OpendTect Programmer's Manual - 7.0

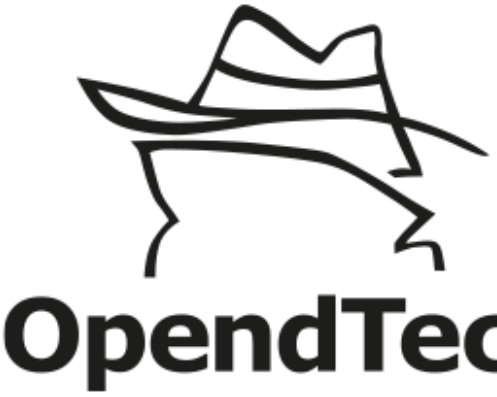


Table of Contents

1 Preface

1.1 About this Manual

This is the manual for developers wanting to use, change or contribute to the source code of OpendTect. It describes all necessary steps taken to download and compile OpendTect from source on Windows, Linux and Mac OS. There is also a separate section for those who want to develop plugins, as it is not necessary to build OpendTect completely to effectively develop and debug your own plugin.

This document was written using MadCap Flare. Two versions are published: an html manual for online use and a pdf version for printing. Both the html and pdf manual can be separately downloaded from the documentation page of the dGB website.

While every precaution has been taken in the preparation of this manual, it is possible that last minute changes to the user interface are not reflected in the manual, or are not described accurately. Please help us improve future editions by reporting any errors, inaccuracies, bugs, misleading or confusing statements you encounter.

Other Manuals:

- User documentation - step by step explanation of all functionality in OpendTect
- How-To Instructions - this documentation describes How-To apply the software effectively.
- Training Manual - comes with a 3D data set for self-training. Download from the documentation page
- Administrator's Documentation.

1.2 Release Notes

This is the programmer documentation for release OpendTect v7.0- an open source post-processing, and seismic interpretation system created by dGB.

OpendTect is released via the internet. Users can download the software from the OpendTect website. It will run without license protection.

OpendTect v7.0 is released under a triple licensing strategy:

- Under the GNU GPL license.
- Under the OpendTect Pro license.
- Under an Academic license.

Under the GNU GPL license OpendTect is completely free-of-charge, including for commercial use.

The OpendTect Pro license gives commercial users access to OpendTect Pro, the commercial version of OpendTect. OpendTect Pro offers extra functionality and allows commercial users to extend the system with additional (closed source) commercial plugins that can either be purchased or leased. The commercial parts of OpendTect are protected by FlexNet license managing software. To obtain a license key for OpendTect Pro and the plugins please contact dGB at info@dgbes.com.

Under the Academic license agreement universities can get free licenses for OpendTect Pro and commercial plugins for R&D and educational purposes.

OpendTect is currently supported on the following platforms:

- Linux (64bit)
- Windows 10 and 11 (64bit)
- macOS 12 (Monterey), 13 (Ventura) and 14 (Sonoma)

1.3 About OpendTect

OpendTect is a free, open source seismic interpretation system and software development platform. The system supports all tools needed for visualizing, analyzing and interpreting 2D, 3D and 4D seismic and Geo_Radar data. The software is written in C++ and the same codebase compiles and runs on Windows, macOS and Linux. It also has a mature plugin programming interface that allows third parties to develop plugins to add functionality to the system without touching the OpendTect source code. A binary installer for OpendTect can be downloaded from the dGB download page.

This source code is released under the GPLv3 or higher license. Commercial and Academic licenses are offered by dGB Earth Sciences for OpendTect Pro, an extension of OpendTect that adds special functions for professional users and the potential to rent or purchase commercial plugins offering access to unique seismic interpretation workflows. The differences in functionality of the open source OpendTect, commercial OpendTect Pro and commercial plugins is compared here.

OpendTect is a flexible and powerful R&D software platform that you can extend for seismic data analysis. dGB Earth Sciences is also available to undertake industry funded (single or multi-client) projects to enhance OpendTect itself or create advanced plugin functionality.

OpendTect is used worldwide by thousands of open source users, thousands of academic users and hundreds of commercial users.

1.4 Copyright

1.5 Acknowledgements

The OpendTect system is developed around concepts and ideas originating from a long-term collaboration between dGB and Statoil. Most of the system was and is developed through sponsored projects. We are indebted to all past, present and future sponsors. To name a few:

- Addax
- ARKCLS
- BG Group
- Chevron
- ConocoPhillips
- Detnor
- DNO
- ENI
- GDF Suez
- Geokinetics
- JGI
- Marathon Oil
- MOL
- OMV
- RocOil
- Saudi Aramco
- Shell
- Statoil
- Talisman
- Tetrale
- The Dutch Government
- Thrust Belt Imaging
- Wintershall
- Woodside

2 Build a standalone plugin

2.1 Introduction to building a stand alone plugin

Making your own software within OpendTect is in principle pretty easy. You could change the software by modifying existing classes and functions, and adding your own stuff to the libs. The advantage is total control. The problem with this approach, however, is that you have to keep the OpendTect sources in sync with new releases. Furthermore, if you cannot convince the opendtect.org people to also make those changes, OpendTect users may not be happy with your work.

An easy way to overcome this is to make your own plugins. Plugins make use of all the facilities of OpendTect but are loaded at run-time and can therefore be developed in a completely independent way. If you then find things that can't be done without modifying the OpendTect environment, it should be much easier to convince the opendtect.org people to take over or even implement those things themselves. One thing you cannot do, is use another compiler than gcc/g++ on Linux/macOS or VC++ on Windows. OpendTect is built with it, if you want to use another compiler (why?) you'll have to make all libs and supporting libs (Qt, OpenSceneGraph) yourself. The make itself should be pretty easy to get started, but there will probably be some porting to do, too.

Requirements:

All platforms:

OpendTect

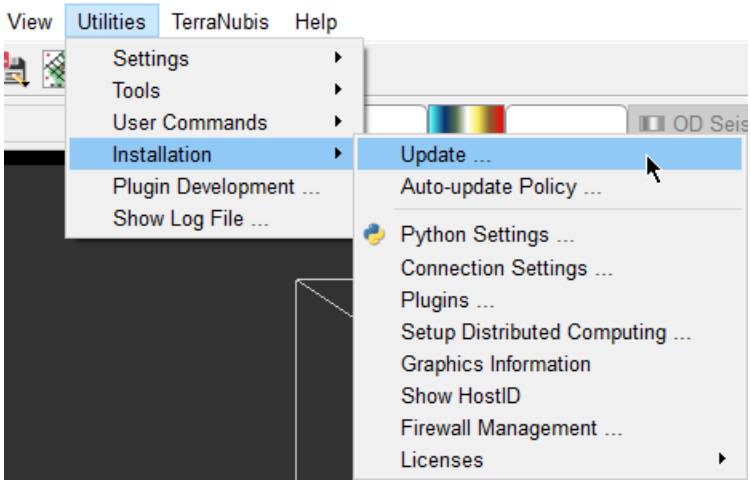
CMake

Windows:

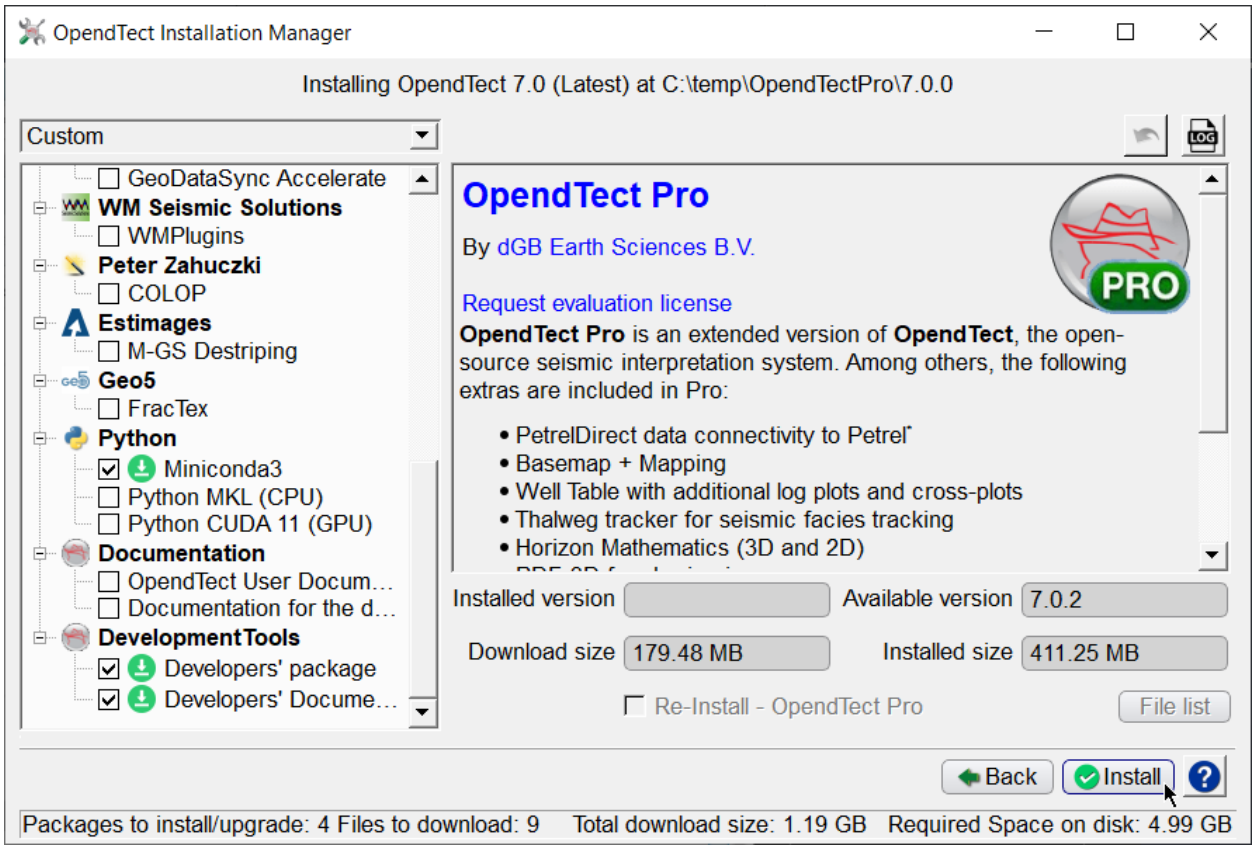
Visual Studio 2022 or 2019 (Community Edition is sufficient)

2.2 Setting up the Environment

For setting up OpendTect to be able to build and test your own plugins; download the installation manager at the OpendTect download page or go to Utilities > Installation > Update in the main menu if OpendTect is already installed.

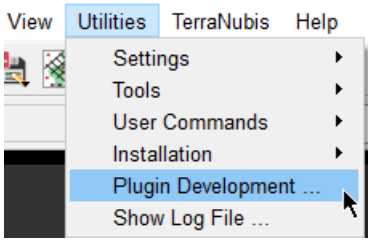


This will open the OpendTect Installation Manager. In the installation manager make sure that besides all plugins and packages you want to have available in OpendTect, the Developer's Package option in Development Tools is checked.

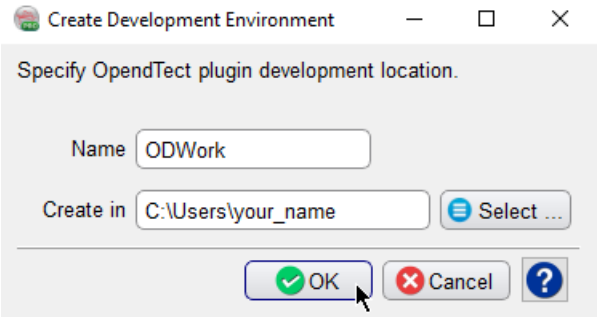


Proceed to download and install the required files. After this you can run OpendTect.

To create the directory where the source code of your plugin will be, go to: Utilities > Plugin Development in the main menu.



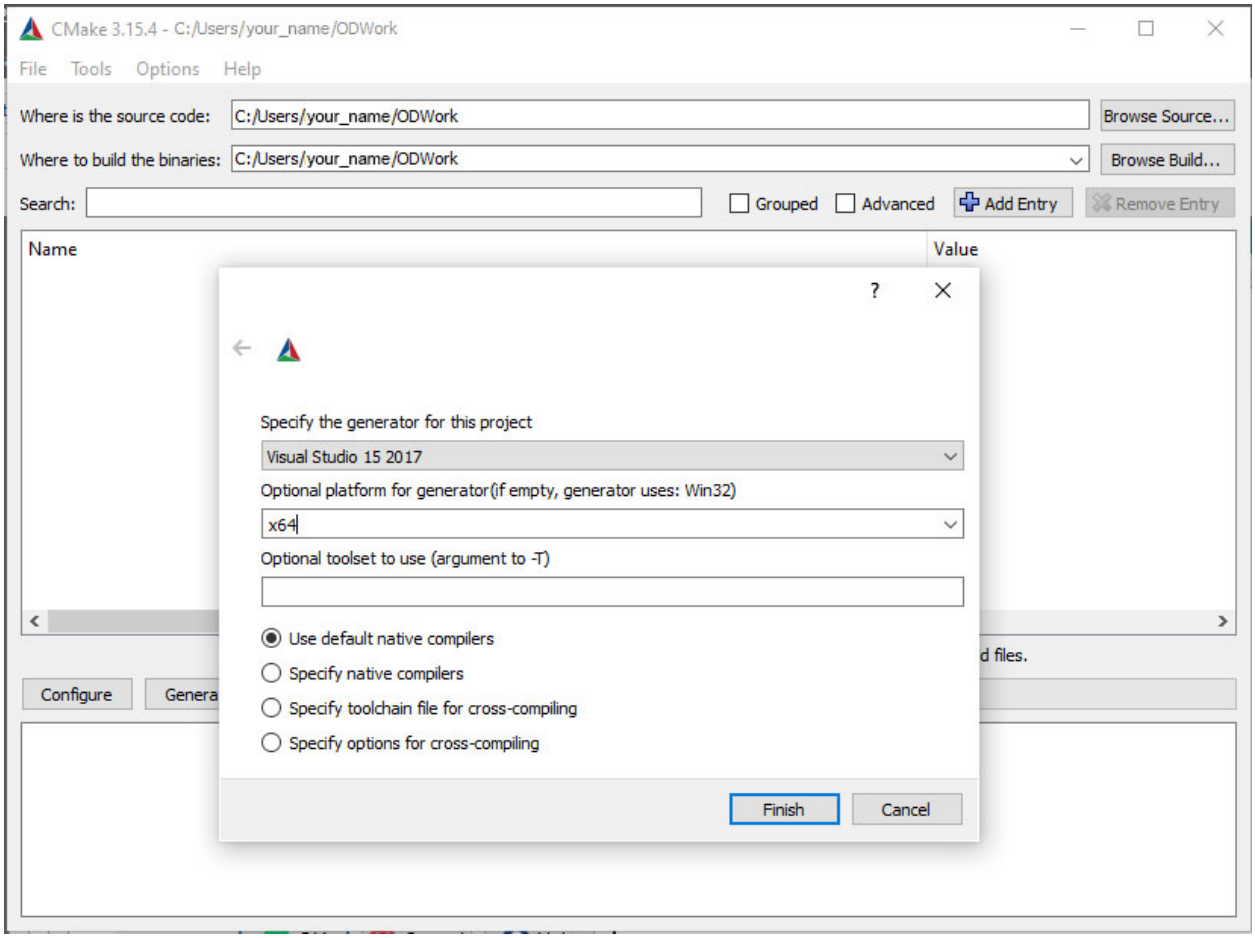
And create the development folder (for example ODWork) at a convenient location. It can be any location, **but not the OpendTect installation folder itself, as this would create conflicts.**



This folder contains example code (Tut Plugin) to easily start developing.

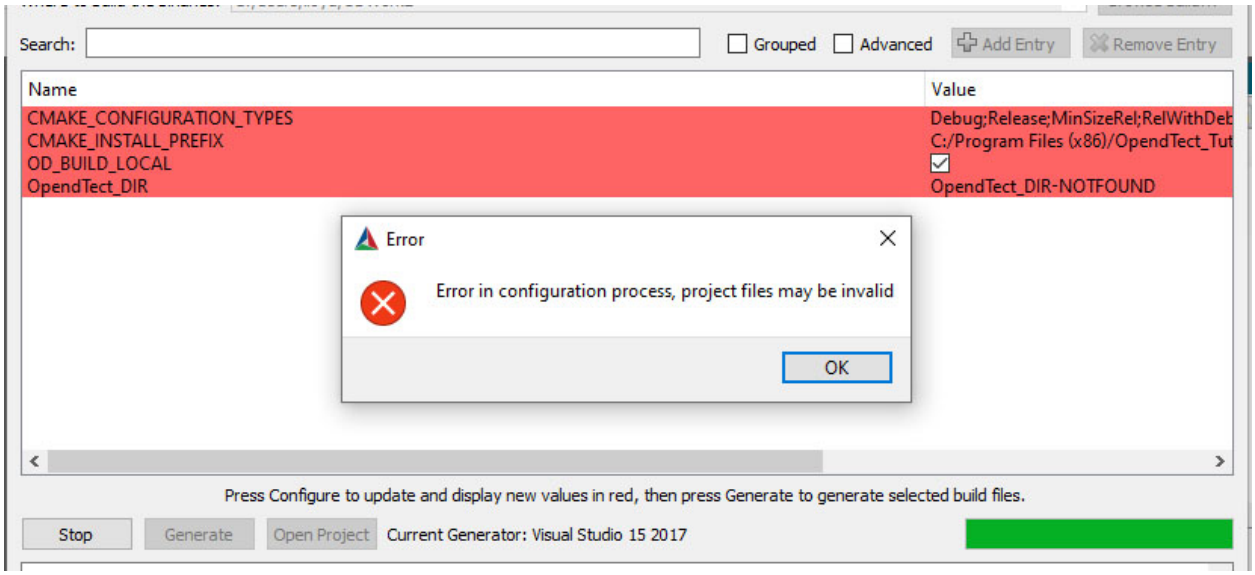
2.3 Building the Tutorial Plugin

After creating the folder, run CMake and select the WORK directory you just created, put the location in both the "source" and "output" fields. Then press the 'Configure' button



Select Visual Studio 15 2017 as the generator and choose x64 as the platform. Click Finish.

The first configuration will probably fail, do not worry, click ok.



And set OpendTect_DIR to point to your installed OpendTect directory, after this you can click Generate to create the necessary project files.

If successful you will see the created files in your WORK folder, and can open the OpendTect_Tutorial_Plugin.sln file to run Visual Studio.

For a more in depth view of the code in the tutorial plugin, please review Chapter 3 of this manual, this chapter will continue with ways to debug and install a plugin.

2.4 Debugging your plugin

Open the plugin solution in Visual Studio, and right-click on the solution item, and open the solution properties window. Here you can specify the debug source file location, so in your case add the location of the OpendTect source files which are installed by the installation manager, i.e. C:\Program Files\OpendTect\7.0. Now select the "od_main" project which is a launcher project to launch od_main debug executable that comes with the developers package. Right click on this project and set to "Set as startup project". When you press F5 for debugging this project will be started. Ignore if you see any Visual studio warnings. The OpendTect main program, od_main.exe will be launched.

From Utilities-Installation-Plugins OpendTect try to load your plugin by browsing to the WORK(D:\WORK)\bin\win64\Debug folder. To debug something, just set a break point in the code. So when the control comes to the break-point it will stop there and you can debug your code. You can also browse into OpendTect source files if you have specified the OpendTect source file location is the solution properties as mentioned above.

2.5 Creating the Help documentation

Like any other commercial application, our plugin also needs a help document which a user can see by clicking on a button in the user interface. The OpendTect help system is quite flexible and allows a plugin to define its own way of showing help information. But in most cases, all you want is to open an HTML file either stored locally or on the web. For this purpose, we have a class called SimpleHelpProvider that provides a key-link based help system. The idea is to have a common base URL (can be a local file path) and then append links for individual help documents to this base URL, based on keys.

So, you need to define your own HelpProvider class as a subclass of SimpleHelpProvider and initialize it when the plugin loads. A good example is the TutHelpProvider defined in uitutpi.cc:

```
class TutHelpProvider : public SimpleHelpProvider
{
public:
TutHelpProvider( const char* baseurl, const char* linkfnm )
    : SimpleHelpProvider(baseurl,linkfnm)
{}

static void initClass()
{
    HelpProvider::factory().addCreator( TutHelpPro-
vider::createInstance, "tut");
}

static HelpProvider* createInstance()
{
    FilePath fp( GetDocFileDir(""), "User", "tut" );
    BufferString baseurl( "file:///" );
    baseurl.add( fp.fullPath() ).add( "/" );

    fp.add( "KeyLinkTable.txt" );
    BufferString tablefnm = fp.fullPath();

    return new TutHelpProvider( baseurl.buf(), tablefnm.buf
() );
}

};
```

The three key elements of this class are:

• The provider key: 'tut' in this case.

• The base URL: Here it is a local path inside the OpendTect installation. But it can as well be a web URL like 'http://doc.opendtect.org/'

• The key-link table, which is read from a file 'KeyLinkTable.txt' here. But you can also make it on-the-fly using the function addKeyLink. That is rather convenient if you are doing it just for a couple of plugins.

Then in the UI you can use a HelpKey comprising of two parts: the provider key ('tut' for example) and the key for the individual UI, like 'hor' in uiHorTools:

```
uiTutHorTools::uiTutHorTools( uiParent* p )
    : uiDialog( p, Setup( tr("Tut Horizon tools"),
                          tr("Specify process parameters"),
                          HelpKey("tut","hor") ) )
```

When the user clicks on the help button the HelpProvider will look for the link for the corresponding key, append the link to the base URL and open the document

2.6 Installation and auto-loading

Once you have made your own plugin, you probably would like it to be loaded automatically whenever OpendTect is started. OpendTect provides some facilities that do just that.

2.6.1 Preparing a plugin for auto-load

`#include "odplugin.h"` is needed for the PluginInfo structure and the PI_AUTO_INIT_xxx defines.

The GetxxxxPluginType() specifies when a plugin is loaded:

• `PI_AUTO_INIT_EARLY : Plugin is loaded before construction of main window`

• `PI_AUTO_INIT_LATE : Plugin is loaded after construction of main window`

The default is `PI_AUTO_INIT_LATE`, so you only have to define anything if the plugin needs to be loaded early: then use `mDefODPluginEarlyLoad(YourPluginName)`.

2.6.2 Installing plugins for auto-load

The auto-load tool of OpendTect looks for plugins to load in two places:

1) Where are the .ALO files stored? The two locations searched are (in this order):

- <userdir>/plugins/<platform_dir>
- <systemdir>/plugins/<platform_dir>

2) Where are the plugin libraries? Locations are:

- <userdir>/bin/<platform_dir>/[Release|Debug]
- <systemdir>/bin/<platform_dir>/[Release|Debug]

The <userdir> is determined as follows:

- If it is set, $OD_USER_PLUGIN_DIR
- Else, the user settings directory is used: ~/.od

On Windows, your 'Personal directory' is located at $HOME if this is defined. Otherwise, $USERPROFILE is used. Also see the specific notes in the windows documentation.

2.6.3 Using .alo files

Auto Load files are simple text files that tell a program which plugins it is supposed to load from the 'libs' directory. Since OpendTect contains multiple programs, each program has its own set of .alo files '<program name>.*.alo', while the plugins can be shared between multiple programs. OpendTect will scan for any file with this naming convention. So od_main.john.alo is perfectly OK.

Since there are multiple vendors and/or plugin sets, each vendor can make his own .alo files. od_main, for example, will look at any file named od_main.*.alo. For this example, the default plugins are specified by od_main.base.alo, while dgb's plugins are specified by od_main.dgb.alo. This way, each vendor can make his own .alo files, without interfering with others.

A .alo file is nothing more then a simple list of plugins, without extensions. For example, this could be in an od_main.base.alo file:

```
Annotations

Madagascar

uiMadagascar

CmdDriver

GMT

uiGMT
```

Note that for each platform, a specific .alo file must be created. Usually, they will be the same, but some plugins might not be relevant or supported on all platforms.

The plugins in the .alo files are loaded in the order as specified in the file. The alo files themselves are handled in alphabetical order.

2.7 Distributing your plugin

The publishing and distribution of OpendTect plugins is pretty straightforward. The .alo files can be installed in the plugins/platform (`$DTECT_APPL/plugins/$HDIR`) directory, while the actual plugins (the .DLL, .so or .dylib files) go in the normal bin sub-directory.

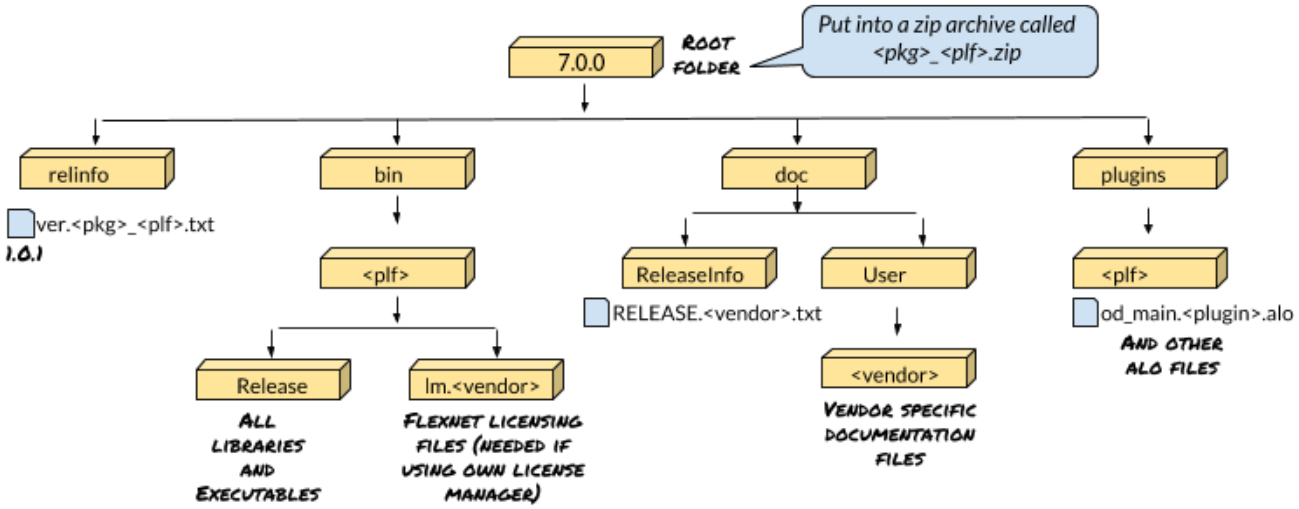
On Unix, this means that you can make a tar.gz or zip file containing the plugins in a directory structure as described above, which can be extracted into the existing OpendTect installation directory.

On Windows this is also possible, but it is more common to use an auto-extracting installer to do this. For more info on this, see the windows documentation.

If you want your plugins to be used around the world, then you may want to contact support@dgbes.com to get your plugin(s) distributed via the OpendTect installation Manager. Be prepared to have thedgbes.com people take a look at your code and test the stability. Then make the packages along the lines described below. You'll also have to provide information about yourself and the plugin - and a picture of a certain size.

Preparing for the installation manager

The general structure of a package is explained in the following diagram:



It is important that you make the packages nicely modular. Even if you have only two platforms yet, still it's a good idea to split the stuff in platform-independent and platform-dependent stuff. And separate documentation. In that case there would be 4 packages:

• The platform-independent part (hidden for the user)

• Part for platform 1

• Part for platform 2

• Documentation

The user will see only two: the plugin itself and the documentation.

Then the naming of the packages. Let's not make a big specification document; you can guess this by looking at what is now in the opendtect.txt file. Specifics:

• We will need a similar package definition file (<vendor>.txt). That is the sort of info we need. Don't worry about the codes you see in there, just the basic info like descriptions and dependencies. That would allow us to make this file. You can deliver the whole file if you want to, but we can also maintain it.

• Provide an image for each package you deliver and one for your company (vendor). The target size would be around 100x100 for the product logo and 16x16 for the vendor logo.

• Make sure every package contains a file: `<OpendTect version>/relinfo/ver.<package_name>[_plf].txt` Like:

`7.0.0/relinfo/ver.jimsinversion_lux64.txt`

`7.0.0/relinfo/ver.pppraytrace.txt`

You can have your own version numbering, but it has to have this form:

`number.number.number[optional_free_text_without_dots_starting_with_non_digit]`

You are completely free in your numbering, and the optional text. The installer uses the '>' operator for every part. The numbers have to be integer numbers, and will be compared as integers.

Users cannot update a package without also updating the packages that these are dependent on. This circumvents the need to specify exactly the dependencies on which versions on what other packages.

3 The Tutorial Plugin

3.1 Introduction

We have created the Tutorial plugins that you can find in your work environment. As is common in OpendTect, there is a plugin 'Tut' for non-ui, real-work stuff, and the 'uiTut' for the GUI part.

The idea of the tutorial plugins is to show a variety of common things that one might want to do, rather than make something useful for end-users. For that we'll make the following tools:

• Manipulating some seismic data (read, process, write)

• The same, but now using an Attribute

• Do some work with horizons

• Do some work with wells

In the process, we'll see how to:

• Create menu items and toolbar icons

• Make right-click tree item menus

• Work in the OpenSceneGraph 'vis' world

• Work with DataPack's and create flat displays

3.2 uiTut plugin

In `uiTut`, the GUI consists of two parts. One deals with opening an independent dialog box via a menu item in the 'Utilities' menu. The other part gets the 'Tutorial' attribute listed in the 'Edit Attributes' dialog and creates the input fields in the same dialog box. It also sends the input parameters to Tutorial for attribute computation

Let us first have a look at the independent dialog part which in turn has two parts -- one for seismic tools and the other for horizon tools. The only interesting part is the `uiIOObjSel` class which allows you to select an item from a set -- a horizon or a seismic cube (subclass `uiSeisSel` is used for seismic cube selection).

Both `uiSeisTools` and `uiHorTools` use the class `uiTaskRunner`, which triggers the Executor's in the Tut plugin. The class `uiTaskRunner` also displays a progress bar which keeps the user informed about the progress of the process.

Now we come to the attribute part. In the uitutorialattrib.cc file we see that although `uiAttrDescEd` is not a `uiDialog` like the the `uiHorTools`, it still is a valid parent (being a `uiGroup`) for the various UI elements. A nice feature of OpendTect is clear from the first line in the constructor: the `inpfld` is a special Attribute UI class which is handled just like any pre-defined `uiBase` or `uiTools` class. This illustrates that in the OpendTect GUI system, not only pre-made GUI elements are 'first class' - new objects with different shape and behavior attached will be usable transparently by any other GUI class.

Coming to the plugin 'main' file uitutpi.cc, like any typical UI plugin, `uiTut` is a LATE plugin, which means that it will be loaded only after the rest of the UI is already in place. Thus, you must not put `mDefODPluginEarlyLoad()`.

Then comes the second 'special' plugin function `GetxxxPluginInfo()`. You may want to refer to the definition of the class PluginInfo for a better understanding of the above function. It allows the plugin manager to make this info available to the world.

```
mDefODPluginInfo(uiTut)
{
    DefineStaticLocalObject( PluginInfo, retpi,(
        "Tutorial plugin",
        "OpendTect",
        "dGB (Raman/Bert)",
        "3.2",
        "Shows some simple plugin development basics."
             "\nCan be loaded into od_main only." );
```

```
    return &retpi;
}
```

And the last 'special' function is the one which gets things going:

```
mDefODInitPlugin(uiTut)
{
    mDefineStaticLocalObject( PtrMan, theinst_, = 0 );
    if ( theinst_ ) return 0;
        theinst_ = new uiTutMgr( ODMainWin() );
    if ( !theinst_ )
        return "Cannot instantiate Tutorial plugin";

    uiTutorialAttrib::initClass();
    TutHelpProvider::initClass();
    return 0;
}
```

3.3 Tut plugin

The responsibility of uiTut is limited to talking to the user and getting the input parameters. The real work is done behind the scene by the non-UI Tut plugin. And that is the reason why it is of type EARLY. This particular plugin tells OpendTect's application manager that it wants to be loaded early - i.e. before any build of tables, data structures or user interfaces are made. That is typical of 'Real Work' plugins. The alternatives are NONE (which is very uncommon) and LATE, which is typical for UI plugins that want to start working when all objects have already been created. In this case, we need to specify that we have an EARLY plugin:

```
mDefODPluginEarlyLoad(Tut)
```

3.4 SeisTools

Let us first look at the direct seismic operations, that are handled by the class `SeisTools`, which in turn is a subclass of class Executor. 'Real work' is done by the function `nextStep()` which is typical of class Executor. Here, three different operations are possible: Scaling, where you can multiply the data values by a certain factor and apply a shift; Squaring, where, as the name suggests, you can take a square of the data values; and Smoothening, where you can take the arithmetic average of 3 or 5 samples depending on the filter strength. Traces are read one-by-one by a `SeisTrcReader` and supplied to the function `handleTrace()` where the actual computation is done. Then a `SeisTrcWriter` writes the output traces one-by-one to the output cube.

```
int Tut::SeisTools::nextStep()
{
    if ( !rdr_ )
        return createReader() ? Executor::MoreToDo()
                              : Executor::ErrorOccurred();

    int rv = rdr_->get( trcin_.info() );
    if ( rv < 0 )
        { errmsg_ = rdr_->errMsg(); return Execut-
or::ErrorOccurred(); }
    else if ( rv == 0 )
        return Executor::Finished();
    else if ( rv == 1 )
    {
        if ( !rdr_->get(trcin_) )
            { errmsg_ = rdr_->errMsg(); return Execut-
or::ErrorOccurred(); }

        trcout_ = trcin_;
        handleTrace();

        if ( !wrr_ && !createWriter() )
            return Executor::ErrorOccurred();
        if ( !wrr_->put(trcout_) )
            { errmsg_ = wrr_->errMsg(); return Execut-
or::ErrorOccurred(); }
    }
```

```
    return Executor::MoreToDo();
}
```

Scaling and squaring are single-sample operations. But as you can see in the implementation of the function `handleTrace()` below, smoothening involves multi-sample computation. It requires separate input and output traces. Otherwise, if we did the operation on the same trace, we would be taking the modified values of samples preceding the current sample. For the sake of simplicity, we make a copy of the input trace to store the output values. This is not a good practice as it results in duplication of data. But since it is a tutorial, our aim is to keep the code as simple as possible and leave the efficiency part for serious programming.

```
void Tut::SeisTools::handleTrace()
{
    switch ( action_ )
    {

    case Scale: {
        SeisTrcPropChg stpc( trcout_ );
        stpc.scale( factor_, shift_ );
    } break;

    case Square: {
        for ( int icomp=0; icomp < trcin_.nrComponents();
icomp++ )
        {
            for ( int idx=0; idx < trcin_.size(); idx++ )
            {
                const float v = trcin_.get( idx, icomp );
                trcout_.set( idx, v*v, icomp );
            }
        }
    } break;

    case Smooth: {
        const int sgate = weaksmooth_ ? 3 : 5;
        const int sgate2 = sgate/2;
        for ( int icomp=0; icomp < trcin_.nrComponents();
icomp++ )
        {
            for ( int idx=0; idx < trcin_.size(); idx++ )
            {
                float sum = 0;
```

```
                int count = 0;
                for( int ismp=idx-sgate2; ismp <= idx+s-
gate2; ismp++)
                {
                    const float val = trcin_.get( ismp,
icomp );
                    if ( !mIsUdf(val) )
                    {
                        sum += val;
                        count++;
                    }
                 }
                 if ( count )
                     trcout_.set( idx, sum/count, icomp );
            }
         }
    } break;
}
nrdone_++;
}
```

3.5 HorTool

Similar to `SeisTools`, `HorTool` performs some simple operations on horizons: thickness computation and smoothening. Each of these operations is handled by a subclass of `HorTool` which is a subclass of `Executor` and as expected the computation is performed by the function `nextStep()`. You may notice here that no object of class `HorTool` is defined anywhere. It is only used as the base class for classes `ThicknessCalculator` and `HorSmoothener`. Let us have a look at the `nextStep()` function in class `ThicknessCalculator` to see how the data values are accessed in a Horizon3D:

```
int Tut::ThicknessCalculator::nextStep()
{
    if ( !iter_->next(bid_) )
        return Executor::Finished();

    int nrsect = horizon1_->nrSections();
    if ( horizon2_->nrSections() < nrsect ) nrsect = hori-
zon2_->nrSections();

    for ( EM::SectionID isect=0; isectgetPos( isect, subid
).z;
        const float z2 = horizon2_->getPos( isect, subid
).z;

        float val = mUdf(float);
        if ( !mIsUdf(z1) && !mIsUdf(z2) )
            val = fabs( z2 - z1 ) * usrfac_;

        posid_.setSubID( subid );
        posid_.setSectionID( isect );
        horizon1_->auxdata.setAuxDataVal( dataidx_, posid_,
val );
    }

    nrdone_++;
    return Executor::MoreToDo();
}
```

Please note the difference in the function `dataSaver` in the two classes. In `ThicknessCalculator`, it saves the auxilary data, whereas in `HorSmoothener`, it saves the geometry.

3.6 The Tutorial Attribute

We have seen the direct seismic approach to simple operations on seismic data in SeisTools. For our purpose, it suits well. But the main problem with this approach is the difficulty in multi-trace handling. Moreover, for large seismic volumes, handling each trace one-by-one may slow down the process. This brings us to another approach called Attributes. In this example, we define the Tutorial attribute to do things once done by SeisTools. As we discuss different aspects of making an attribute, we will also discuss its advantages over the direct seismic approach.

The main plugin file "tutpi.cc" makes a call to `Tutorial::initClass()`. The class Tutorial (tutorialattrib.h) is defined as a subclass of `Attrib::Provider` class. Every attribute is a provider, each can thus be used as input for another attribute.

3.7 Steering

A Steering cube, as the name suggests, works as a guiding cube. It stores the Inline dip and Crossline dip at each point, which guides the attribute engine in multi-trace computations. In case of our Tutorial attribute, we can use the steering data for horizontal smoothening. The key function is `initSteering()` which makes the steering data available in the form of shifts relative to the central trace. To understand how this shift is used during computation, please refer to the horizontal smoothening section in the function `computeData()`.

Some fundamental attribute functions are listed here:

`createInstance()`

This function is standard for every attribute, here is the attribute constructor called. Use the macro `mAttrDefCreateInstance` to define `createInstance`:

```
mAttrDefCreateInstance(Tutorial)
```

`initClass()`

This static function initializes the attribute: sets up the parameters and the number and type of the inputs and outputs. You can compare this to what you see in Opendtect in the attribute definition window after loading the `uiTut` plugin.

If you look at the parts of the implementation carefully, (tutorialattrib.cc) you'll see that each parameter is built up following this example:

```
EnumParam* action = new EnumParam( actionStr() );
    action->addEnum( "Scale" );
    action->addEnum( "Square" );
    action->addEnum( "Smooth" );
    desc->addParam( action );
```

Every parameter is required by default, to overrule this use `setRequired(false)`

`initClass()` also adds the attribute to the attribute factory. In this case, as every attribute is a provider, the Tutorial attribute is added to PF() (the `Attrib::ProviderFactory` singleton access function).

`updateDesc()`

Will be used not only to update the parameters but also the number and type of the outputs and to add or disable some inputs. If you look at the implementation for the tutorial attribute, this function just allows to enable or disable the inputs (factor, shift and smooth) according to the action chosen by the user

getInputOutput()

we need to define this initialization function because we have Steering. Steering always carries two outputs and we need them both.

initSteering()

If we are using steering data, this function prepares the steering input for use in computation. A subvolume is generated around the central trace, with the size of the subvolume specified by the stepout. This data contains the shifts in terms of number of samples for each trace in the subvolume relative to the central trace.

```
void Tutorial::initSteering()
{
    if ( inputs[1] && inputs[1]->getDesc().isSteering() )
        inputs[1]->initSteering( stepout_ );
}
```

getInputData()

Before the work can be done, some input has to be given. This function is the place where you specify how to get your input data. For the Tutorial this is the seismic data. But it can also be Steering Data or any other attribute.

```
bool Tutorial::getInputData( const BinID& relpos, int zintv
)
{
    if ( inpdata_.isEmpty() )
        inpdata_ += 0;
    const DataHolder* data = inputs[0]->getData( relpos,
zintv );
    if ( !data ) return false;
    inpdata_.replace( 0, data);

    if ( action_ ==2 && horsmooth_ )
    {
        steeringdata_ = inputs[1] ? inputs[1]->getData(
```

```
relpos, zintv ) : 0;
        const int maxlength = mMAX(stepout_.inl,
stepout_.crl)*2 + 1;
        while ( inpdata_.size() < maxlength * maxlength )
            inpdata_ += 0;

        for ( int idx=0; idxgetData( relpos + posand-
steeridx_.pos_[idx] );
                if ( !data ) continue;
            inpdata_.replace( posandsteeridx_.steeridx_
[idx], data);
        }
    }

    dataidx_ = getDataIndex( 0 );

    return true;

}
```

You will notice from here that the calculation of the attributes is not done on traces but using a different object, the `DataHolder`. The dataholder contains a set of `ValueSeries` which holds the value of every sample of the `SeisTrc`. Advantage: in case of an attribute which has other attributes as inputs, data is available in the corresponding dataholders, it thus saves a lot of time (easier and much faster to read some floats in a `ValueSeries` than to get values from a `SeisTrc`). Stored data are read from cubes of seismic traces and written the same way.

The `DataHolder` is also carrying some specific information about the trace to be processed, like the start sample number and the number of samples you wish to calculate.

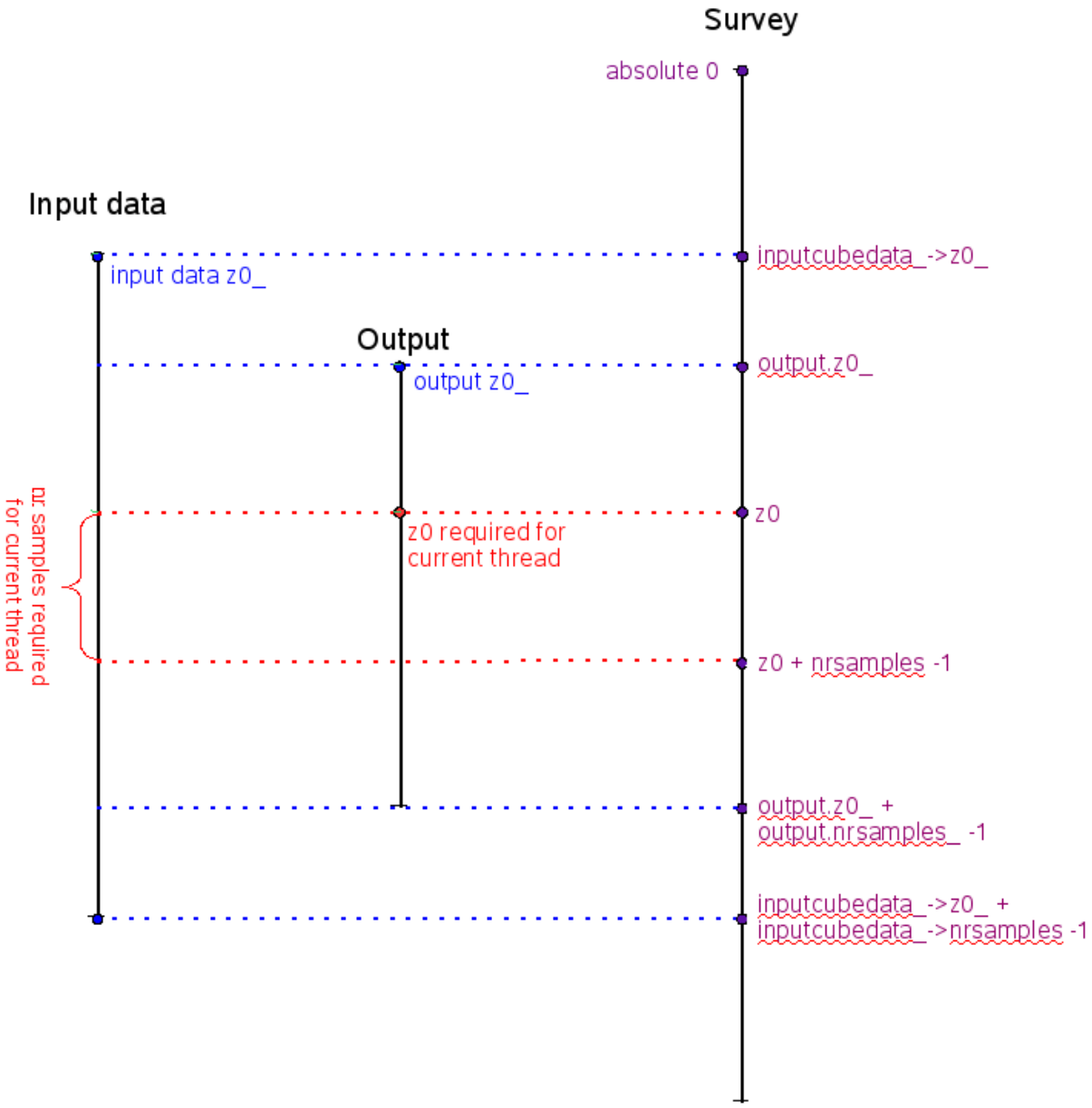
Another important remark: calculation is made using sample numbers, not time or depth

Most of the rest of the methods are there to comply with the `Attrib::Provider` interface - see the `Attrib::Provider` documentation. The basic idea is that for each sample of each trace one or more attribute values can be calculated. The number of attribute values (or outputs) is defined in the `initClass()` function. If your input requires additional samples (timegate) or neighbouring traces (stepout), you will have to define `reqZMargin()` and `reqStepout()` respectively.

`computeData()`

When we want to look at the actual work, the place to be is the `computeData()` method. This is the place where you define the mathematics for calculating the attribute. This function is called for each trace of your output cube.

In the `computeData()` method, we are faced with a number of Z ranges. To be able to support multi-threading, `computeData` must be ready to only process part of the trace. Then, also, we can have input cubes that are larger than requested or desired, or smaller than that. This delivers a rather nasty picture of Z indexes that we really cannot circumvent. To make things at least clear, the indexes are all related to the the absolute Z=0. This is where everything refers to. Then, we have different start Z indexes for each of the input cubes and the output cube. These are named 'z0_' in the corresponding DataHolders.



Let us have a look at the `Tutorial::computeData` function and compare it with the code in `SeisTools`. The algorithm for actual computation is the same in both the cases, but there is a marked difference in the manner in which seismic data is accessed in each case.

```
bool Tutorial::computeData( const DataHolder& output, const
BinID& relpos,
                            int z0, int nrsamples, int
threadid ) const
{
```

```
    for ( int idx=0; idx < nrsamples; idx++ )
    {
        float outval = 0;
        if ( action_==0 || action_==1 )
        {
            const float trcval = getInputValue( *inputdata_,
dataidx_,
                                                idx, z0 );
                                                outval =
action_==0 ? trcval * factor_ + shift_ :
                                            trcval * trcval;
                                            }
        else if ( action_==2 && !horsmooth_ )
        {
            float sum = 0;
            int count = 0;
            for ( int isamp=sampgate_.start; isamp <= samp-
gate_.stop; isamp++ )
            {
                const float curval = getInputValue( *in-
pdata_[0], dataidx_,
                                            idx + isamp, z0 );
                                            if ( !mIsUdf(curval)
)
                {
                    sum += curval;
                    count ++;
                }
                }
            outval = sum / count;
        }
        else if (action_ == 2 && horsmooth_ )
        {
            float sum = 0;
            int count = 0;
            for ( int posidx=0; posidx < inpdata_.size();
posidx++ )
            {
                if ( !inpdata_[posidx] ) continue;
                const float shift = steeringdata_ ?
                        getInputValue(
```

```
*steeringdata_,posidx, idx, z0 ) : 0;
                        const int sampidx = idx + ( mIsUdf
(shift) ? 0 : mNINT(shift) );
                if ( sampidx < 0 || sampidx >= nrsamples )
continue;
                const float val = getInputValue( *inpdata_
[posidx],
                                                 dataidx_, sampidx,
z0 );
                                                 if ( !mIsUdf(val) )
               {
                   sum += val;
                   count ++;
               }
               }
          outval = sum / count;
        }

        setOutputValue( output, 0, idx, z0, outval );
    }

    return true;

}
```

4 Build OpendTect from source

4.1 Introduction

This section lays out a step by step setup of a standard build environment on Windows 10/11, MacOS and Linux. Using a different setup might be possible, but probably requires extra work.

Requirements

A C++ compiler and compilation tool chain:

- *Windows*: msvc2022 64 bit (>= v17.3.1) or msvc2019 64 bit (>= v16.7.5). The free community edition is sufficient.
- *macOS*: SDK 10.14
- *Linux:* gcc 64 bit version 5.4.0 or higher

CMake version 3.14 or higher

The c++14 version is enabled by default on all platforms.

Dependencies

To build the software you need to also download and install/build a few dependencies which probably are not installed in your system. The version of dependencies varies between the branches. The Qt dependencies are available in binary installers, the others have to be built from source.

BRANCH	DEPENDENCIES
main	Qt 5.15.2, OpenSceneGraph 3.6.5, Proj 9.0.1 (optional), Sqlite 3.38 (optional), HDF5 1.12.2 (optional)
od7.0	Qt 5.15.2, OpenSceneGraph 3.6.5, Proj 9.0.1 (optional), Sqlite 3.38 (optional), HDF5 1.12.2 (optional)
od6.6_rel, od6.6	Qt 5.15.2, OpenSceneGraph 3.6.5, HDF5 1.12.2 (optional)
od6.4.5, od6.4	Qt 5.9.6, OpenSceneGraph 3.6.3

4.2 Setting up the environment

OpendTect source code

To build OpendTect from source you need to pull the source code from https://-github.com/OpendTect/OpendTect. Then build the solution with CMake. In this case launch CMake and browse to your OpendTect source folder and start configuring and generating from CMake. CMake will prompt for Qt and OpenSceneGraph directory location, select them from CMake itself and continue. Once the projects and solution are generated you can start building OpendTect.

You can use the following git commands to clone the repositories to your local machine.

```
git clone https://github.com/OpendTect/OpendTect.git --single-branch --branch main od
```

```
git clone https://github.com/OpendTect/OpendTect.git --single-branch --branch od7.0 od7.0
```

CMake

Download CMake from https://cmake.org/download/.

Qt

For the Qt install, Qt Online Installer is the easiest and recommended way. The following components must be selected depending on your build platform:

- Desktop msvc2019 64- bit (Windows), SDK 10.14 (macOS) or gcc 64 bit (Linux)
- QtWebEngine
- OpenSSL (optionally, but is required for using https URL to download and/or using tools in network)
- Optionally source code or debug information files

For building the main, od7.0, od6.6_rel and od6.6 branch Qt 5.15.2 is required. For branches od6.4.5 and od6.4 Qt 5.9.6 is required.

OSG

Get the OSG binaries here: https://objexx.com/OpenSceneGraph.html. The 3.6.5 release and/or debug version is needed for building the main, od7.0, od6.6_rel and od6.6 branches of OpendTect. The 3.6.3 release and/or debug version is needed for building the od6.4.5, od6.4 and od6.5 branches.

Configure using CMake, compile and install.

Proj (optional)

Configure using CMake, compile and install.

Sqlite (optional)

Retrieve from their download site or the OpendTect SDK.

HDF5 (optional)

Get HDF5 here: https://www.hdfgroup.org/downloads/hdf5. The link to HDF5 requires to provide the path to an existing HDF5 installation. All versions above 1.10.3 are supported, but using the current API 1.12 is preferred. Installation is best done using their binary installations (on Windows especially), or from the package manager on Linux. Windows developers however need to recompile the sources since no debug binary libraries can be downloaded.

4.3 Building OpendTect

Windows

Run CMake and select the folders for the source code and where to build the binaries (for example C:\dev\od for both).

Configure CMake. Select generator Visual Studio 16 2019 or Visual Studio 17 2022 and platform x64. Then set the following variables:

- QTDIR= set this to the Qt install path for the appropriate version of Qt for the OpendTect version, e.g. C:\Qt\5.15.2\msvc2019_64
- OSG_DIR="OpenSceneGraph install path"
- PROJ_DIR="PROJ install location" or BUILD_PROJ=ON or OD_NO_PROJ=ON to disable it
- SQLITE_DIR="SQLITE install location" (optional, but required by Proj)
- HDF5_ROOT="HDF5 install path" (optional)

Press configure. If it says configuring done, press the generate button to build the solution files.

Start Visual Studio 2019 or Visual Studio 2022 and open the OpendTect build solution (OpendTect.sln file). Before starting the build it is a good idea to set od_main as the Startup Project. Then start the build.

Linux

Configure CMake ensuring to set the following variables:

- QTDIR= set this to the Qt install path for the appropriate version of Qt for the OpendTect version

- OSG_DIR="OpenSceneGraph install path"
- PROJ_DIR="PROJ install location" or BUILD_PROJ=ON or OD_NO_PROJ=ON to disable it
- SQLITE_DIR="SQLITE install location" (optional, but required by Proj)
- HDF5_ROOT="HDF5 install path" (optional)
- OpenGL_GL_PREFERENCE=LEGACY
- ZLIB_INCLUDE_DIR= set this if not being found by CMake
- ZLIB_LIBRARY= set this if not being found by CMake

Run make in the toplevel folder (i.e. where this README.MD file is located)

5 Contributing to the Source Code

5.1 Introduction

The repository contains a number of release branches and 2 development branches. The current stable release branch is 7.0

The development branches are:

BRANCH DESCRIPTION

main

This is the bleeding edge where migration of OpendTect to new versions of its major dependencies, Qt and OpenSceneGraph, is tested and major new functionality is added.

od7.0

This is the main development branch for the next stable release series 7.0. No major new features are currently being added to this branch as it is being prepared for release.

Very short

OpendTect:

• Is in C++ and a tiny bit of C

• Uses CMake, which makes it easy to port across platforms

• Is built with design principles and strict separations of functionality

• Uses exclusively open source tools

• Can be extended using plugins

C++

OpendTect is a C++-based environment. A couple of years ago, C++ was declared dead by many as a result of the Java hype. In some areas, Java is indeed far better suited. In our part of the world (geosciences, in particular geophysics-related), we don't think Java can match the advantages of C++: Fast yet flexible, Low-level yet supporting high-level OO constructs. And, we would be terrified having to program without templates.

That doesn't mean that programming in C++ auto-magically delivers good software, and neither that performance comes easy. Those are some of the things that can be reached by a good design.

5.2 Design principles

There are many aspects of software that can be categorised as 'good'. These include robustness, flexibility, high performance, compactness, maintainability, understandability. Software engineering is all about making choices - every aspect costs resources and there's always a limit to that. So, even if one tries to optimise all 'good' aspects, there will be different degrees of emphasis on each of them.

As OpendTect was developed in a research-minded environment, flexibility is a high priority. The Object-Oriented toolkit delivers many tricks to make software more flexible. Some of these tricks nowadays have labels - 'Design patterns' - like the ones in the 'Design Patterns' book (Factories,Singletons,etc.). Many constructs in the software are fit to match the problem though, always with a number of design principles in mind:

• OCP Open/Closed: classes and modules should be open for extension, but closed for modification.

• SRP Single responsibility: only one class or module does one thing well and complete.

• LSP Substititutability: inheritance for interfaces makes classes substitutable.

• DIP Dependency inversion: depending on abstract base classes inverts dependencies from high-level on low-level into dependency on stable high-level abstractions.

• DIF Don't Implement the Future: All source code present should actually be used now.

• NBS No BullShit: Create constructs if needed, not because they're cool or anything.

The last two are, [cough] of our own making. DIF ensures that there are no large amounts of unused code lying around to be maintained, NBS delivers a system that is as simple as possible given the constraints.

You may also want to look at the design/coding rules described in Chapter 6.

5.3 Isolation of external services

When services from another package (Qt, OpenSceneGraph, ...) are used, there is always an isolating layer - either a complete module or a class that uniquely uses those services. For software engineers this is an obvious action were it only to reduce the dependency problems.

There is however more to it. External services tend to be designed for much more general purposes than what is needed by OpendTect. Furthermore, some services that will be very important for us won't be available. And, we may not like the form in which the services are presented; moreover, the data structures used in the external package seldomly fit nicely with ours.

To overcome all this, and get a nice insulation at the same time, all external services are embedded in service layers that:

• Do exactly what we need

• Don't expose the external package's header files

• Use data structures that are common in OpendTect

Isolation like this is present for a variety of subjects, from threads, sockets, file handling to User Interface building and Data loading.

5.4 Modules

5.4.1 Introduction

A group of classes that handle a certain area of our domain is what could be called a module. Sometimes these modules have their own namespace, most often not (sometimes because the code pre-dates good support of namespaces by gcc). In any case, it does correspond with two physical directories in the source tree: include/module_name and src/module_name. Thus earth model related classes go in the EarthModel directories.

5.4.2 The separation

The separation of include and src is first of all a visibility issue. The include files can be 'seen' by other modules, the src files not. Conceptually, the separation is roughly interface versus implementation. Roughly, because small functions are often implemented in the header file.

Another separation that is very important is between UI- and 'Real Work' modules. No (direct) user interface work is done in the RW-modules. The amount of Real Work in the UI modules is minimised. Within the user interface part, there is a separation between basic UI (Qt-based in OpendTect) and 3D visualistion (OpenSceneGraph-based). Both types of user interface modules have a prefix: 'ui' and 'vis' respectively.

Making all these modules as opposed to just dumping everything in one big directory does have the effect that it becomes necessary to precisely know what's dependent on what. That's exactly what's described in the data/ModDeps.od file. This file is used by OpendTect to automatically load module libraries.

5.4.3 Real Work modules

First of all, there are the Basic, Algo and General modules. General depends on Algo, which in turn depends on Basic. The separation is a bit arbitrary, and the idea was that Basic would be tools also usable outside OpendTect. It's easy to find a counterexample like survinfo which was placed there to provide other Basic classes with good defaults.

In any case, Basic handles basic stuff like file-related, extra string utils, Ascii keyword-value files, positions (coords, inline/crossline), our own 'string' class the BufferString (not just a relict: it works better with C environments), sets: TypeSet, ObjectSet and BufferStringSet, OS dependent things like threads, stream opening, and a variety of basic algorithms.

Algo is for, well, Algorithmic stuff. General handles all sorts of things, like fast dynamic large N-D arrays (ArrayND etc.), the CBVS format for volumestorage, the IOObj, IOMan and other data store related classes, Translators enabling different formats, transforms and a few more things.

The domain-specific modules like Well, EarthModel, Seis etc. will be recognized by a geoscientist. There's also the Attribute and AttributeEngine, the seismic attribute modules, and the engine that does the work.

5.4.4 UI modules

For most of the RW-modules, there is a UI counterpart. This is made possible by the basic UI modules uiBase, uiTools and uiIo, the basic 3D visualisation module visBase and the basic combined stuff in uiOSG. On top of everything is the UI application logic in uiODMain.

The uiBase module is one of the two modules that access Qt services. Where uiOSG specifically handles the bridge bewteen Qt and OpenSceneGraph, uiBase is Qt only. These two are therefore isolation layers. uiBase's main task is to provide access to Qt widgets and implement the dynamic layout concept. In short, the OpendTect user interface was not painted with a paint tool, but rather programmed by attaching elements to each other. See the uiBase class documentation.

The uiTools module depends on uiBase only. It provides some general UI elements from the uiBase basic objects. Most notably, the uiGenInput class, providing generalised data input.

The uiIo module is intended mainly for object selection (in the data store).

The OpenSceneGraph-based visualisation services are made available in the visBase layer. Based on that, visSurvey delivers OpendTect-specific functionality.

It all comes together in the uiODMain module. Being the top-level module it is dependent on all other modules. To keep the amount of knowledge contained in this module low, much of the functionality is obtained from the 'UI Part servers'. For example, the uiSeisPartServer is the isolation class for all seismic-related user interface work. The uiODApplMgr object is only coordinating the flow of information between the various part servers.

5.5 Contributing

You can contribute to the enhancement of OpendTect either by:

• providing bug fixes or enhancements to the OpendTect source code following the usual Github Fork-Pull Request process.

• or independently by developing and releasing open source plugins from your own Github or equivalent repository. See the wmplugins repository as an example of this approach.

An overview of the design principles and preferred coding style/practices employed by dGB in the development of OpendTect are described in dGB's coding guide.

6 Principles and best practices in OpendTect coding

6.1 Introduction

Software engineering is a game of trade-offs. Performance vs generality, flexibility vs stability, priorities vs general goals, and so on, and so on. A good software engineer weighs all pros and cons and comes up with (near-)optimal solutions, often trying to get the best of everything. Of course, in fact, sometimes things can be completely ignored (e.g. in dialog-UI's performance is seldomly an issue).

Goals

When creating and maintaining software code, invariably one wonders what choice will come out optimally. To define 'optimal', we have to define the goals we want to maximise. The most obvious is:

(1) Total time spent (man-hours)

Less obvious, but also very important is:

(2) Total programming pleasure

The issue is that we software developers have to 'live' in the code, for many hours each day. Nothing is more dissatisfying than to have to go through code that looks like crap, even though the code may be working. Not being able to find what you need, is another issue. Too complex code, too. Finally, maybe an item not really fitting in the list, but still:

(3) Team-readyness

If you find pleasure in typing 15-level '?'-statements - so be it. If your entire team likes it there is no problem either. But chances are you will be crucified by your team members when they have to do anything with your code.

6.2 Requirements

6.2.1 Nice and neat

Good code should look good. You have to find joy in making the things you deliver look as good as (reasonably) possible, and as easy to understand as possible. Compare these two class definitions:

```
class SizeKeeper
{
public:
                SizeKeeper() : sz_(0)         {}

    int         size() const { return sz_; }
    void        setSize( int n ) { sz_ = n; }

protected:

    int         sz_;
};
```

and:

```
class X { public: X() : n(0) {}
protected: int n;
public: int N() const { return n; } void sn(int p) { n = p;
} };
```

The second class definition looks like crap, and is difficult to understand, especially when you imagine the code where the class is used. Maybe the first definition isn't that clear for everybody immediately at the start, but it's easy to see that once you get used to the style, it will be easy and fun to work with this kind of code.

For this to work, a team must agree on a style. The style characteristics are chosen so they match the requirements of esthetics/pleasure and time minimisation.

It may look like the second class definition has an advantage over the first in the time spent creating it. Nothing is more true. Time in software development is spent on many things, and actually typing the code is just a tiny component:

• Time spent typing

• Time spent thinking/designing/creating

• Time spent changing

• Time spent understanding

• Time spent reworking

• Time spent debugging

For most practical purposes, the influence of typing time on the total time spent can be neglected.

That leads to an important principle:

Rule (1): Make your code look good right from the start.

Waiting for a clean-up stage is a serious mistake. Already during creation of the software, from the very start, the effects of sloppy code will hit you where it hurts. Even if you have to re-type sections 10 times, it is better to have the code really neat at all times. Only then you can see that the re-working is necessary. The earlier you detect that constructions are not intuitive, logical, and easy to understand the earlier you detect that your code is actually bad.

6.2.2 Uniform

This is not a point to be taken lightly. Other team members will at some point have to change your code, other team members will at some point have to debug your code. Uniformity makes sure this is as easy as possible. Remember this: changing code you haven't made yourself is never easy, so do everything you can to help your team members. Moreover, changing code you've made some time ago is never easy, so you're even doing it for yourself.

The implications are simple although a lot of programmers have a lot of problems with it:

Rule (2): Make your code look just like all the other code.

Combining (1) and (2) could casually be described as: make sure all team members feel at home in your code at all times.

6.2.3 Simple and Easy

Any complex process can be broken up into simple steps, any complex object can be broken up into simple objects. Always consider yourself as publishing something that needs to be read by others. Take them by the hand and make it easy to understand what you are doing, and why you are doing it. Avoid repetitions, complex constructions or long lines. Make things compact if that will make things clearer, or uncompact if needed.

The best code you will ever make will invariably look as if it has cost hardly any time to make. Like good dancers make it look like there is no effort involved, an excellent solution always looks simple, compact and easy to use.

6.3 Explicit Rules

6.3.1 Introduction

The way we do things in OpendTect is not a 100% fixed body of rules. Moreover, we tend to say 'rather do this than that', or sometimes we change our point of view. Still, we almost unanimously agree on almost every issue. To lower the time to discover how we do things, next to going through lots of code, you can make use of the rules that follow.

6.3.2 OO and general rules

• Try to avoid pure implementation inheritance. Inheritance of 'mainly interface' is usually OK. In all cases, ask yourself whether there really is a 'isA' relation between the classes. Prefer delegation in any doubt.

• Be very aware of dependency management. Avoid using services from classes that were designed for something else. In doubt, split that class into the common part and the part that you are not interested in.

• Anything adding to the complexity has to be justified. Don't use patterns or other nifty tricks without a good reason. Certainly, there are often good reasons. Factories for example are almost always there for a good reason. But, always ask yourself: is it worth the extra effort? The simple alternative may not be as flexible, but do I really need that extra flexibility?

• Do not implement anything that isn't used (yet). Don't go for 'complete classes' or that kind of mumbo-jumbo. Figure out which methods are indispensible (like copy constructors) and then add functions when they are needed. Things that seem to be sure to be used tend to never be used, instead they add to the burden of the maintainer. Sometimes pre-cooked stuff is removed in a re-work without ever being used. On the other hand, if you think something is needed later, you need to design the interface in such a way that if necessary, it is not unnecessarily hard to add. 'Prepare, don't implement'.

• Jokes and surprises are not funny. They may seem to you at the time but they are not. A mildly ironic comment once a year should be enough.

6.3.3 C++ rules

• We do not use exceptions. Exceptions are the horror story of C++, try looking at C++ report, November-December 1994, Tom Cargill: "Exception handling: A false sense of security". There are more reasons, for example that you have to use a certain paradigm throughout: RAII (Not a bad principle but not always easy to do and never enforcable). If you want to I can explain a lot of those reasons by e-mail. The bottom line is: don't use exceptions. External software using exceptions must be isolated with try { } catch (...) {}.

• We are not using all the STL stuff and the std::string class. This is not because we don't like it, but more because we don't see the need. Using this is not a problem but will not work well with the rest of the system so in general the classes are not used. For external software try moving to our own classes as quickly as possible and beware of problems with exceptions.

• All code must be const-correct except in specific areas where that would not give any gain: there it's optional. Learn the subtleties of const in various places. Don't cast away const unless you are certain about it, consider the possibility you need to use 'mutable'. Caching variables etc. should always be declared mutable. In OD, GUI classes and classes working with legacy stuff can be non-const correct. We do make them const-aware, which means they smoothly work together with classes that are const-correct.

• Operator overloading can be used very sparingly, in situations of simple classes with absolutely trivial usage. In any doubt, don't use it. It does more harm (sometimes a lot more) than it returns benefit. Even the ubiquitous examples like matrix calcutations are almost surely better made with good old-fashioned method calls.

• In cases that you don't know whether a language feature can be used, do not give the feature the benefit of the doubt. You can always ask your team members first. A C++ language feature should only be used if you can prove that it is useful, clear, fitting in our style and not easily possible with other means.

• We increasingly try to use name spaces. In many places namespaces should have been used and this is a legacy problem which we want to gradually get rid of.

• Do not pollute with things that are not C++, like M$-windows directives. If absolutely unavoidable design a strategy to minimise the impact of these horrible things.

• Consider implementing in a header file only if unavoidable (templates), or:

• Is the implementation stable? If not, dependencies will trigger each time the implementation is changed.

• The implementation must be completely trivial or useful for a reader. In the latter case, it replaces comments with something that is fundamentally up-to-date.

• The space taken may not be huge - then implement in the .cc file anyway.

6.4 Semantical/typographical rules

First of all: the naming of classes, variables, namespaces, etc. is extremely important. You want to optimise understandability and compactness, in doubt always go for understandability. Naming should be as intuitive as possible. If you cannot find an intuitive name, consider the possibility that your design is not right. Well designed classes and methods hardly ever have non-intuitive names. If you are really convinced you're right but still you can't find anything intuitive, make sure you explain the meaning in comments.

• Classes and name spaces have a well-chosen name. Very well chosen. Do not rest before you have a name that really suits the class well. Name spaces tend to have short names, classes tend to be longer. If you cannot find a good name you probably have to split up or redefine the class. Typographically, every syllable of the class/namespace name starts with an upper case character.

• Class methods also need carefully designed names. First of all, we have adopted the early Smalltalk rules: * First syllable: all lower case * further syllables: start with upper case. Then, how the method is named is dependent on what it does. The rule is that the resulting code must read as if it is English text, and that it does what it says. most often `verbNoun` is OK. Bad are:

- `bool moderator()` - bool functions must be usable directly in '`if`' or '?' statements. Imagine '`if ( moderator() )`'. Depending on what it does, consider '`isModerator()`' or '`moderate()`'.
- `void wordChanger()` - a word changer could be an object but not a function. Consider '`changeWord()`' or '`changeWords()`' or a re-design.
- `int applesAndPears()` - what does this thing do? Make sure there is at least one verb.

• Variables are in lowercase. Class members should get an underscore at the end, further variables should generally be free of underscores. Special cases are Keys, 'hard' constants and Notifier names. There is a namespace '`sKey`' and there are variables '`sKeyXxx`' for key strings. Examples are '`sKey::Yes`' and '`sKeyTraceLength`'. Hard constants are like '`cMaxNrPatches`'. Notifiers are defined in Basic/callback.h.

• Macros are like constants but with prefix '`m`': '`mErrRet(msg)`'. As usual, inline funtions, constants and templates are preferred but macros are still indispensible in real C++.

6.5 Layout

No other subject brings up this many discussions. While it's simple: choose a policy and stick to it. The end result is what counts: readability, compactness, understandability. Thus all rules can be broken if it really helps those properties, but they rarely are.

• Indentation:

4 spaces per level, 8 spaces = one tab. Use tabs whenever possible, also inside a line.

• Alignment:

The maximum number of characters on one line is 80. So when you exceed this number, start on the new line with a couple of tabs. Align function arguments as much as possible.

```
MyClass::functionWithLongName( const char* arg1,
                               const char* arg2 ) const
```

When implementing functions in header files, align the implementations.

```
getPtr   ()      const            {    return      ptr;      }
getValue() const      { return val; }
```

• Braces '{ }':

On a line by themselves.

```
if                      (           b                     )
{
    stmt1                                                 ();
    stmt2                                                 ();
}
```

Single statements need not be braced.

```
if                    (          b                    )
    stmt1();
```

Two or three small statements should not be packed on one line with braces. Although that this is good for readability, it should be avoided due to that it is not good for debugging.

```
if                    (          b                    )
    { stmt(); return; }
```

• Parentheses:

Pad with a space on both sides, for outer parentheses. For inner parentheses, no spaces should be provided. When using `tr("<STRING>")`, no spaces should be used.

```
        if                  (          x                    )
if ( (x && y) || (z1 && z2) )
```

• Array brackets:

No space between array and first bracket. Pad index if that makes things more clear.

```
x = arr[0];
```

• Equality-type operators:

Pad with spaces, unless it really helps seeing the grouping.

```
if          (       a          ==          b          )
x = 15;
```

If more clear, use:

```
if ( c<d && e>f )
```

rather than:

```
if ( c < d && e > f )
```

• Semicolons:

Attach to last character of statement:

```
doIt                                                    ();
for ( int idx=0; idx<10; idx++ )
```

• '?'-statements:

Use only and always if the same thing must be done or used depending on a condition:

```
   return       isOK   ()       ?       10       :       20;
x      =      a      >      10      ?      10      :      a;
prTxt( isOK() ? "Yes" : "No" );
```

• Class declarations:

Just look at examples, but a nice template may be:

```
class                                                    Y;
class                                                    Z;

namespace                                                 X
{

class           A           :           public           B
{
public:
                          A   (   const      C&      c      )
                          :                    B          (c)
                          ,       var_   (0)               {}

    enum      Type                    {    T1,      T2      };
    void                                 setType       (Type);

    bool                  isNice ()   const         {  return
true;                                                     }
    bool                  isOK(float) const:
```

```
                            //!< will not handle values < 0
well!

    void                    doIt (int  base_ count,const  Y&);

protected:

    float                   var_;

private:

    void                    init                          ();
    friend                                  class          Z;

};
```

Remarks:

- The tab alignment can be 2, 3 or 4 tabs, dependent on the length of things.
- functions implemented immediately get a normal space padding for the arguments. Functions only declared get no padding for the arguments. Put variable names only if it adds to the understanding.
- Comments can help but can also make things a mess. Use them sparingly, only to clearly specify what a method does, or to indicate pre-conditions etc.

6.6 Adapting code

From now on, new code will be as described above. What to do if the code you're changing is not good according to these standards? That depends on the amount of work vs the amount of time vs the importance of the deviation.

The rules are:

• Make sure the code is, after you're done with it, 'internally consistent'. For example, when adding a member to a class with already 20 members without an underscore you will likely add a member without an underscore, rather than changing all the other members. In no case make your new member the only one with an underscore. That is even more confusing to the reader of the .cc code.

• If you doubt whether to re-do parts, give changing it the benefit of the doubt unless it's really horribly dangerous to do so. But the rule is: In doubt => Change.

• Functions or members that seem to be unused but non-trivial: always try removing them or flagging them with messages (e.g. copy constructors and the like will be made by the compiler after you remove them and then you still don't know whether they are used). An unused non-trivial fuction must be considered 'unbearable' (see next item).

• Some things are 'unbearable'. Unbearably bad naming. Spaces instead of tabs. Unused code. Replace this kind of stuff where you see it. If these things happen too much, consider asking the producer of the crap to remove it him/herself.

7 Class Documentation and other resources

- OpendTect Class Documentation
- OpendTect developers Google Group
- dgbpy framework documentation
- odpy framework documentation
- OpendTect Machine Learning Developers Discord Community

Glossary

C

Closed Source

Software that is released in binary form only. The commercial plugins to OpendTect are released as closed source extensions. Such extensions are only permitted if OpendTect is run under a commercial (or academic) license agreement.

G

GPL License

Gnu General Public License (http://www.gnu.org/licenses/gpl.html) is an open source license under which OpendTect can be run. The license allows redistribution of (modified) source code under the same licensing conditions (copy left principle). It is not allowed to combine the open source part with closed source plugins, which is why OpendTect is also licensed under a commercial license agreement and under an Academic license agreement.

O

Open Source

Software that is released with its source code. OpendTect is released as open source product that can be extended with closed source plugins. Such extensions are only permitted if OpendTect is run under a commercial (or academic) license agreement.