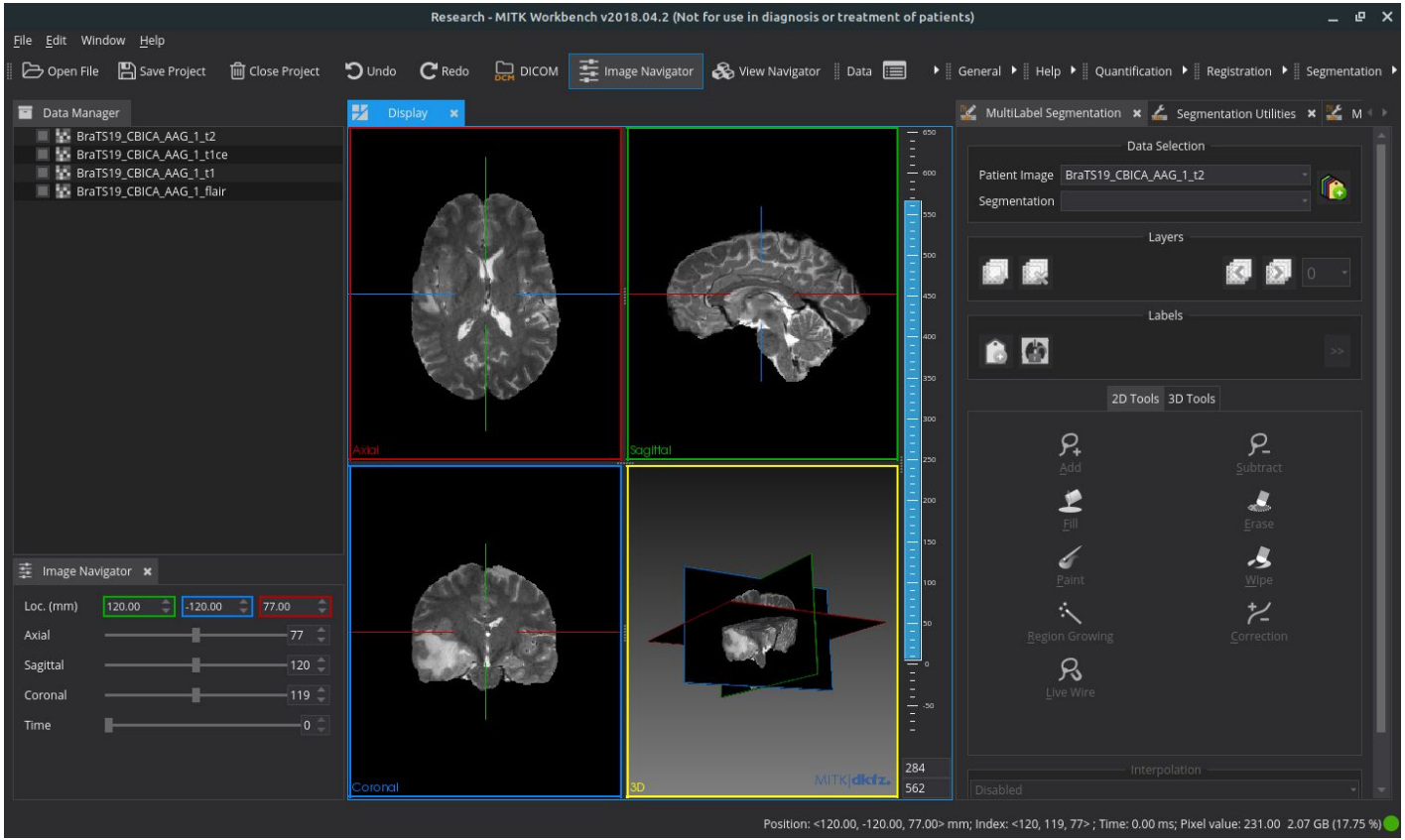
[Github Repository]

[Mailing List Archive]

Architecture Diagram



MITK Workbench



MITK Workbench

- The default MITK application.
- The main interface elements are: the *Data Manager*, the *Display*, and the *Action Bar* (all explained further in the MITK Concepts section).
- *Display* is the main UI element (i.e., the *editor*). Every other *view* can be drag-and-dropped anywhere around the *editor*, be tabbed with other views, or be fully detached.
- Images can be drag-and-dropped to the *Data Manager*, or be loaded through the file menu.

MITK Workbench

- *Plugins* can be launched from the *Action Bar* at the top.
- Themes and more can be changed through the preferences.
- Documentation for various plugins and the UI is included in *Help>Open Help Perspective*. To go back click *Window>Open Perspective>Research*.

MITK Concepts

High Level Terms

1. Modules

- Typically reusable elements. Usually non-UI.
- Meant to be used by other modules/plugins/cmd-applications.

2. Plugins

- Typically non-reusable UI elements and interactions. *Plugins* can contain functionality, but they usually just invoke *modules* for that.
- Usually create an entry in the *Action Bar* of the application and provide a UI with functionality. Additionally, plugins can serve other purposes.

High Level Terms

3. Applications

- An MITK *application* typically doesn't hold much source code. It just defines everything needed to create a custom application, like its name, which *plugins* to include in the interface, and more.
- MITK Workbench is an MITK *application*.

4. Project

- A structured directory where all your code resides, outside of MITK sources.

Project: More details

- When developing with MITK, all your code goes into your *project* and you are not supposed to interact with the MITK repository.
- Your project (i.e., your repository) should be structured. It typically has the following directories: Modules/, Plugins/, Applications/, CMake/, CMakeExternals/. All of these directories are optional.

Modules: More details

- MITK libraries that can be used by other modules/plugins/cmd-apps.
- **Built by default if something requires them** and don't need to be turned on/off.
- Offer a separation between UI and processing.

Plugins: More details

- Structured CTK plugins.
- Usually provide views the user can interact with.
- Can also define context-menu entries, themes, and more.
- Can be turned on/off through CMake, or “cherry picked” in the definition of an *application*.

Images

MITK uses *mitk::Image* to represent medical images. Images, and other similar data structures, all inherit from mitk::BaseData. Internally, mitk::Image uses the same memory structures as ITK, but it offers some additional benefits.

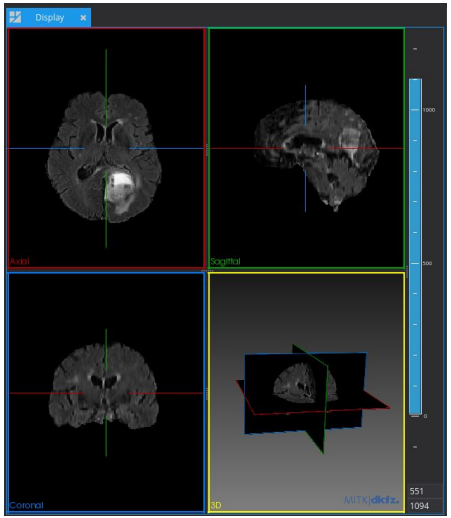
Benefits

- mitk::Image is not templated, but can be n-dimensional.
- A lot of convenience methods like GetStatistics().
- Can be transformed to itk::Image and back without the need to allocate new memory for the data.
- mitk::LabelSetImage offers *a lot* of functionality for segmentations.
- *Properties* can hold a lot of additional information.

Node Properties

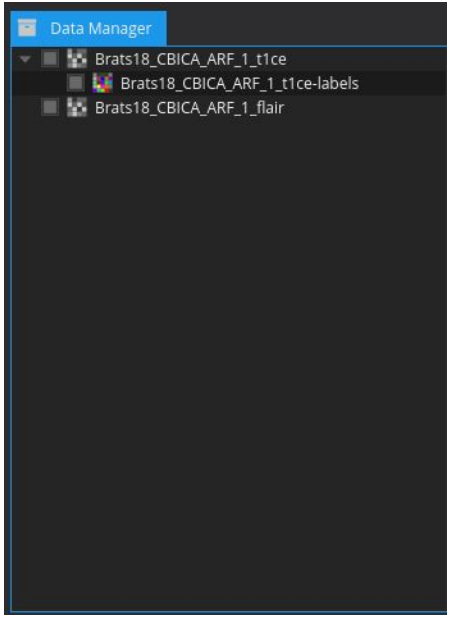
- Hold information about the data.
- Can be information like DICOM headers.
- Can also be custom properties. For instance if you want to know that an image originated from your plugin you can give it a boolean property with the name of the plugin.
- Easily accessible through both the UI and code.

The Display



- 4 render windows: Axial, Saggittal, Coronal, and 3D.
- Provided by the org.mitk.gui.qt.stdmultiwidgeteditor plugin.
- *Editor* is the main UI element and everything else is positioned around it. In MITK Workbench, the *Display* is the editor.
- You can potentially choose a different editor for your application, like a simpler render window or something completely different.

Data Manager (and Data Storage)



- *Data Manager* is the plugin that displays the loaded images.
- Internally, each image (or segmentation, surface, etc) directly corresponds to a node in *mitk::DataStorage*.
- *mitk::DataStorage* is a singleton class and is always available to the plugins and modules.
- The structure is hierarchical and images can have their related images as “parents”. For instance, a denoised image could be the “child” of the original image.

The Action Bar

- A set of buttons that perform actions.
- Provides basic actions, like opening files and showing the data manager.
- Each plugin that provides a view also shows up here. If the action belongs to a category, then it is grouped (like the segmentation plugins in the picture).
- New categories can be added.

Perspectives

- Perspectives provide a selection of plugins and settings which synergize well in order to solve certain problems.
- This can include placement of views as well in order to create an environment suited to solve the problem at hand.
- In MITK Workbench, to switch perspectives use “Window”>”Open Perspective”.

The “build” and the “superbuild”

- Two different compilation targets.
- The *superbuild* encapsulated the *build* and always builds it.
- The *superbuild* takes cares of dependencies and setting up the *build*.
- The *build* takes care of building and enabling plugins, applications, tests, and creates the packages.
- Typically you invoke the *superbuild* the first time and afterwards only to enable additional dependencies.
- The *build* resides in the MITK-build/ directory of the *superbuild*.

The *superbuild* directory and its contained **MITK-build/** (*build*) directory can both be opened with cmake and be built.

Building the *superbuild* is slow, don't do it for no reason.

Building MITK

Prerequisites

- 64bit system, it won't work on 32bit.
- git 2.21.0.windows.1 (version not important unless the interface changes)
- cmake 3.12.2
- MITK v2018.04.2, more instructions on this below
- Qt 5.11.1: Download and install using open-source license. Look for it in the archive when installing. We need the following components: "Desktop gcc (or msvc2017) 64-bit", "Qt WebEngine", and "Qt Script". On Windows install in short path, like C:\Qt. On linux you can install it wherever you want.

Projects

- **You can build MITK without any extension projects.**
- If you want to experiment with building a project before developing your own, consider using the MITK Project Template https://github.com/MITK/MITK-ProjectTemplate
- You can also consider experimenting with *CBICA/InteractiveSegmentation* which offers a very detailed guide on how to build it https://github.com/CBICA/InteractiveSegmentation/blob/master/docs/BUILD_INSTRUCTIONS.md

Using the stable version of MITK

```
git clone https://github.com/MITK/MITK
cd MITK
git fetch --all --tags
git checkout v2018.04.2 -b v2018.04.2-branch
```

There are two relevant repositories. MITK which you are not supposed to edit, and your project, where all your code resides.

You are always building the MITK sources. Your project doesn't stand on its own. It is simply an extension to the MITK compilation.

Configuring the superbuild

Using CMake, with the MITK repo as the “sources”:

- Set Qt5_DIR to point to the “lib/cmake/Qt5” dir of your Qt installation.
- Set "CMAKE_INSTALL_PREFIX" to a non-root dir.
- Set "MITK_USE_OpenCV" to true and other MITK dependencies you might want to use.
- **Set "MITK_EXTENSION_DIRS” to your own MITK *project*.**
- Under linux, use CMAKE_BUILD_TYPE to set Debug/Release.

Configuring the build

Using CMake, with the MITK repo as the “sources”:

- When the *superbuild* is built there would be an MITK-build/ directory inside the superbuild directory. Opening that with CMake allows you to configure the *build*.
- MITK_BUILD_* variables allow you to turn plugins on/off. MITK_BUILD_APP_* variables do the same for applications.

Building

Windows

"Open Project" through cmake allows you to "Build Solution" for both the *superbuild* and the *build*.

You can also do batch build from Visual Studio that will build both Debug and Release versions simultaneously.

Linux

Run "make -j4" inside the *superbuild* and the *build* directory.

The debug and release superbuilds should go in different directories.

Running the generated binaries

Windows

You can run the program by running the bat files in MITK-build\bin\. Because the dependencies are not in the path, use the scripts provide. For example:

- startMitkWorkbench_release.bat starts the program in Release mode.
- startMitkWorkbench_debug.bat starts the program in Debug mode.

Linux

Your binaries reside in MITK-build/bin/ and don't require anything to be in the path.

Packaging

Windows

- Remember to have NSIS 2.51 installed (not 3.x!).
- Build the PACKAGE project in Visual Studio (in the *build* solution).
- This will create a zip package and a NSIS installer.

Linux

- Simply run “make package” in the *build* directory, MITK-build/.
- It will create a _CPack_Packages/ directory that contains the package.
- Package is a tarball (tar.gz).

Modules

Modules

- Modules/ is a top-level directory in your project. It contains one directory for each module and a file ModuleList.cmake (explained in the next slide).
- Modules are built automatically when needed. They can't be turned on or off.

ModuleList.cmake

- Typically simply contains the list of modules (dependencies go first!):

```
set(MITK_MODULES
    YourModule1
    YourModule2)
```

- By default, MITK adds “Mitk” as a prefix to all modules. The above modules will be available as “MitkYourModule1” etc.
- To change that behavior, add the following to the top of the file. The prefix can be empty, “”.

```
set(MITK_MODULE_NAME_PREFIX "MyPrefix")
```

Module structure

- Each module contains: include/ for headers, src/ for source files, and optionally cmdapps/ for creating command line programs that use the module. Even though cmdapps/ resides inside the module it is not part of the same compilation unit. The module is a CMake dependency of the cmd-app and the cmd-app can be turned on/off. More on this in the “Command line programs” section.
- Additionally, each module contains CMakeLists.txt and files.cmake
- Examples can be found in the MITK Project Template (https://github.com/MITK/MITK-ProjectTemplate).

CMakeLists.txt of modules

- Example:

```
mitk_create_module(
  PACKAGE_DEPENDS ITK VTK Qt5|Core+Widgets
  DEPENDS PUBLIC MitkYourModule1 MitkYourModule2
)
add_subdirectory(cmdapps) # if there is a cmd-app
```

- PACKAGE_DEPENDS is for external dependencies.
- DEPENDS is for mitk module dependencies.

files.cmake of modules

```
set(CPP_FILES
 Module1.cpp
 Module1Helper.cpp
)

set(H_FILES
 include/Module1.h
 include/Module1Helper.h
)
```

- Notice that src/ is not used for cpp files, even though they reside there.

Exports

Module classes and functions need to be exported for compilation reasons.

For example, for a module named MyModule with the default Mitk prefix, you need to include this (it gets automatically generated):

```
#include <MitkMyModuleExports.h>
```

Your classes/functions definitions should look like this

- class MITKMYMODULE_EXPORT CaPTkSurvival
- MITKMYMODULE_EXPORT void functionName()

Command line programs

Cmd apps

- A directory inside a module, called cmdapps/.
- add_subdirectory(cmdapps) should be in the CMakeLists.txt of the module.
- cmdapps/ contains a CMakeLists.txt and the source files for the cmdapp(s).
- cmdapps/ directories are inside modules, but cmd apps are not part of the same compilation unit and have the module as a dependency. Unlike modules, cmd apps can be turned on/off through CMake.
- The defined cmd apps become binaries in the MITK-build/bin/ directory.

CMakeLists.txt for cmd apps

```
option(BUILD_FooCmdApp "Build command-line app for foo" ON)

if(BUILD_FooCmdApp)
 mitkFunctionCreateCommandLineApp(
   NAME FooCmdApp # target name
   CPP_FILES FooCmdApp.cpp
   PACKAGE_DEPENDS Qt5|Core+Widgets
   DEPENDS MitkMyModule
 )
endif()
```

- PACKAGE_DEPENDS for external dependencies,
- DEPENDS for module dependencies.

Source file for cmd app

- Example: https://github.com/MITK/MITK-ProjectTemplate/blob/master/Modules/ExampleModule/cmdapps/ExampleCmdApp.cpp
- Parsing through: `#include <mitkCommandLineParser.h>`

Parser initialization for cmd app

```
mitkCommandLineParser parser;

/**** Set general information about the command line app ****/
parser.setCategory("MyProject Cmd App Category");
parser.setTitle("My Foo Cmd App");
parser.setContributor("CBICA");
parser.setDescription(
        "This command line app accepts a directory of subjects and produces either a"
        " newly-trained model or a foo prediction result "
        " depending on the provided parameters.");

parser.setArgumentPrefix("--", "-");
```

Arguments for cmd app's parser

```
parser.addArgument(
        "input", # -- long arg name
        "i", # - short arg name
        mitkCommandLineParser::InputFile, # type
        "Input file", # short description
        "Path to the desired input file where the model resides", # long description
        us::Any(), # should always be here
        false); # false -> required arg, true -> optional arg
```

- Type can be: String, Bool, StringList, Int, Float, InputDirectory, InputFile, InputImage, OutputDirectory, OutputFile
- After all args are defined, parse with:

```
std::map<std::string, us::Any> parsedArgs = parser.parseArguments(argc, argv, &parseSuccess);
```

Handling required args

```
if (parsedArgs["input"].Empty() || parsedArgs["output"].Empty())
{
  MITK_INFO << parser.helpText();
  return EXIT_FAILURE;
}
```

- MITK_INFO is from #include <mitkLogMacros.h>

Parsing args

- Value:

```
std::string model = “model1”;
if (parser.argumentParsed("model"))
{
    model = parsedArgs["model"].ToString();
}
```

- Boolean:

```
if (parser.argumentParsed("train"))
{
    trainNewModel = true;
}
```

Plugins

Plugins

- Plugins have names that contain words separated by dots. For example, “org.mitk.gui.qt.dicom”.
- Plugins/ is a top-level directory in your project. It contains one directory for each plugin and a file PluginList.cmake (explained in the next slide).
- Plugins can be enabled/disabled through CMake and the option is generated automatically.

PluginList.cmake

- Typically simply contains the list of modules (dependencies go first!). ON/OFF defines whether to enable them in cmake by default:

```
set(MITK_PLUGINS
  upenn.cbica.myproject.brain.foo:OFF
  upenn.cbica.myproject.lung.bar:ON)
```

- By default, MITK requires “org.mitk” as a prefix to all modules.
- To change that behavior, add the following to the top of the file.

```
### Add upenn.cbica.* to the list of allowed naming schemes
list(APPEND MITK_PLUGIN_REGEX_LIST "^upenn_cbica_[a-zA-Z0-9_]+$")
```

CMakeLists.txt for plugins

- Example (notice the underscores instead of dots)

```
project(upenn_cbica_myproject_brain_foo)

mitk_create_plugin(
  PACKAGE_DEPENDS ITK OpenCV
  MODULE_DEPENDS MitkMyModule
 )
```

- PACKAGE_DEPENDS is for external dependencies.
- MODULE_DEPENDS is for mitk module dependencies.

Plugins structure

- Each module contains: src/ for source files and optionally: resources/ for non-programming files, and documentation/ for creating doxygen documentation for the plugin, that will also show under Help in the application.
- Additionally, each plugin contains CMakeLists.txt, files.cmake, manifest_headers.cmake, and plugin.xml.
- Examples can be found in the MITK Project Template (https://github.com/MITK/MITK-ProjectTemplate).

files.cmake of plugins

```
set(SRC_CPP_FILES
)
set(INTERNAL_CPP_FILES
 mitkPluginActivator.cpp
 QmitkCaPTkFooView.cpp
)
set(UI_FILES
 src/internal/QmitkCaPTkFooControls.ui
)
set(MOC_H_FILES
 src/internal/mitkPluginActivator.h
 src/internal/QmitkCaPTkFooView.h
)
```

```
set(CACHED_RESOURCE_FILES
 resources/icon.svg
 plugin.xml
)
set(QRC_FILES
  resources/resources.qrc
)
### Conveniently adding src/ and src/internal/
set(CPP_FILES)
foreach(file ${SRC_CPP_FILES})
 set(CPP_FILES ${CPP_FILES} src/${file})
endforeach(file ${SRC_CPP_FILES})
foreach(file ${INTERNAL_CPP_FILES})
 set(CPP_FILES ${CPP_FILES} src/internal/${file})
endforeach(file ${INTERNAL_CPP_FILES})
```

Plugin-specific files

manifest_headers.cmake

```
set(Plugin-Name "MyProject Brain Foo")
set(Plugin-Version "1.0.0")
set(Plugin-Vendor "CBICA")
set(Plugin-ContactAddress
"https://www.med.upenn.edu/cbica/")
set(Require-Plugin org.mitk.gui.qt.common
org.mitk.gui.qt.datamanager)
```

- General information about the plugin.
- Require-Plugin defines plugins that are required for this plugin to be operational. For instance, a plugin that expects images to be loaded into MITK doesn't make sense without the data manager plugin.

Plugin-specific files

plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
 <extension point="org.blueberry.ui.views">
  <view id="upenn.cbica.myproject.brain.foo"
    name="Foo"
    category="Brain"
    icon="resources/icon.svg"
    class="QmitkMyProjectBrainFooView" />
 </extension>
 </plugin>
```

- This will create an entry in the *Action Bar* under the “Brain” category.
- When clicked it will open a side tab with the title “Foo” that will contain what is defined in class QmitkCaPTkBrainFooView.
- The Qt class should look like:

```
class QmitkMyProjctBrainFooView :
public QmitkAbstractView, public
mitk::ILifecycleAwarePart
```

Working with MITK

MITK naming conventions

Modules

- MITK classes and functions are namespaced with *mitk*. Your code could have its own namespace.
- One class/function per file
- The name of the file follows the class/function.
- For example, mitk::Image in mitkImage.h
- Headers in include/, source files in src/.

Plugins

- All source files in src/. If the file is not meant to be visible to anything else it goes in src/internal/.
- If the file contains a Qt class, its name starts with *Qmitk* and is **not** namespaced with *mitk*. For example, QmitkStdMultiWidgetEditor.h. Your code could have its own prefix.

Working with MITK Images

Loading an MITK Image

```
auto image = mitk::IOUtil::Load<mitk::Image>(imagePath);
```

Saving an MITK Image

```
mitk::IOUtil::Save(image, imagePath);
```

Cloning an MITK Image

In order to clone an image, you can simply call the inherited method Clone(). It returns an itk::SmartPointer and works also with const image pointers.

```
mitk::Image::Pointer image2 = image1->Clone();
```

Working with MITK Images

- mitk::BaseData is the base class for all data objects. Thus, mitk::Image inherits from it.
- Since multiple modules/plugins might want to interact with the same image, there is the concept of the image accessors; locks that make sure that only one operation is happening on the data each time.

That is relevant only if your code transforms images, rather than using images as input to produce output. More info http://docs.mitk.org/nightly/MitkImagePage.html.

- mitk::LabelSetImage is for segmentations and inherits from mitk::Image.

Label Set Images (Segmentations)

- Convert from mitk::Image to mitk::LabelSetImage:

```
mitk::LabelSetImage::Pointer imLabels =
        mitk::LabelSetImage::New();
imLabels->InitializeByLabeledImage(image);
```

- mitk::Label represents a label and has a name, value, and color.
- mitk::LabelSet represents a set of mitk::Label objects.

- An mitk::LabelSetImage can have multiple label sets, each in a *layer*. Usually though, this complicates things so it's better to use only one layer and retrieve it through mitk::LabelSetImage::GetActive LabelSet().
- mitk::LabelSetImage and mitk::LabelSet support a lot of convenience methods for working with segmentations.

Iterating the labels of a LabelSetImage

```
mitk::LabelSet::Pointer labelSet = segmentation->GetActiveLabelSet();
mitk::LabelSet::LabelContainerConstIteratorType it;
for (it = labelSet->IteratorConstBegin();
     it != labelSet->IteratorConstEnd();
     ++it)
{
    if (it->second->GetValue() != 0)
    {
        values.push_back(it->second->GetValue());
        labels.push_back(it->second->GetName());
    }
}
```

Nodes and mitk::DataStorage

- mitk::DataNode objects are used in the mitk::DataStorage to hold *data* and their name, alongside properties (meta-information) about the data.
- Nodes can have parent/children nodes.
- Data storage has Add() and Remove() for nodes

```
m_DataStorage->Add(node);
```

- Adding a node and settings its parent

```
m_DataStorage->Add(node, parentNode);
```

An example of a node with a child:

Node properties

- Properties hold additional non-data information about the data and belong to the node.
- There are a lot of types supported, like strings, integers, etc.

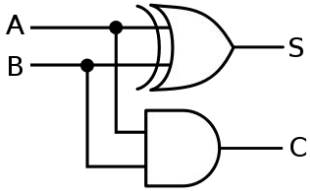
```
//1: Read a property
mitk::Property::Pointer readProperty =
       node->GetProperty("color"); //read
       the property "color" from the data
       node

//2: Write a property
mitk::BoolProperty::Pointer newProperty =
       mitk::BoolProperty::New( true );
       //create a new property, in this
       case a boolean property
node->SetProperty( "IsTensorVolume",
       newProperty ); //add this property
       to node with the key
       "IsTensorVolume"
```

http://docs.mitk.org/nightly/PropertiesPage.html

Iterating the nodes of Data Storage

```
mitk::DataStorage::SetOfObjects::ConstPointer all =
    GetDataStorage()->GetAll();
for (mitk::DataStorage::SetOfObjects::ConstIterator it = all->Begin();
     it != all->End(); ++it)
{
  if (it->Value().IsNotNull())
  {
      std::cout << "Name of node: "
                << it->Value()->GetName()
                << "\n";
      images.push_back( dynamic_cast<mitk::Image*>(it->Value()->GetData()) );
  }
}
```



Predicates

- *Predicates* are a mechanism to construct **logical expressions** to filter the existing data. For example, getting only the data which are segmentation images, or having a combo box that only includes 3D images as options.
- On the right, we filter the data storage for data that is (“mitk::Image”) **and** (**not** “helper object”).

```
auto predicateIsImage =
  mitk::TNodePredicateDataType<mitk::Image>::New();
auto predicatePropertyHelper =
  mitk::NodePredicateProperty::New("helper object");
auto predicatePropertyNotHelper =
  mitk::NodePredicateNot::New(predicatePropertyHelper)
auto predicateFinal = mitk::NodePredicateAnd::New();
predicateFinal->AddPredicate(predicateIsImage);
predicateFinal->AddPredicate(predicatePropertyNotHelper);

// Get an iterable set of the objects that fit our final predicate
mitk::DataStorage::SetOfObjects::ConstPointer nodesOfPredicate =
   GetDataStorage()->GetSubset(predicateFinal);
```

Adaptors

Adaptors are functions and classes for interfacing (mostly) with ITK.

- Conversion and casting between mitk::Image and itk::Image are done by *adaptors* (more in the next slide).
- Important adaptors are also the AccessByItk family of adaptors, particularly AccessByItk_n() (more in two slides). They allow you to use ITK images from MITK images without ever doing any casting/conversion. To do that, a function that uses the itk::Image has to be constructed in a particular way and then called through an AccessByItk *adaptor*.

Converting images from/to ITK

__Note:__ Converting, especially from MITK to ITK, expects that the pixel type would be compatible between what is stored in the memory structure of the source and target. Casting, while slower, is typically safer.

ITK to MITK

Casting

```
void mitk::CastToMitkImage<...>(
        itk::Image<...>, mitkImage);
```

Converting

```
mitk::Image
mitk::ImportItkImage(itk::Image<...>)
```

MITK to ITK

Casting

```
void mitk::CastToItkImage(mitkImage,
                          itk::Image<...>)
```

Converting

```
using ImageToItkType =
    mitk::ImageToItk<ImageType>;
typename ImageToItkType::Pointer imagetoitk =
    ImageToItkType::New();
imagetoitk->SetInput(mitkImage);
imagetoitk->Update();
typename ImageType::Pointer itkImage =
    imagetoitk->GetOutput();
```

AccessByItk_n

- The target function has to be templated for pixel type and dimension. The first parameter has to be an itk::Image raw pointer.

```
template <class TPixel, unsigned int VDim>
static void Run(
    itk::Image<TPixel,VDim>* inputItkImage,
    int maximum,
    mitk::Image::Pointer& outputImage);
```

- If there are no extra arguments, AccessByItk(inputImage, functionName) can be used instead.

- The output image, if one is needed, has to be mitk::Image; only the first is converted automatically. Conversions from itk to mitk are easier as there is no template overhead.
- Called by:

```
AccessByItk_n(input, Run, (maximum, output));
```

- The first parameter is the input mitk::Image (which is automatically converted to itk). The second is the name of the function to call. The third is a tuple of additional arguments.

Resources

If resources are needed, they should be placed in the resources/ directory of the plugin.

QRC File

Inside the resources/ directory there should also be a .qrc file. The QRC file should also be defined in files.cmake.

Example QRC File

```
<RCC>
  <qresource prefix="/org_mitk_my_plugin>
    <file>icon1.png</file>
    <file>icon2.png</file>
  </qresource>
</RCC>
```

To access resources use the colon punctuation mark, followed by the prefix and resource name.

```
QIcon(":/org_mitk_my_plugin/icon1.png"))
```

External dependencies

- External dependencies can be used for linking third-party source code and data that should not be part of the repository.
- The process works by creating CMake files in CMake/ and CMakeExternals/.
- The integration of “GuidelinesSupportLibrary” of the MITK Project Template can be used as an example. Particularly, the files CMakeExternals/GuidelinesSupportLibrary.cmake, CMake/FindGuidelinesSupportLibrary.cmake, and CMake/PackageDepends/MITK_GuidelinesSupportLibrary_Config.cmake.
- Each dependency has to be added in CMakeExternals/ExternalProjectList.cmake, e.g.,

```
mitkFunctionAddExternalProject(NAME GuidelinesSupportLibrary ON DOC "Use Microsoft's implementation of the Guidelines Support Library (GSL)")
```

External dependencies can be enabled/disabled, and built only in the *superbuild*.

External dependencies: Data

- Add

  mitkFunctionAddExternalProject(NAME CaPTkData OFF ADVANCED)

  to CMakeExternals/ExternalProjectList.cmake.

- Also use the file on the right and the file in the next slide.
- The data have to reside somewhere on the internet and not require credentials to access.

CMake/FindMyData.cmake

```
set(MyData_DIR
  ${MITK_SUPERBUILD_BINARY_DIR}/ep/src/MyData)
if(EXISTS ${CaPTkData_DIR})
 set(MyData_FOUND TRUE)
else()
 set(MyData_FOUND FALSE)
endif()
```

External dependencies: Data

CMakeExternals/MyData.cmake

```
if(MITK_USE_MyData)
if(DEFINED MyData_DIR AND NOT EXISTS ${MyData_DIR})
 message(FATAL_ERROR "MyData_DIR variable is defined but
corresponds to non-existing directory")
endif()

set(proj MyData)
set(proj_DEPENDENCIES)
set(MyData_DEPENDS ${proj})

set(MyData_DIR ${ep_prefix}/src/${proj}/)
 ExternalProject_Add(${proj}
   URL https://MyFileHosting.com/MyData.tar.gz
   UPDATE_COMMAND ""
   CONFIGURE_COMMAND ""
   BUILD_COMMAND ""
   INSTALL_COMMAND ""
   DEPENDS ${proj_DEPENDENCIES})
else()
 mitkMacroEmptyExternalProject(${proj} "${proj_DEPENDENCIES}")
endif()
endif()
```

Context-menu actions

- These show up when you right click an image in the data manager.
- To define in the plugin.xml (this will show up only for *LabelSetImage* entries):

```
<extension point="org.mitk.gui.qt.datamanager.contextMenuActions">
  <contextMenuAction nodeDescriptorName="LabelSetImage"
label="Perform action foo" icon="" class="QmitkMyProjectFooAction" />
</extension>
```

- In mitkPluginActivator.cpp:

```
void PluginActivator::start(ctkPluginContext *context) {
  BERRY_REGISTER_EXTENSION_CLASS(
      QmitkMyProjectFooAction, context)
   this->m_context = context;
}
```

- Example of the Qt class that performs the action in the next slide. Override the Run() method to add functionality. You probably won't need SetSmoothed(), SetDecimated(), and SetFunctionality().

Context-menu actions: Class structure

```
class MYPLUGIN_EXPORTS QmitkMyProjectFooAction :
      public QObject, public mitk::IContextMenuAction
{
 Q_OBJECT
 Q_INTERFACES(mitk::IContextMenuAction)

public:

 QmitkMyProjectFooAction();
 virtual ~QmitkMyProjectFooAction();

 //interface methods
 void Run( const QList<mitk::DataNode::Pointer>&
selectedNodes );
 void SetDataStorage(mitk::DataStorage* dataStorage);
 void SetSmoothed(bool smoothed);
 void SetDecimated(bool decimated);
 void SetFunctionality(berry::QtViewPart* functionality);
private:
 mitk::DataStorage* m_DataStorage;
};
```

Logging

From mitkLogMacros.h:

```
MITK_INFO << "My message";

MITK_INFO("MyCategory")("MySubCategory") << "Categorized message";

MITK_WARN << "This is a warning."

MITK_ERROR << "Error message."
```

http://docs.mitk.org/nightly/LoggingPage.html

Exception Handling

From mitkExceptionMacro.h:

Throwing

```
mitkThrow() << "This is an exception message";
```

Catching

```
catch (const mitk::Exception& e)
```

http://docs.mitk.org/nightly/ExceptionPage.html

Deploying files that are part of the repository

- In this example, we want to deploy a trained model that is in the form of a directory, my_module_model/. Specifically, inside our module, it is in resources/models/my_module_model/.
- We want my_module_model/ to exist inside bin/models/ both in our *build* **and** in the packaged version of our application.
- We want to deploy it as a directory under bin/models/ to potentially allow other modules to deploy there, too.

CMakeLists.txt of the module

```
[...]
# ---- Model deploying ----
# standard cmake way of copying files -- doesn't copy into the packaged install, but into the build directory
file(COPY resources/models/my_module_model
     DESTINATION ${CMAKE_BINARY_DIR}/bin/models)
# the below macro does the work necessary to install the models into the packaged executable
MITK_INSTALL( DIRECTORY resources/models)
# ------------------------
```

In code (could be called by the module or something else)

```
QString cbicaModelDir = QCoreApplication::applicationDirPath() +
                                    QString("/models/my_module_model");
```

Preferences

- Each plugin can have its own preferences, that show up in the application preferences as “nodes”. An example can be found in Plugins/org.mitk.gui.qt.multilabelsegmentation/ of the MITK repo.
- *Keywords* are for filtering the settings through search.
- Code instructions in the next slide.

In plugin.xml

```
<extension point="org.blueberry.ui.preferencePages">
  <page id="org.mitk.gui.qt.application.MultiLabelSegmentationPreferencePage"
name="MultiLabel Segmentation" class="QmitkMultiLabelSegmentationPreferencePage">
    <keywordreference
id="org.mitk.gui.qt.application.MultiLabelSegmentationPreferencePageKeywords"></keywo
rdreference>
  </page>
</extension>

<extension point="org.blueberry.ui.keywords">
  <keyword
id="org.mitk.gui.qt.application.MultiLabelSegmentationPreferencePageKeywords"
label="multi segmentation label multilabel multilabelsegmentation 2d display 3d outline
draw transparent overlay show volume rendering data node selection mode enable
auto-selection mode smoothed surface creation smoothing value decimation
rate"></keyword>
</extension>
```

Preferences: Blueberry code

- Your preferences class should inherit from: public QObject, public berry::IQtPreferencePage
- QObject because you need to use Qt widgets to represent the preferences and you will need slots to handle them.
- You will also need a berry::IPreferences::Pointer object:

```
berry::IPreferencesService* prefService
  = berry::Platform::GetPreferencesService();
m_SegmentationPreferencesNode =
prefService->GetSystemPreferences()->Node("/org.mitk.views.multilabelsegmentation");
```

- All your widgets should be part of a view, like m_MainControl in MITK's code, and that "main" is returned, to be shown, through GetQtControl().
- PerformOk() saves the preferences that were changed.
- In Update() you should update the widgets based on the stored preferences.
- Access/Set the stored preferences through the various methods of the preferences node, like GetBool("setting name", true), the second parameter is the default value.

Tests

- Tests reside in your *module*'s test/ directory.
- In your *module*'s CMakeLists.txt put:

```
if(MODULE_IS_ENABLED)
 add_subdirectory(test)
endif()
```

- In your test/files.cmake:

```
SET(MODULE_CUSTOM_TESTS
       MyModuleTestSuite.cpp)
```

- In your test/CMakeLists.txt:

```
MITK_CREATE_MODULE_TESTS()
mitkAddCustomModuleTest(
       MyModuleTestSuite
       MyModuleTestSuite)
### USE THIS FOR SLOW TESTS
#if(NOT MITK_FAST_TESTING)
#mitkAddCustomModuleTest(...)
#ENDIF()
```

- What goes in the source of a test suite is explained in the next slide.

Tests

- The test suite inherit from CppUnit::TestFixture.
- Inside the class:

```
CPPUNIT_TEST_SUITE(MyModuleTestSuite);

TESTMETHOD(Try2D);

TESTMETHOD(Try3D);

CPPUNIT_TEST_SUITE_END();
```

- You can override void setUp() for actions that happen before each test and void tearDown() for action afterwards.
- You should also implement whatever you define with the TESTMETHOD macro as a void member function. For example, here void Try2D() and void Try3D() should be defined and actually perform tests.
- Use `CPPUNIT_FAIL("Reason");` to manually fail a test.
- To fail based on a boolean expression:

```
CPPUNIT_ASSERT_MESSAGE(
   "Expected & Test images are not equal",
   Equal<ImageType>(outputImage,expectedImage)
);
```

Build configurations

- Can be used to enable/disable CMake flags automatically.
- Particularly useful for enabling/disabling plugins/applications/dependencies.
- Multiple configuration can be added as .cmake files under CMake/BuildConfigurations/.
- An example follows in the next slide.
- Note that unless you change configuration, changes you make to these variables in CMake will not work; the build configuration runs in every “configure” CMake step.

Build configuration example

- Enabling/disabling various dependencies, *applications*, *plugins*, as well as other CMake variables.

```
# Enable optional external dependencies
set(MITK_USE_OpenCV ON CACHE BOOL "" FORCE)
# Enable/disable applications (even if they don't show up in the superbuild!)
set(MITK_BUILD_APP_MyApp ON CACHE BOOL "Build MyApp" FORCE)
set(MITK_BUILD_APP_Workbench OFF CACHE BOOL "Build the MITK Workbench executable" FORCE)
# Enable/disable plugins (even if they don't show up in the superbuild!)
set(MITK_BUILD_org.mitk.gui.qt.multilabelsegmentation ON CACHE BOOL "Build the org.mitk.gui.qt.multilabelsegmentation Plugin." FORCE)
# Activate in-application help generation
set(MITK_DOXYGEN_GENERATE_QCH_FILES ON CACHE BOOL "Use doxygen to generate Qt compressed help files for MITK docs" FORCE)
set(BLUEBERRY_USE_QT_HELP ON CACHE BOOL "Enable support for integrating bundle documentation into Qt Help" FORCE)
# Disable console window
set(MITK_SHOW_CONSOLE_WINDOW OFF CACHE BOOL "Use this to enable or disable the console window when starting MITK GUI Applications" FORCE)
# Enable exporting of compile commands (useful for intellisense in vscode etc)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON CACHE BOOL "Enable/Disable output of compile commands during generation." FORCE)
```

Customization

Custom applications with custom UI

How MITK Workbench does it

- MITK Workbench is the default *application*, defined in the MITK sources. *Applications* define which functionality should be included.
- **MITK Workbench is configured to include all cmake-enabled plugins.** Alternatively, other applications can “cherry pick” whichever plugins they want.

How MITK Workbench does it

org.mitk.gui.qt.extapplication

The application (in Applications/) needs to point to a "product". The product is defined in this plugin.

Welcome screens, perspectives and more are also defined here.

org.mitk.gui.qt.ext

Controls, icons, and about page are defined in this plugin. Also, the action bar at the top and some default actions.

You can potentially copy these two plugins (with different names) and link your application to them. That way you can have something that looks like MITK Workbench and make edits there. You will need to change the relevant line in the "manifest_headers" of your extapplication to "set(Require-Plugin YOUR_EXT_PLUGIN_NAME)".

Creating your own *application*

- There are various files needed (CMakeLists.txt, etc) inside the directory of an *application*. You can copy and edit them from MITK Workbench.
- Since MITK Workbench is configured to pick up everything there is no definition of which plugins to include in its CMakeLists.txt, but there is a way to exclude some.

To “cherry pick” plugins don’t use exclusions. Instead, use:

```
set(_plugins MY_PLUGIN1 MY_PLUGIN2)
```

and

```
mitkFunctionCreateBlueBerryApplication(
        NAME YourAppName
        DESCRIPTION "What your App does"
        PLUGINS ${_plugins}
        ${_app_options})
```

Working with blueberry

- Blueberry is the framework used to create the UI. There is some migration though, in the future in might be pure CTK.

Important classes (to inherit from)

- **berry::IPerspectiveFactory**

  To define a perspective

- **berry::WorkbenchWindowAdvisory**

  Menu bar, tool bar and status bar are configured here, and the window title for is being set. The workbench window can be customized.

- **berry::QtWorkbenchAdvisor**

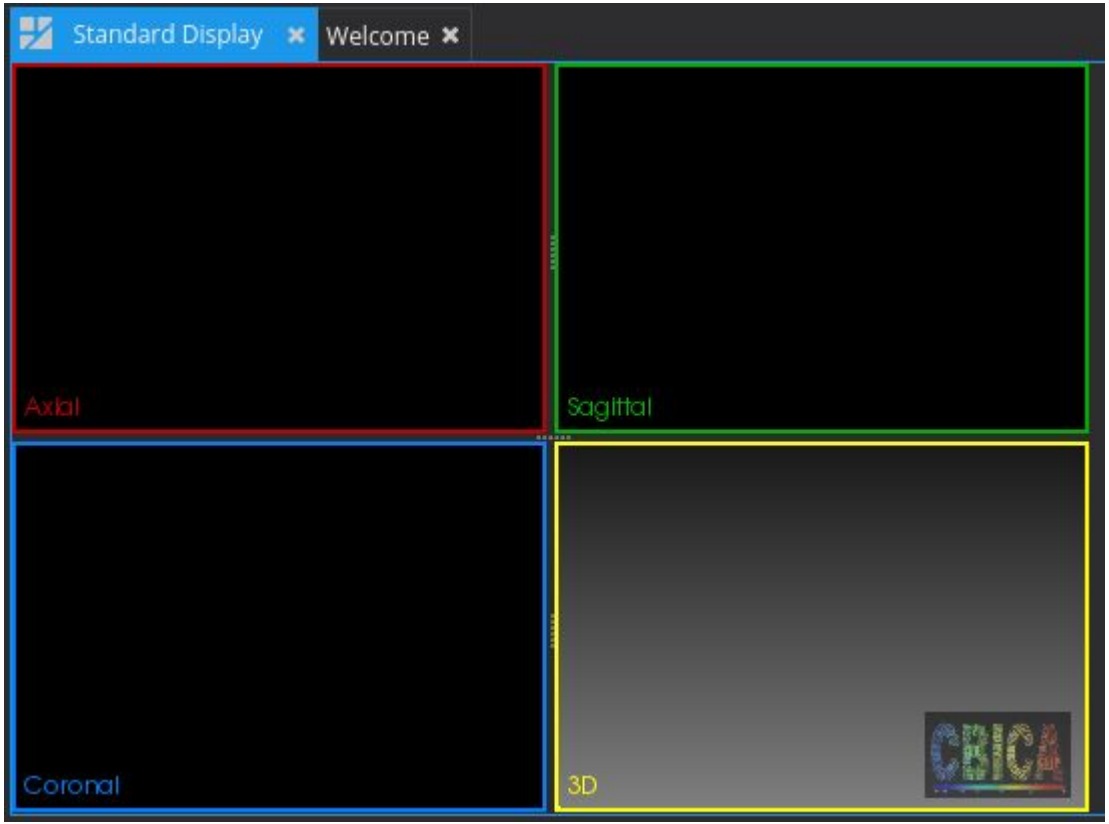
  Adds and sets a Qt-Stylesheet file to the berry::QtStyleManager during the initialization phase for customization purposes.

Custom logo

In your *extapplication* plugin, in the class that inherits from berry::AbstractUICTKPlugin, in the start() method:

```
mitk::WorkbenchUtil::SetDepartmentLogoPreference(":/my_plugin_resources/cbica-logo.png", context);
```

The logo should also be in the relevant qrc file.

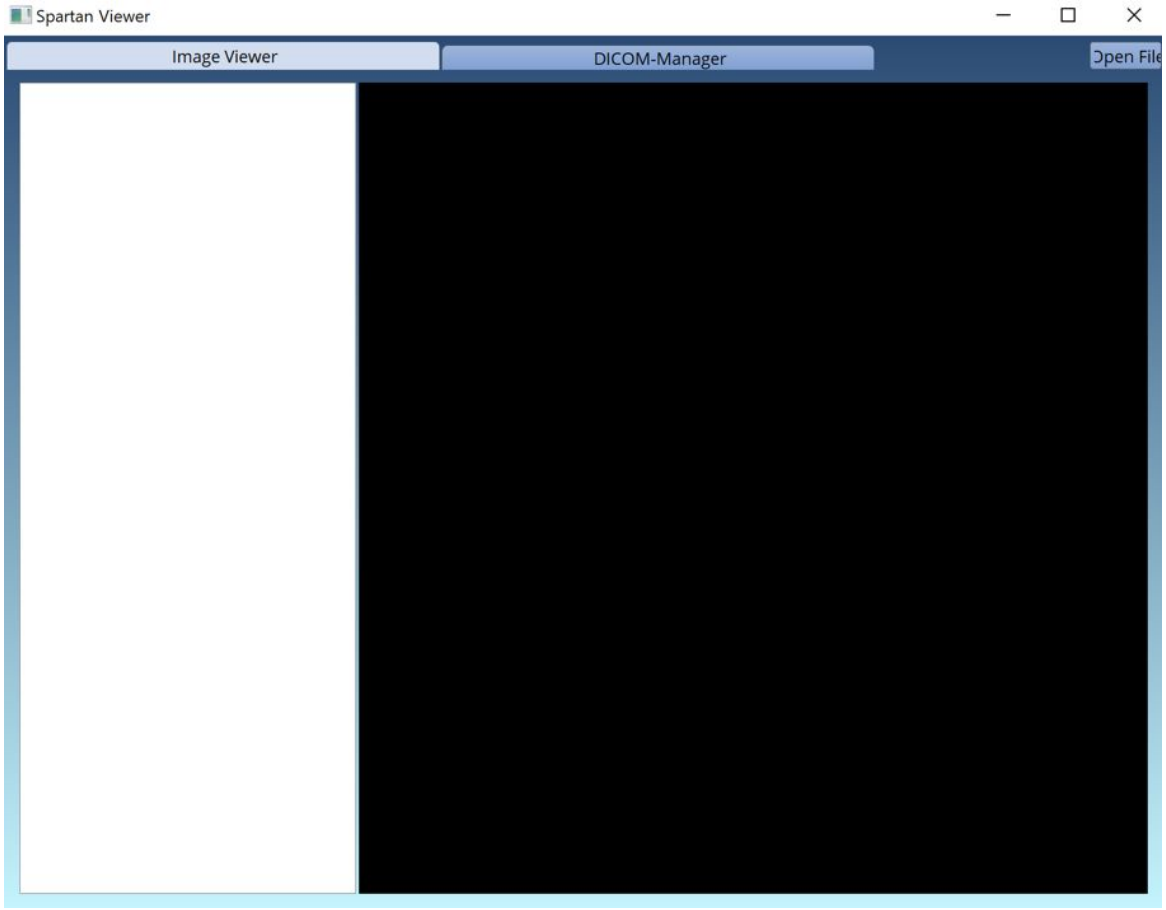


The UI of *customviewer*

In the MITK repository, the UI on the next slide can be achieved with the plugins (in Example/Plugins/):

org.mitk.example.gui.customviewer
org.mitk.example.gui.customviewer.views

- The tabs on the top switch between different perspectives.
- The user interface differs significantly from MITK Workbench.



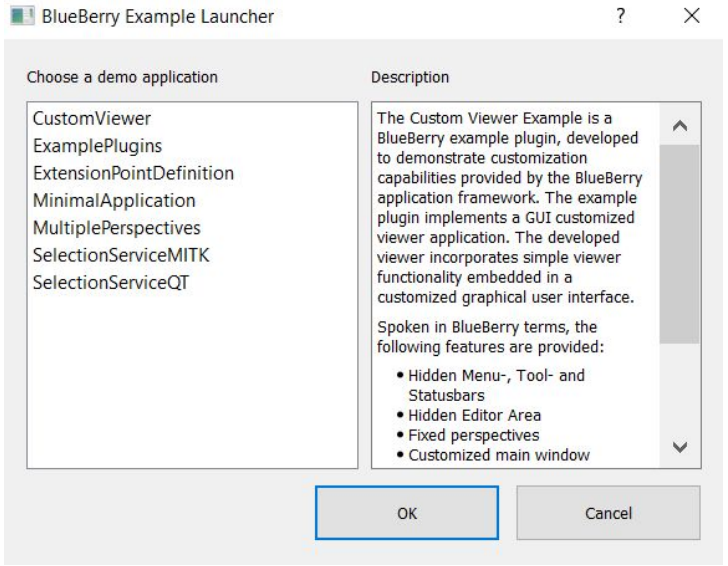
UI Customization: Window Configuration

```
void CustomViewerWorkbenchWindowAdvisor::PreWindowOpen()
{
    berry::IWorkbenchWindowConfigurer::Pointer configurer =
            this->GetWindowConfigurer();

    configurer->SetTitle("Spartan Viewer");
    configurer->SetShowMenuBar(false);
    configurer->SetShowToolBar(false);
    configurer->SetShowPerspectiveBar(false);
    configurer->SetShowStatusLine(false);
}
```

More Information on UI customization

- http://docs.mitk.org/nightly/BlueBerryExamples.html
- http://docs.mitk.org/nightly/BlueBerryExampleMultiplePerspectives.html



Packaging

- Packaging creates a package that contains **all** cmake-enabled applications and **all** cmake-enabled cmd-apps.
- **The name of the package is the name of your *project***, meaning the actual name of the directory. The name of each application/cmd-app is self-defined.
- **Versioning does not happen per application/cmd-app.** The version is the same for everything and is defined as the *git tag* of the MITK sources. By default if you checkout the stable version of MITK that would be v2018.04.2, but you can create new tags by running “git tag vX.X.X” in the cloned MITK repository. Make sure that a tag actually exists in the MITK repo, otherwise “NO TAG FOUND” is used, which might create issues because it contains a space. To check run “git describe --tags” on the MITK repo.

The end!