Plots

Volume 3

Ocean Software Development Framework for Techlog
Version 2023

SLB-Private

slb

Copyright © 2006-2023 Schlumberger. All rights reserved.

This work contains the confidential and proprietary trade secrets of Schlumberger and may not be copied or stored in an information retrieval system, transferred, used, distributed, translated or retransmitted in any form or by any means, electronic or mechanical, in whole or in part, without the express written permission of the copyright owner.

Trademarks & Service Marks

Schlumberger, the Schlumberger logotype, and other words or symbols used to identify the products and services described herein are either trademarks, trade names or service marks of Schlumberger and its licensors, or are the property of their respective owners. These marks may not be copied, imitated or used, in whole or in part, without the express prior written permission of Schlumberger. In addition, covers, page headers, custom graphics, icons, and other design elements may be service marks, trademarks, and/or trade dress of Schlumberger, and may not be copied, imitated, or used, in whole or in part, without the express prior written permission of Schlumberger. Other company, product, and service names are the properties of their respective owners.

An asterisk (*) is used throughout this document to designate a mark of Schlumberger.

Contents

Overview 1
Plot base class 3
  Plot domain object 3
  Plot signals 4
    PlotResized Signal 4
    PlotSelectedZoneChanged Signal 5
    PlotZoomChanged Signal 5
Logview 6
  Logview domain object 7
    SelectedTrackItemsChanged Signal 10
    ReferenceUnitChanged Signal 11
    SelectedTracksChanged Signal 12
    TrackCreated Signal 14
    MouseModeChanged Signal 16
    ScrollChanged Signal 17
    ZoomChanged Signal 17
    Logview and layout template 18
  Track domain objects 23
    NormalTrack domain object 23
    ReferenceTrack domain object 25
    ZonationTrack domain object 28
  TrackItem domain objects 30
    LineTrackItem Domain Object 35
    ArrayTrackItem Domain Object 38
    TraceArrayTrackItem Domain Object 43
    ArrayBHITrackItem Domain Object 46
    DateTimeTrackItem Domain Object 48
    TextTrackItem Domain Object 51
    AcousticTrackItem Domain Object 54
    AcousticWaveformTrackItem Domain Object 55
    AcousticMatrixTrackItem Domain Object 62
    DipTrackItem Domain Object 66

Schlumberger Private - Customer Use

SLB-Private

ImageTrackItem Domain Object 66
BHATrackItem Domain Object 67
WellSchematicTrackItem Domain Object 69
Track template 73
Area fills 75
Areafill Domain Object 75
SingleAreafill Domain Object 76
DoubleAreafill Domain Object 78
Logview display options 81
Plot group into a track 84
CrossPlotGroup domain object 84
DispersionPlotGroup domain object 88
Plots single well 90
CrossPlot and PlotScale Domain Objects 93
LinePlot Domain Object 103
CrossPlotArrayVariableArray Domain Object 107
CrossPlotArrayArray Domain Object 111
Histogram Domain Object 116
Plots multi well 126
MultiWellCrossPlot Domain Object 127
SelectedDatasetChanged Signal 131
Advanced plotting 133
Field3DView Domain Object 133
Regression 137
Regression Domain Object 137
Customize plots 140
GraphicsScene Domain Object 140
Create a GraphicsScene in a Plot 141
Create a GraphicsScene in a NormalTrack of a Logview 141
GraphicsScene display options 143
Domain object collections 144
GraphicsScene signals 144
ChartParameter Domain Object 150
GraphicsItem Domain Objects 155
Shape Domain Objects 156
Ellipse2d Domain Object 157

Line2d Domain Object 157
Rectangle2d Domain Object 158
Image2d Domain Object 159
Triangle2d Domain Object 160
PlotAnnotation2d Domain Object 161
PointsList Domain Objects 162
PointsCloud Domain Object 163
Polyline2d Domain Object 163
MarkerLabel Domain Object 164
GraphicsItem signals 166
Action Domain Object 170
Custom plots 178
CustomPlot Domain Object 179
ContainerPlot Domain Object 189

Overview

In Techlog the plot viewers allow you to compare multiple measurements made at a single reference. There are several types of plots splitted in different groups and accessible in Techlog through the Plot tab.

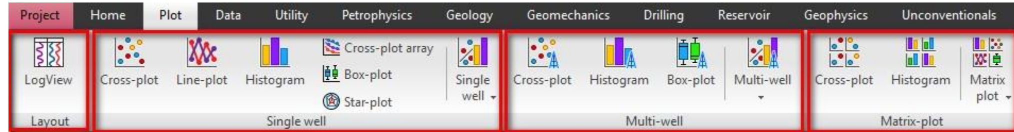


Figure 1 Plots in Techlog

The first one is the **Logview** in Layout group. **Logview** is the Techlog tool that allows you to display any variable present in the **Project browser**. The actions available to you in **Logview** are classified by type. The actions available to you in the dynamic toolbar depend on the object you select. The fixed toolbar and mouse mode always provide you with the same actions.

The rest of the plot tools are in the same menu as **Logview**. You can access those features from the **Plot** menu where three categories of plots are available:

- Single well plot
- Multi-well plot
- Matrix-plot

The most common plots are on the left side of the menu bar. More plot style options are available in the drop-down menu for each plot category.

Plot	Shortcut
Box-plot	Ctrl+Shift+P, B
Histogram	Ctrl+Shift+P, H
Pie-chart	Ctrl+Shift+P, H, P
Cross-plot	Ctrl+Shift+P, C
Line-plot	Ctrl+Shift+P, I
3D cross-plot	Ctrl+Shift+P, 3
Spectrum analyzer	Ctrl+Shift+P, S
Star-plot	Ctrl+Shift+P, U
Ternary-plot	Ctrl+Shift+P, T
Histogram variable vs. array	Ctrl+Shift+P, A, L
Cross-plot array vs. array	Ctrl+Shift+P, A, A
Cross-plot AVA	Ctrl+Shift+P, A, V, A

Figure 2 Plots available in drop-down menu of single well category

Single well plots available with Ocean:

- Cross-plot
- Cross-plot AVA (Array Variable Array)
- Cross-plot array
- Histogram

Schlumberger Private - Customer Use

SLB-Private

- Line plot

Multi well plots available with Ocean:

- Cross-plot

A matrix-plot called container plot is also available through Ocean to display several plots in only one view.

These viewers can incorporate several dimensions or variables in a single window, along with filters, charts, secondary variables, user-defined regressions, and equations.

All plots are linked to all other plots in your Techlog workspace, which means that any point selection you make on a plot is interactively displayed in other plots.

Others plots are more domains-oriented and they are available with Ocean as:

- Waveform plot in Geophysics domain
- Dispersion plot in Geophysics domain
- Borehole section plot in Geology domain
- Stereonet plot in Geology domain
- 2D well trajectory plot in Geology domain

SLB-Private

Plot base class

A plot in Techlog is any views displayed in the Techlog workspace. All the plots available with Ocean inherit from the **`Plot`** base class and do not support name.

A **`Plot`** object cannot be created directly from the **`Plot`** class but can be retrieved from plots collection returned by the **`plots()`** public method of the parent Workspace class (See the "Workspace Domain Object" section in *Ocean for Techlog Developer Guide - Basics*).

Plot domain object

The **`Plot`** class makes available the following generic properties for every plot:

- getter and setter for the window title
- getter and setter for the window size (width and height)
- ability to minimize and maximized the window
- ability through **`setHelpId`** method to link the **`Plot`** with some Techlog help content (HTML page) deployed at the plug-in storage level (plug-in folder)
- adjust the horizontal axis of the plot on variable data displayed on X axis
- adjust the vertical axis of the plot on variable data displayed on Y axis
- adjust both horizontal and vertical axes of the plot on variable data displayed on X and Y axes

See "How to add plug-in documentation to Techlog Help Center" tutorial in *OceanForTechlog.chm* file.

```
class Plot : public DomainObject
{
public:
 …
  QString windowTitle() const;
  void setWindowTitle(const QString &windowTitle);

  void resize(const QSize &size);
  void resize(int width, int height);

  int width() const;
  void setWidth(int width);

  int height() const;
  void setHeight(int height);

  bool isMinimized() const;
  void setMinimized(bool state);

  bool isMaximized() const;
  void setMaximized(bool state);

  void setHelpId(const QString &guid);
  QString helpId() const;
```

```
  void adjustHorizontalAxis();
  void adjustVerticalAxis();
  void adjustBothAxes();
  …
};
```

The **Plot** class makes available the following domain object collection for every plot:

- return the collection of graphics scenes in the chart area
- return the collection of annotations in the plot
- return the collecton of custom actions added to the mouse bar, tool bar or context menu of the plot

```
class Plot : public DomainObject
{
public:
 …
  const DomainObjectCollection<GraphicsScene> graphicsScenes()
  const;
  const DomainObjectCollection<PlotAnnotation2d> annotations()
  const;
  const DomainObjectCollection<Action> actions() const;
  …
};
```

Plot signals

A **Plot** object can listen to some generic events subscribing to the signals of the **Plot** class displayed below.

```
class Plot : public DomainObject
{
public:
 …
 enum EventType
 {
   PlotResized,
   PlotSelectedZoneChanged,
   PlotZoomChanged
 };
}
```

Those signals are emitted whenever:

- **PlotResized** – the plot is resized
- **PlotSelectedZoneChanged** – the selected zones in the plot for a given zonation have changed
- **PlotZoomChanged** – the zoom of the plot is changed

PlotResized Signal

The **PlotResized** signal includes an argument that gives the **Plot** instance the slot was connected to through the sender method inherited from the **SignalArgs** base class. This argument is modeled in Ocean for Techlog through the **PlotResizedArgs** class.

```
class PlotResizedArgs : public SignalArgsT<Plot>
{
};
```

PlotSelectedZoneChanged Signal

The **PlotSelectedZoneChanged** signal includes an argument that gives the **Plot** instance the slot was connected to through the sender method inherited from the **SignalArgs** base class. This argument is modeled in Ocean for Techlog through the **PlotSelectedZoneChangedArgs** class.

```
class PlotSelectedZoneChangedArgs :
public SignalArgsT<Plot>
{
};
```

PlotZoomChanged Signal

The **PlotZoomChanged** signal includes an argument that gives the **Plot** instance the slot was connected to through the sender method inherited from the **SignalArgs** base class. This argument is modeled in Ocean for Techlog through the **PlotZoomChangedArgs** class that gives to the plug-in the new display limits of the plot.

```
class PlotZoomChangedArgs :
public SignalArgsT<Plot>
{
  double minX() const;
  double maxX() const;
  double minY() const;
  double maxY() const;

};
```

Logview

Logview can display any type of data, including log, zone, core image, seismic, array data, and core measurements.

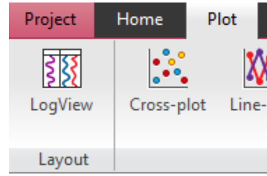


Figure 3 Logview under the Plot tab

A Techlog end-user can open a new Logview window by clicking on Logview under the Plot tab. Ocean provides an API to create programmatically an instance of the Logview window through the `Logview` class.

To display data in a Logview, the user drags one or more variables from the Project browser into a Logview window. You can have several tracks in a Logview containing themselves several track items. There is only one variable displayed per track item.

In the following screenshot a track item displays gamma ray variable data into the first track of the Logview and in the second track two track items display respectively neutron and density variables data.

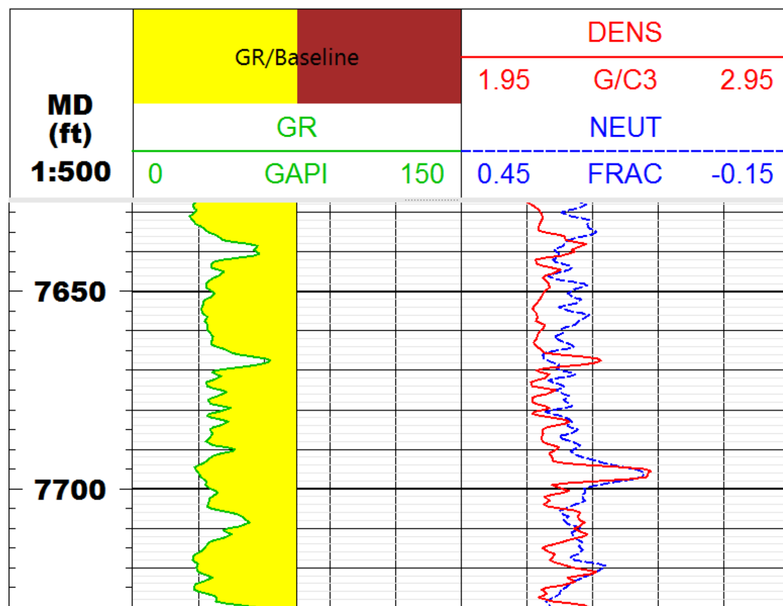


Figure 4 Logview, tracks and track items

Each of these elements is modeled in Ocean through differents domain objects that can be represented schematically as follows:

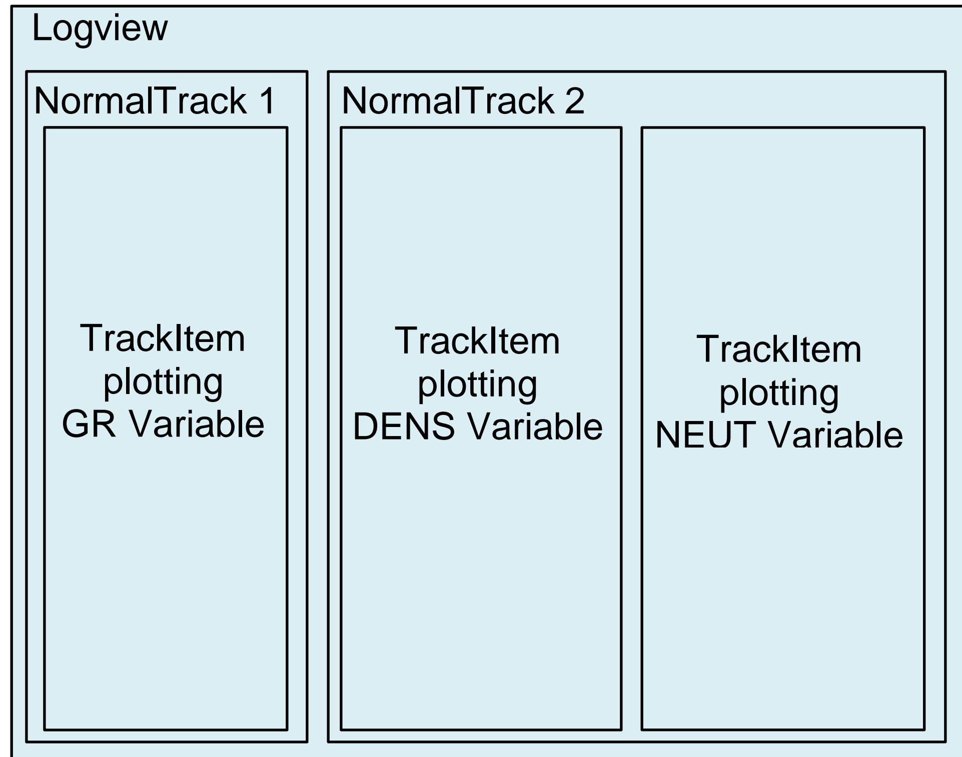


Figure 5 Logview, Track and TrackItem Ocean domain objects

Logview domain object

The **`Logview`** class inherits from the **`Plot`** class and does not support name. A **`Logview`** object is instancied through **`create`** static method of the class with the parent **`Workspace`** object as argument. The uniqueness of a **`Logview`** object is handled by the system. The only way to retrieve a **`Logview`** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

A **`Logview`** can also be added to a container plot (matrix-plot) using the dedicated **`create`** static method.

See the “ContainerPlot Domain Object” section on page 189 for more information on how to create a Logview into a container plot.

```
class Logview : public Plot
{
public:
 static Logview create(Workspace workspace);

 static Logview create(const ContainerPlotPosition
   &containerPlotPosition);
 …
 DomainObjectCollection<TrackItem> selectedTrackItems() const;

 void setSelectedTrackItems(const QList<TrackItem>
                                 &trackItemsToSelect);

 DomainObjectCollection<Track> selectedTracks() const;
```

```
 const DomainObjectCollection<Track> tracks() const;

 void moveTrack( Track& track, int position );
 int trackPosition( const Track& track ) const;
 …
 QString referenceUnit() const;
 void setReferenceUnit(const QString &unit);
 …
 enum EventType
 {
   SelectedTrackItemsChanged,
   ReferenceUnitChanged,
   SelectedTracksChanged,
   TrackCreated,
   MouseModeChanged,
   ScrollChanged,
   ZoomChanged
 };
}
```

From the **Logview** domain object you can:

- get the collection of **TrackItem** domain objects selected in the **Logview**
- select well logs displayed in the **Logview** through **TrackItem** objects

Note: Only derived **TrackItem** objects exposed today with Ocean are selectable using the **setSelectedTrackItems** function.

- get and set the position of the **Track** in the **Logview** through **trackPosition** and **moveTrack** functions

Note: Those functions can only be applied to normal tracks returned by the **tracks** function. See "NormalTrack domain object" section on page 23 for more information about normal tracks.

- get and set the unit to be used ib Logview reference
  - ignored if the unit is not compatible with the existing one
  - ignored if **Logview** has no reference track
  - it will have no effect if this **setReferenceUnit** is called before releasing the **Lock** where the **Logview** is being created
- and much more …

A **Logview** object can listen to events subscribing on the following signals:

Those signals are emitted whenever:

- **SelectedTrackItemsChanged** – signal emitted whenever the **Logview::selectedTrackItems** domain object collection has changed
- **ReferenceUnitChanged** – signal emitted whenever the **referenceUnit** property of the **Logview** has changed
- **SelectedTracksChanged** – signal emitted whenever the **Logview::selectedTracks** domain object collection has changed

- **TracksCreated** – signal emitted when a new Track is added to the **Logview::tracks** domain object collection
- **MouseModeChanged** – signal emitted when the MouseMode enum value is changed
- **ScrollChanged** – signal emitted when the user is scrolling into the Logview or changing the Logview scale.
- **ZoomChanged** – Signal emitted when the user is zooming in / out into the Logview

To include signal arguments and declare slot receivers in the activity header file:

```
#include "tsdkselectedtrackitemschangedargs.h"
#include "tsdklogviewreferenceunitchangedargs.h"
#include "tsdkselectedtrackschangedargs.h"
#include "tsdktrackcreatedargs.h"
#include "tsdkmousemodechangedargs.h"
#include "tsdkscrollchangedargs.h"
#include "tsdkzoomchangedargs.h"
```

```
private slots:
  void onSelectedTrackItemsChanged(
  const Slb::Ocean::Techlog::SelectedTrackItemsChangedArgs
&args);
  void onReferenceUnitChanged(
  const Slb::Ocean::Techlog::LogviewReferenceUnitChangedArgs
&args);
  void onSelectedTracksChanged(
  const Slb::Ocean::Techlog::SelectedTracksChangedArgs &args);
  void onTrackCreated(
  const Slb::Ocean::Techlog::TrackCreatedArgs &args);
  void onMouseModeChanged(
  const Slb::Ocean::Techlog::MouseModeChangedArgs &args);
  void onScrollChanged(
  const Slb::Ocean::Techlog::ScrollChangedArgs &args);
  void onZoomChanged(
  const Slb::Ocean::Techlog::ZoomChangedArgs &args);
```

Logview domain object can subscribe to those signals as follow:

```
void activity::createLogviewSignals(Logview logview)
{
  logview.connect(Logview::SelectedTrackItemsChanged,this,
  SLOT(onSelectedTrackItemsChanged(
  const Slb::Ocean::Techlog::SelectedTrackItemsChangedArgs
&)));

  logview.connect(Logview::ReferenceUnitChanged,this,
  SLOT(onReferenceUnitChanged(
  const Slb::Ocean::Techlog::LogviewReferenceUnitChangedArgs
&)));
```

```
  logview.connect(Logview::SelectedTracksChanged,this,
  SLOT(onSelectedTracksChanged(
  const Slb::Ocean::Techlog::SelectedTracksChangedArgs &)));

  logview.connect(Logview::TrackCreated,this,
  SLOT(onTrackCreated(const
Slb::Ocean::Techlog::TrackCreatedArgs &)));

  logview.connect(Logview::MouseModeChanged,this,
  SLOT(onMouseModeChanged(
  const Slb::Ocean::Techlog::MouseModeChangedArgs &)));

  logview.connect(Logview::ScrollChanged,this,
  SLOT(onScrollChanged(
  const Slb::Ocean::Techlog::ScrollChangedArgs &)));

  logview.connect(Logview::ZoomChanged,this,
  SLOT(onZoomChanged(
  const Slb::Ocean::Techlog::ZoomChangedArgs &)));

  _action1 = Action::create("Action1", ActionTypeToggleGroup,
   ActionLocationMouseBar, logview);
  _action2 = Action::create("Action2", ActionTypeToggleGroup,
  ActionLocationMouseBar, logview);
}
```

SelectedTrackItemsChanged Signal

The **`SelectedTrackItemsChanged`** signal includes an argument that gives the **`TrackItem`** domain objects that have been added or removed from **`selectedTrackItems`** domain object collection. This argument is modeled in Ocean for Techlog through the **`SelectedTrackItemsChangedArgs`** class.

```
class SelectedTrackItemsChangedArgs :
public SignalArgsT<Logview>
{
public:
 …
 DomainObjectCollection<TrackItem> newSelectedTrackItems()
 const;
 DomainObjectCollection<TrackItem> unselectedTrackItems()
 const;
};
```

The slot handler accesses the needed information from the signal argument.

```
void activity::onSelectedTrackItemsChanged(
const Slb::Ocean::Techlog::SelectedTrackItemsChangedArgs &args)
{
```

```
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  // collection of new track items selected in the Logview
  DomainObjectCollection<TrackItem> newSelectedTrackItems =
  args.newSelectedTrackItems();
  foreach (TrackItem newSelectTrackItem, newSelectedTrackItems)
    qWarning() << "new selected variable: "
    << newSelectTrackItem.findVariable().name();

  // collection of track items unselected in the Logview
  DomainObjectCollection<TrackItem> unselectedTrackItems =
  args.unselectedTrackItems();
  foreach (TrackItem unselectTrackItem, unselectedTrackItems)
    qWarning() << "unselected variable: "
    << unselectTrackItem.findVariable().name();

  lock.release();
}
```

ReferenceUnitChanged Signal

The **`ReferenceUnitChanged`** signal includes an argument that gives the former and new unit used by the **`Logview`** as reference. This argument is modeled in Ocean for Techlog through the **`LogviewReferenceUnitChangedArgs`** class.

```
class LogviewReferenceUnitChangedArgs :
public SignalArgsT<Logview>
{
public:
 …
 const QString & oldUnit() const;
 const QString & newUnit() const;
};
```

The slot handler accesses the needed information from the signal argument.

```
void activity::onReferenceUnitChanged(
const Slb::Ocean::Techlog::LogviewReferenceUnitChangedArgs
&args)
{
  Logview logview = args.sender();
  QString oldUnit = args.oldUnit();
  QString newUnit = args.newUnit();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, logview);

  qWarning() << "Unit of Logview " << logview.windowTitle()
    << " changed from " << oldUnit << " to " << newUnit;
```

```
  lock.release();
}
```

SelectedTracksChanged Signal

The **`SelectedTracksChanged`** signal includes an argument that gives the **`Track`** domain objects that have been added or removed from **`selectedTracks`** domain object collection. This argument is modeled in Ocean for Techlog through the **`SelectedTracksChangedArgs`** class.

```
class SelectedTracksChangedArgs :
public SignalArgsT<Logview>
{
public:
 …
 DomainObjectCollection<Track> newSelectedTracks() const;
 DomainObjectCollection<Track> unselectedTracks() const;
};
```

The following example shows how to enable or disable some actions in the mousebar following tracks selection and unselection:

```
void activity::onSelectedTracksChanged(
const Slb::Ocean::Techlog::SelectedTracksChangedArgs &args)
{
  Logview logview = args.sender();
  if (args.newSelectedTracks().count() != 0)
  {
    Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
    Track newSelectedTrack = args.newSelectedTracks().at(0);

    if (!newSelectedTrack.isA<NormalTrack>())
    {
      lock.release();
      return;
    }
    NormalTrack normalTrack =
      newSelectedTrack.cast<NormalTrack>();

    foreach(TrackItem trackItem, normalTrack.trackItems())
    {
      if (trackItem.findVariable().name() == "GAMM")
      {
        _action1.setEnabled(true);
        break;
      }

      if (trackItem.findVariable().name() == "NEUT")
      {
```

```
        _action2.setEnabled(true);
        break;
      }
    }
    lock.release();
  }

  if (args.unselectedTracks().count() != 0)
  {
    DomainObjectCollection<Track> unselectedTracks =
    args.unselectedTracks();
    Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

    foreach(Track unselectedTrack, unselectedTracks)
    {
      if (unselectedTrack.isA<NormalTrack>())
      {
        NormalTrack normalTrack =
          unselectedTrack.cast<NormalTrack>();
        foreach(TrackItem
          trackItem,normalTrack.trackItems())
        {
          if (trackItem.findVariable().name() == "GAMM")
            _action1.setEnabled(false);

          if (trackItem.findVariable().name() == "NEUT")
           _action2.setEnabled(false);
        }
      }
    }
    lock.release();
  }
}
```

The following screenshot shows the action 1 is disabled when the track 1 is unselected and the action 2 is enabled when the track 2 is selected:

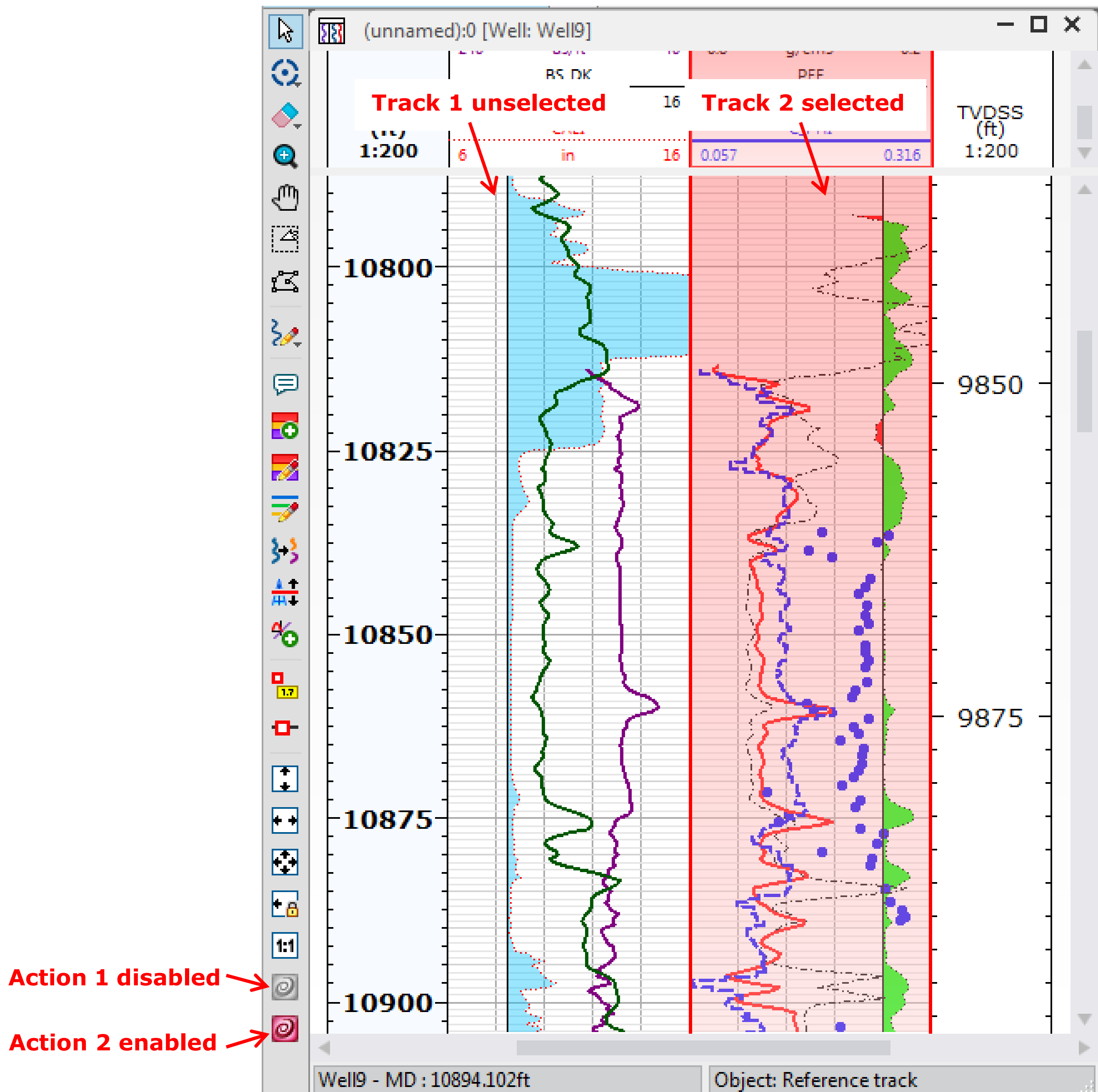


Figure 6 Track selection and unselection notification

TrackCreated Signal

The **TrackCreated** signal includes an argument that gives the new **Track** domain object that has been added to the **tracks** domain object collection. This argument is modeled in Ocean for Techlog through the **TrackCreatedArgs** class.

```
class TrackCreatedArgs : public SignalArgsT<Logview>
{
public:
 …
 Track createdTrack()const;
```

```
};
```

The following slot handler example shows that you need to lock the Logview before to get the new track created from the signal argument:

```
void activity::onTrackCreated(
const Slb::Ocean::Techlog::TrackCreatedArgs &args)
{
  Logview logview = args.sender();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, logview);

  Track track = args.createdTrack();

  qWarning() << "track created: " << track.name();

  lock.release();
}
```

Note: The **`TrackCreated`** signal isn't triggered when track types that aren't exposed with Ocean are added to the **`Logview`** as "Annotation creation track" and "Separation track".

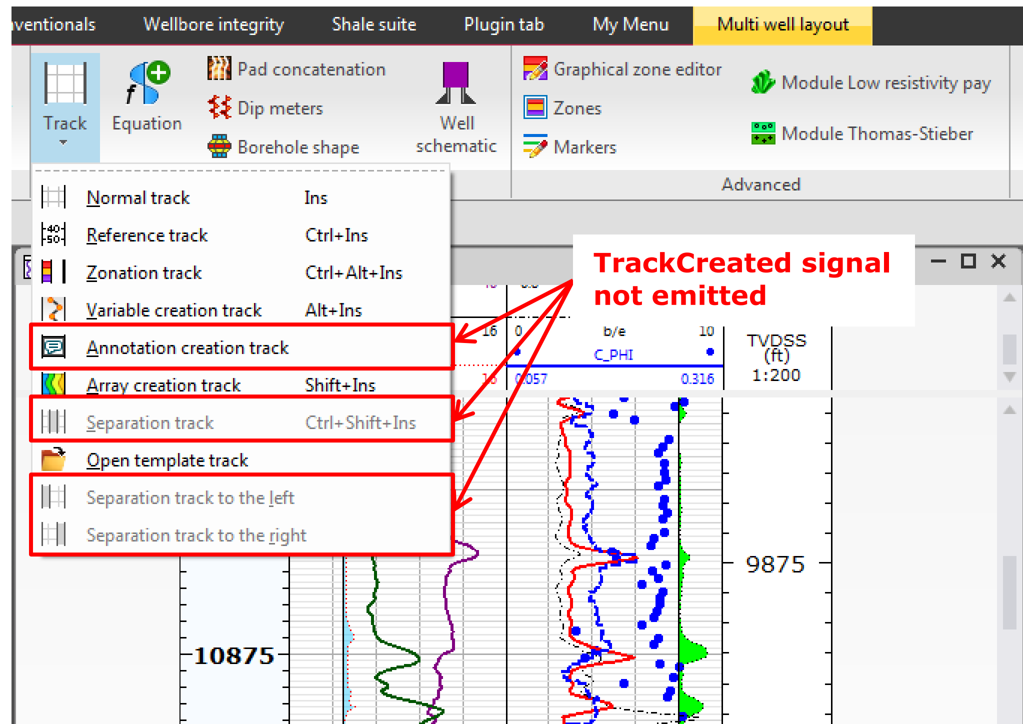


Figure 7 TrackCreated signal not emitted for some track types

MouseModeChanged Signal

The **MouseModeChanged** signal includes an argument that gives the current and previous **Logview MouseMode**. This argument is modeled in Ocean for Techlog through the **MouseModeChangedArgs** class.

```
class MouseModeChangedArgs : public SignalArgsT<Logview>
{
public:
 …
 const MouseMode currentMouseMode() const;
 const MouseMode previousMouseMode() const;
};
```

The slot handler accesses the needed information from the signal argument.

```
void activity::onMouseModeChanged(
const Slb::Ocean::Techlog::MouseModeChangedArgs &args)
{
  qWarning() << "Current Mouse Mode is : " <<
  ArgChecker::toString(args.currentMouseMode());

  qWarning() << "Previous Mouse Mode is : " <<
  ArgChecker::toString(args.previousMouseMode());
}
```

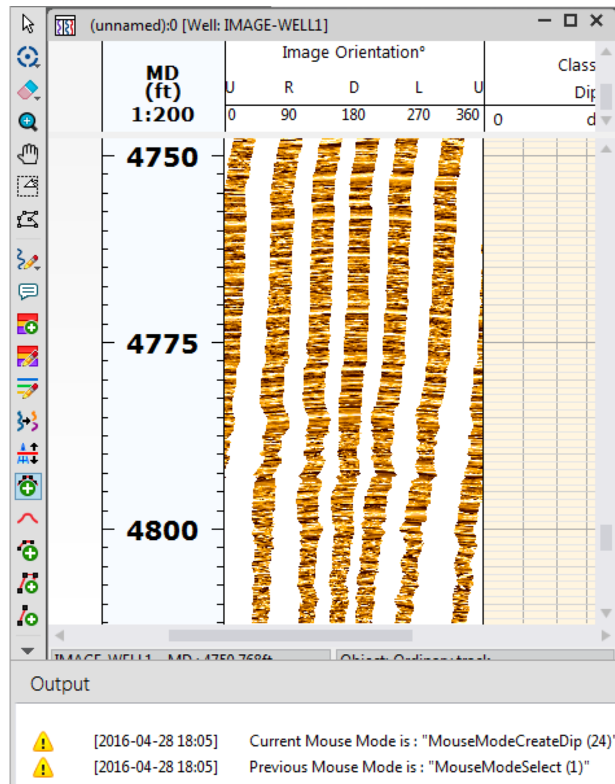


Figure 8 Logview mouse mode changed

ScrollChanged Signal

The **ScrollChanged** signal includes an argument that gives the scrolling delta and orientation (horizontal or vertical). This argument is modeled in Ocean for Techlog through the **ScrollChangedArgs** class.

```
class ScrollChangedArgs : public SignalArgsT<Logview>
{
public:
 …
 int scrollDelta() const;
 Qt::Orientation orientation() const;
};
```

ZoomChanged Signal

The **ZoomChanged** signal includes an argument that gives the new min and max vertical Logview window limits after the zoom is applied. Those values are returned in the Logview reference unit. This argument is modeled in Ocean for Techlog through the **ZoomChangedArgs** class.

```
class ZoomChangedArgs : public SignalArgsT<Logview>
{
public:
 …
 double minY() const;
 double maxY() const;
};
```

Shown below is an example:

```
void activity::onZoomChanged(
const Slb::Ocean::Techlog::ZoomChangedArgs &args)
{
  Logview logview = args.sender();
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, logview);

  qWarning() << "New zoom window: minY = " << args.minY()
    << ", maxY = " << args.maxY() << "(" + logview.referenceUnit()
    +")";

  lock.release();
}
```

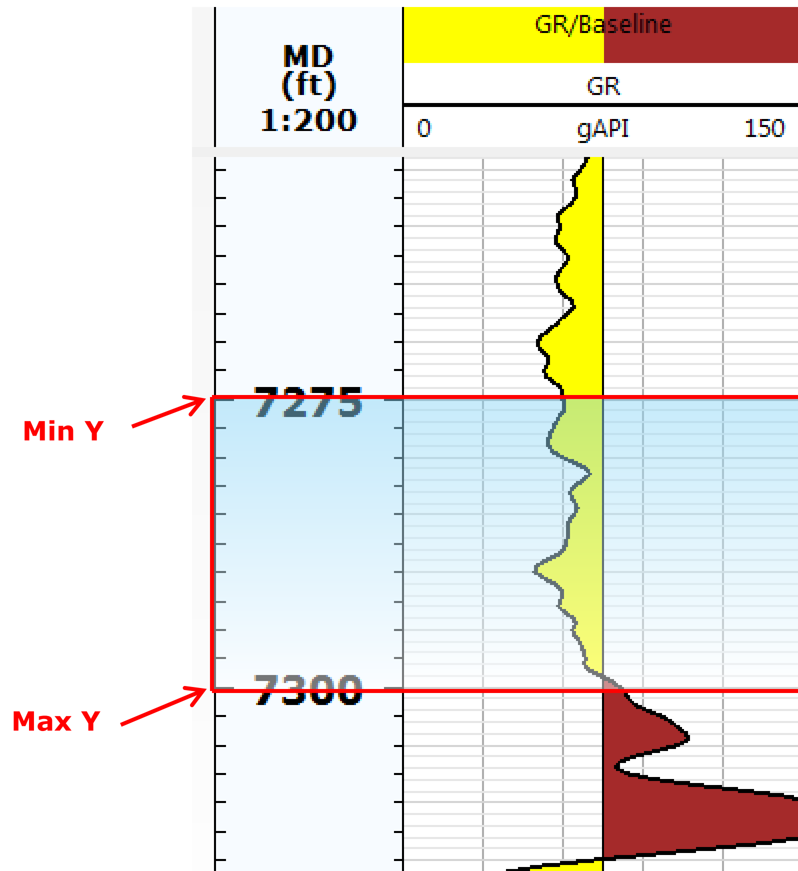


Figure 9 Logview zoom changed

Logview and layout template

In the **`Logview`** class a create static method is available to instantiate a **`Logview`** object passing the parent **`Workspace`** and a **`LogviewTemplateDataBinding`** instance.

A **`Logview`** can also be created by template into a container plot (matrix-plot) using the dedicated **`create`** static method.

See the “ContainerPlot Domain Object” section on page 189 for more information on how to create a Logview by template into a container plot.

```
class Logview : public Plot
{
public:
  …
  static Logview create(Workspace workspace, const
    LogviewTemplateDataBinding &logviewTemplateDataBinding);

  static Logview create(const ContainerPlotPosition
    &containerPlotPosition, const LogviewTemplateDataBinding
    &logviewTemplateDataBinding);
  …
}
```

The **LogviewTemplateDataBinding** contains the layout template saved at a storage level and the well, the list of wells, the dataset or the list of datasets to apply to this template.

```
class LogviewTemplateDataBinding
{
public:
  LogviewTemplateDataBinding(const LogviewTemplate
  &logviewTemplate, const Well &well);

  LogviewTemplateDataBinding(const LogviewTemplate
  &logviewTemplate, const QList<Well> &wells);

  LogviewTemplateDataBinding(const LogviewTemplate
  &logviewTemplate, const Dataset &dataset);

  LogviewTemplateDataBinding(const LogviewTemplate
  &logviewTemplate, const QList< Dataset > &datasets);
}
```

In Techlog after setting a layout for a well, you can save the layout as a template.

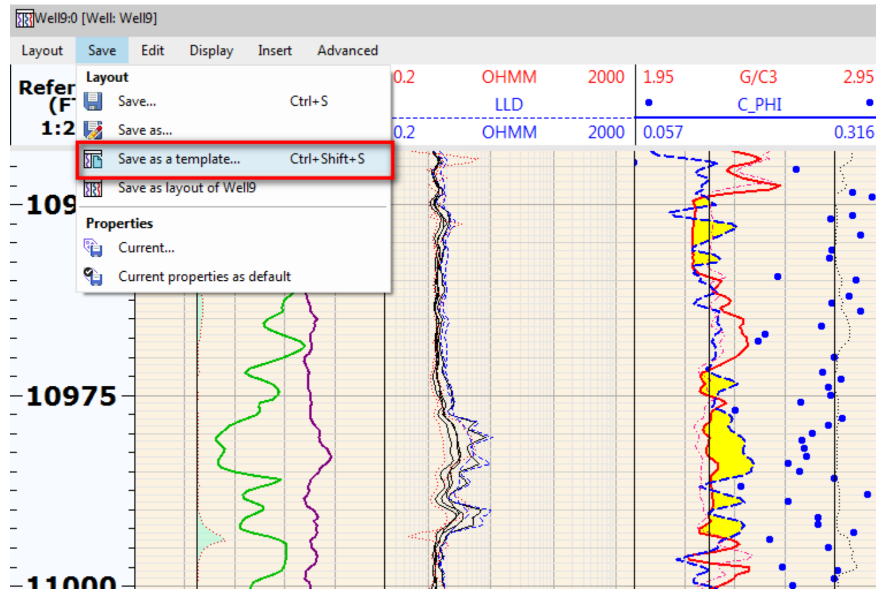


Figure 10 Layout saved as a template

Note: The difference between a layout and a template is that a template saves only the content of a layout, whereas a layout saves specific variables and the complete display.

You will specify the template name and at the level where is stored the template, modeled in Ocean with **LogviewTemplate** class and enum class **StorageLevel**.

```
class LogviewTemplate
{
public:
  static LogviewTemplate get(const StorageLevel level, const
  QString &name);
  static bool exists(const StorageLevel level, const QString
  &name);
```

```
  StorageLevel level() const;
  QString name() const;
   …
}
```

```
enum StorageLevel
{
  StorageLevelProject,
  StorageLevelCompany,
  StorageLevelUser,
  StorageLevelTechlog,
  StorageLevelPlugin
};
```

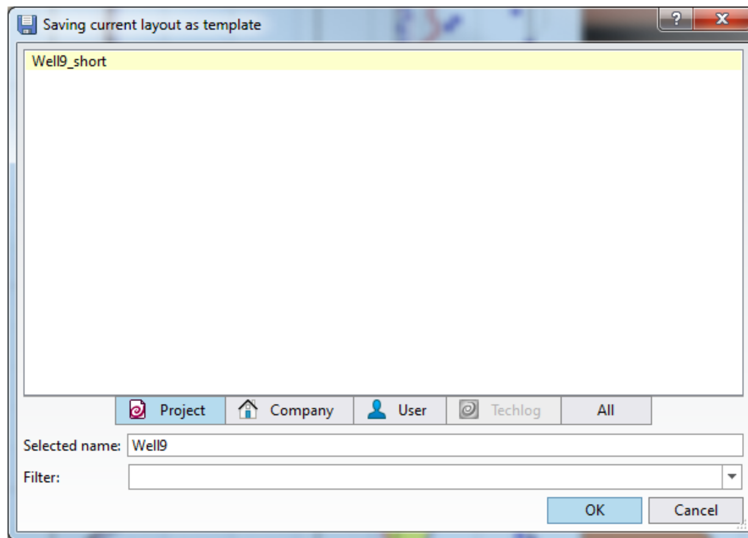


Figure 11 Techlog storage levels

Ocean introduces an additional level which is the plug-in level and the ability to provide layout templates in the plug-in folder. At the plug-in level the layout template has to be stored in a directory named "LayoutTemplates" closed to the plug-in dll in the plug-in folder.

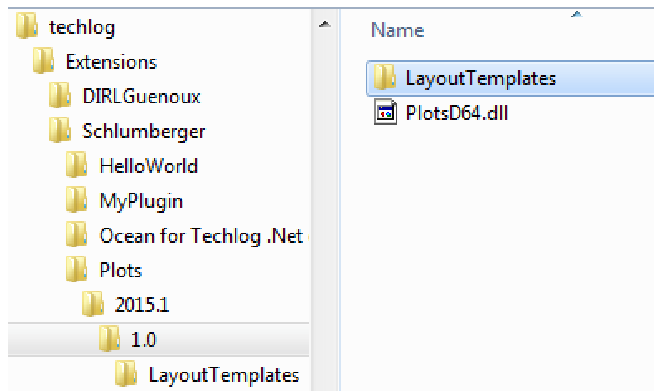


Figure 12 LayoutTemplates folder that contains layout templates at the plug-in level

The layout template is saved to the Project browser under Layouts. You can retrieve a layout template in the project later, double click on the template and decide to apply the layout template as well template or as a dataset template.

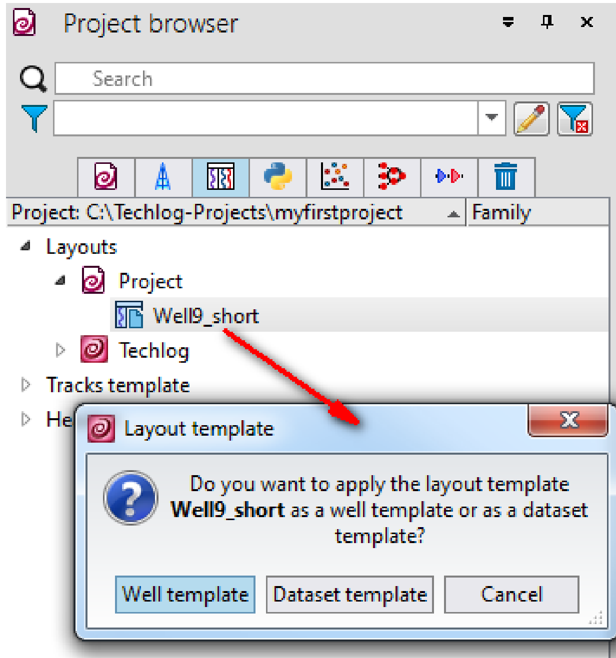


Figure 13 Layout template stored in the project browser at the project level

A Logview layout opens with the well data displayed. From the "Layout" menu you can apply it to dataset(s) or well(s) in the same layout or to individual layouts.

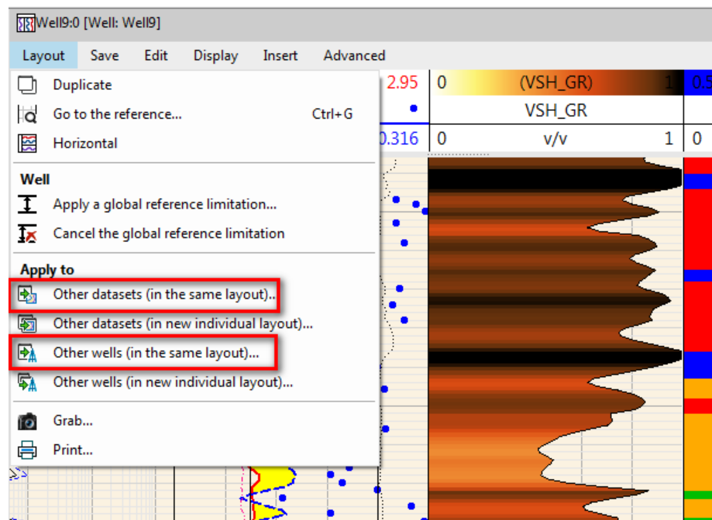


Figure 14 Apply layout to other dataset(s) or other well(s)

Ocean through the **`Logview::create`** methods passing a **`LayoutTemplate`** instance (containing template name and storage level) allows you to display the logs saved in the template for different wells or datasets in the same layout returning a unique **`Logview`** domain object instance. This multi-well-layout or multi-dataset-layout is also called cross-section.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
```

```
Project project = Session::current().mainProject();

Workspace workspace = Session::current().currentWorkspace();

// Apply the template for all the wells in the project
QList<Well> wells = project.wells().toList();

if (!LogviewTemplate::exists(StorageLevelProject,
"Well9_short"))
{
  lock.release();
  return;
}

LogviewTemplate logviewTemplate =
LogviewTemplate::get(StorageLevelProject, "Well9_short");

LogviewTemplateDataBinding templateDataBinding =
LogviewTemplateDataBinding(logviewTemplate, wells);

Logview crossSection = Logview::create(workspace,
templateDataBinding);

lock.release();
```

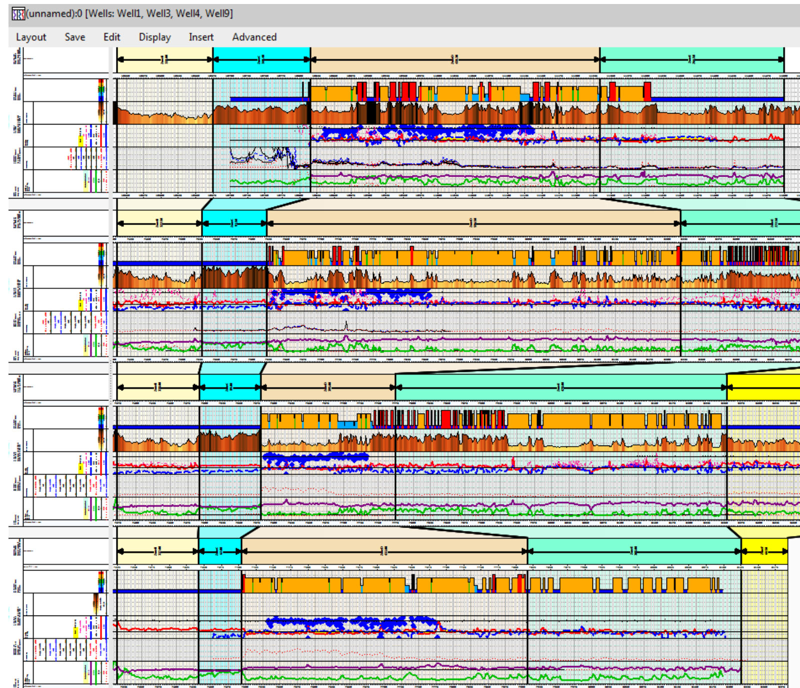


Figure 15 multi-well-layout or cross-section

SLB-Private

Track domain objects

Ocean exposes the ability to add different types of tracks to the **Logview**.

1. Create normal tracks (or ordinary tracks) which are tracks where variables are displayed. When you drag a variable into a layout, the variable is automatically displayed in a normal track.
2. Create a reference track that displays the reference variable of the dataset (MD reference depth by default) or if a reference dataset (index dataset) is present in the well other reference datum can be displayed in the track as TVD, TVDSS and TVDBM.
3. Create a zonation track that displays correlation line between wells. By default, the zonation track displays the selected zones of the current zonation dataset. However, the zonation track might display other zonation datasets existing in the database.

These tracks are handled by different Ocean classes grouped under the **Track** base class. The class diagram shows these different classes:

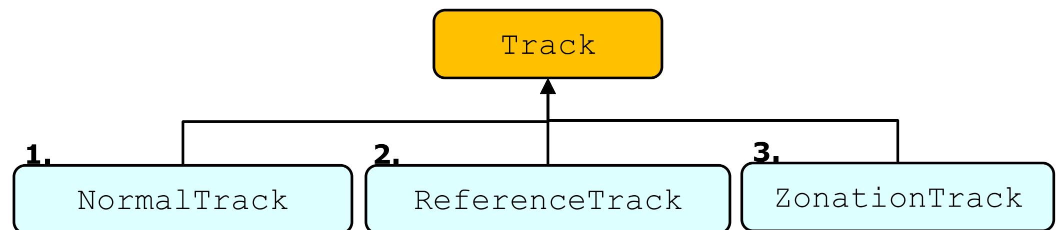


Figure 16 Track class diagram

The **Track** class holds properties common to all tracks that you can add to the **Logview**.

```
class Track : public DomainObject
{
public:
  Well findWell() const;
  Well well() const;
  float width() const;
  void setWidth(const float value);
  Logview logview() const;
  …
};
```

The **Track** class allows you to:

- get the **Well** of **Variable** data displayed into the **Track**
- get and set the **width** of the **Track**
- get the parent **Logview** of the **Track**

NormalTrack domain object

A **NormalTrack** is added to the **Logview** to display data. A **NormalTrack** domain object can be enumerated from a **Logview** instance through the **normalTracks** domain object collection.

SLB-Private

```
class Logview : public DomainObject
{
public:
  DomainObjectCollection<NormalTrack> normalTracks() const;
  …
};
```

In this domain object collection a **NormalTrack** has to be unique by its name so this is recommended to check for normal track name existence in the parent **Logview** before creating it.

```
class NormalTrack : public Track
{
public:
  static NormalTrack create(const QString& name,
   Logview logview);

  const DomainObjectCollection<TrackItem> trackItems() const;

  void setHorizontalGridDisplay(const TrackGridDisplay
   &display);
  void setVerticalGridDisplay(const TrackGridDisplay &display);
  TrackGridDisplay horizontalGridDisplay() const;
  TrackGridDisplay verticalGridDisplay() const;

 enum EventType
 {
   TrackItemCreated
 };

  …
};
```

A **NormalTrack** domain object can listen to the **TrackItemCreated** event that allows to be notified when a **TrackItem** is added to the **NormalTrack**.

The **NormalTrack** object does not display directly the data but **Variable** data are plotted in the **NormalTrack** through a **TrackItem** object.

See “TrackItem domain objects” section on page 30 for more information on how to plot a **Variable** through a **TrackItem** in a **Track** of the **Logview**.

The **NormalTrack** class holds functions to manage **Logview** track grid appearance. You can modify grid track settings (horizontal / vertical grid below / above curves or no grid) and/or query the current grid settings.

The **TrackItemCreated** signal includes an argument that gives new **TrackItem** that has been created in the parent **NormalTrack**. This argument is modeled in Ocean for Techlog through the **TrackItemCreatedArgs** class.

```
class TrackItemCreatedArgs : public SignalArgsT<Track>
{
public:
 …
 TrackItem newTrackItem() const;
};
```

ReferenceTrack domain object

A **ReferenceTrack** is automatically added to the **Logview** when a first **NormalTrack** that plots **Variable** data is created in the **Logview**. This **ReferenceTrack** displays the reference **Variable** of the dataset.

The reference tracks can be enumerated from a **Logview** instance through the **referenceTracks** domain object collection.

```
class Logview : public DomainObject
{
public:
  DomainObjectCollection<ReferenceTrack> referenceTracks()
   const;
  …
};
```

A **ReferenceTrack** can be created and then added to the **referenceTracks** domain object collection. To create a **ReferenceTrack** you can use the **create** static method passing a name that has to be unique in the **referenceTracks** collection and a **Well**. This **Well** parameter means that a **ReferenceTrack** can't be added to the **Logview** if there isn't a **NormalTrack** that already exists in the **Logview** and plotting **Variable** data from this parent **Well**.

```
class ReferenceTrack : public Track
{
public:
  static ReferenceTrack create(const QString &name,
   const Logview &logview, const Well &well);

  void setReferenceSpace(const Family &referenceSpace);
  Family referenceSpace() const;
  QList<Family> possibleReferenceSpaces();
  ReturnValue<bool> canSetReferenceSpace(const Family
   &referenceSpace);

  void setReferenceUnit(const Unit &unit);
  Unit referenceUnit() const;
  …
};
```

The **ReferenceTrack** is created with a defaut **referenceSpace** that corresponds to the reference family of the reference **Variable** of the dataset. This dataset is the dataset that contains the first **Variable** plots to the **Logview** through a **NormalTrack** for the given **Well** passed to the **ReferenceTrack create** static function.

In the example below the neutron **Variable** is the first variable added to a **NormalTrack** in the **Logview**. This **Variable** comes from the Well1.LQC dataset with a TVDSS (True Vertical Depth) reference **Variable.** The gamma ray and density **Variable** added in second and third tracks are from the Well1.DATAFULL dataset with a MD (Measured Depth) reference **Variable.** The **ReferenceTrack** variable takes here the **referenceSpace** of the Well1.LQC dataset.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
```

```
Project project = Session::current().mainProject();

Workspace workspace = Session::current().currentWorkspace();

Well well = project.getWell("Well1");
Dataset datafull = well.getDataset("DATAFULL");
Dataset lqc = well.getDataset("LQC");

Variable gr = datafull.getVariable("GR");
Variable neut = lqc.getVariable("NEUT");
Variable dens = datafull.getVariable("DENS");

Logview logview = Logview::create(workspace);

NormalTrack normalTrack1 = NormalTrack::create("NormalTrack1",
logview);
LineTrackItem::create(normalTrack1, neut);
NormalTrack normalTrack2 = NormalTrack::create("NormalTrack2",
logview);
LineTrackItem::create(normalTrack2, gr);
NormalTrack normalTrack3 = NormalTrack::create("NormalTrack3",
logview);
LineTrackItem::create(normalTrack3, dens);

ReferenceTrack referenceTrack =
ReferenceTrack::create("ReferenceTrack1", logview, well);

qWarning() << "Reference space of reference track is " <<
referenceTrack.referenceSpace();

lock.release();
```

Show below is the Logview display:

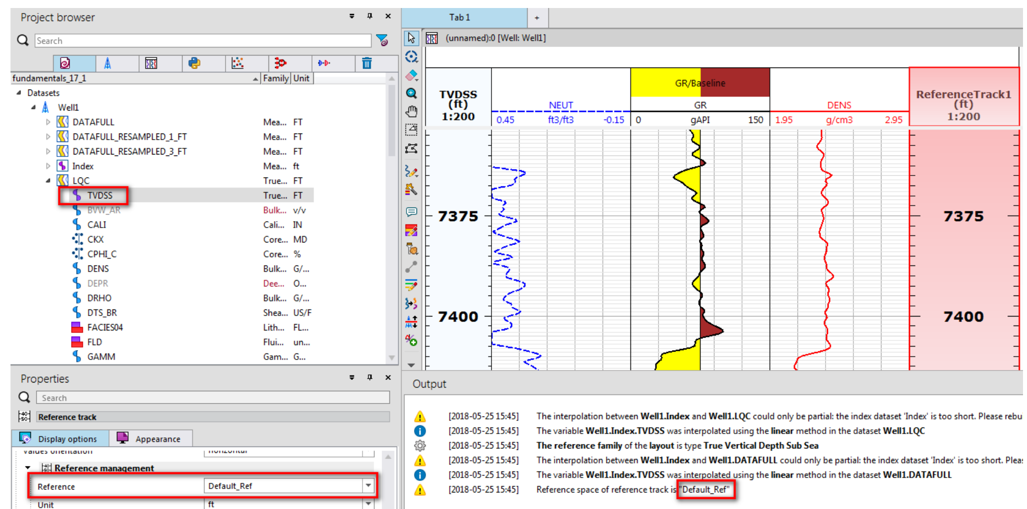


Figure 17 reference track with default reference

The **referenceSpace** of the **ReferenceTrack** can be set thought the **setReferenceSpace** function. The list of possible reference families that can be passed to the function are returned by the **possibleReferenceSpaces** function and a valid reference family for the **ReferenceTrack** can be checked using the **canSetReferenceSpace** function.

The list of **possibleReferenceSpaces** is populated in Techlog from the Index dataset of the well. The Index dataset allows you to switch easily between references in a **Logview** (from MD to TVDSS for example). If this Index dataset isn't computed for the well, the only reference available in the **possibleReferenceSpaces** list is the default reference of the dataset.

In the screenshot below the list of references that can be applied to the ReferenceTrack corresponds to the list of reference variables available in the Index dataset of the well.

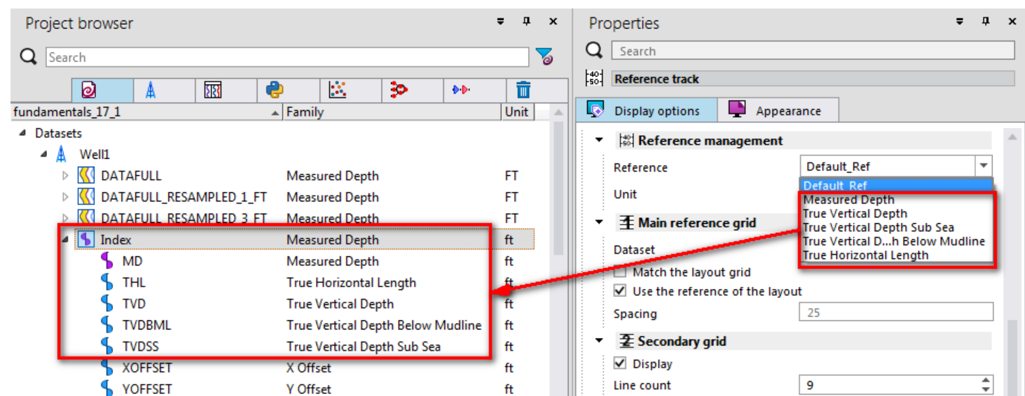


Figure 18 possible reference spaces for the reference track

The example below shows how to set a valid **referenceSpace** to the **ReferenceTrack** and change the **referenceUnit** to a convertible unit.

```
Lock lock2 = LOCK_CREATE_AND_ACQUIRE_ALL(lock2);

ReferenceTrack referenceTrackTVD = ReferenceTrack::create("TVD",
logview, well);

Family family = "True Vertical Depth";
if (referenceTrackTVD.canSetReferenceSpace(family))
{
  referenceTrackTVD.setReferenceSpace(family);
  referenceTrackTVD.setReferenceUnit("m");
}

lock2.release();
```

The **ReferenceTrack** is added at the end of the Logview with the TVD **referenceSpace** and depth values in meters.

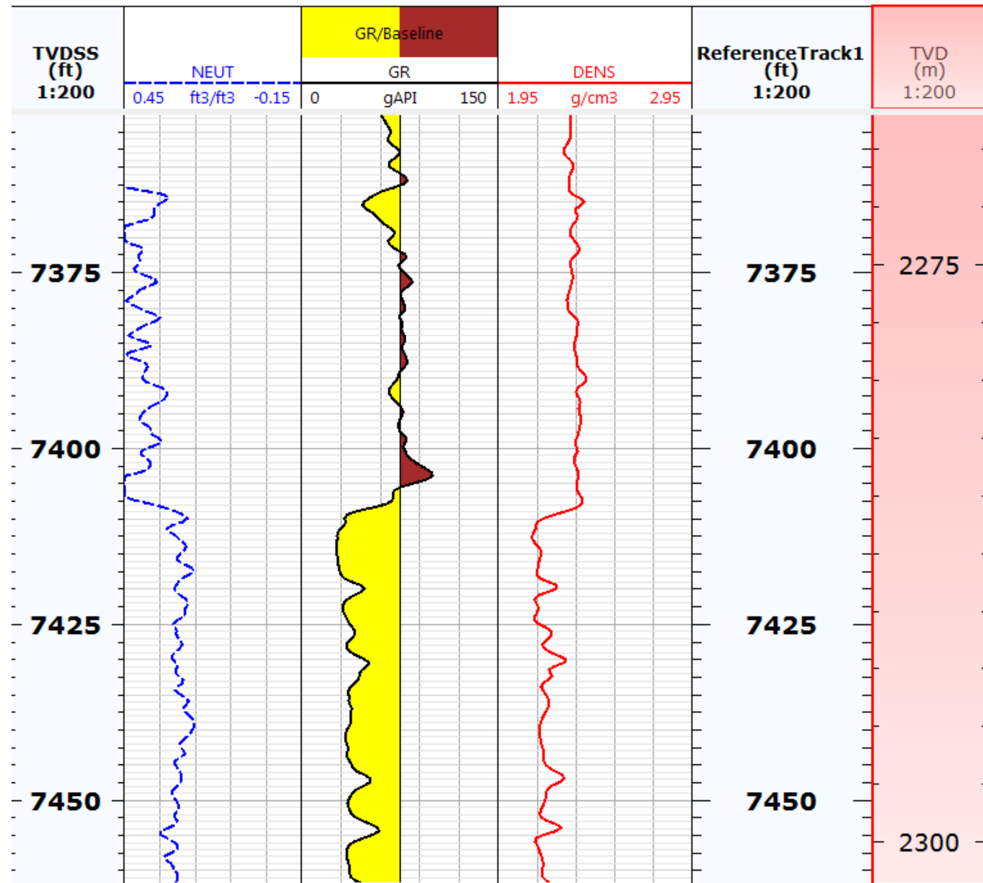


Figure 19 reference track with a reference space

ZonationTrack domain object

The zonation tracks can be enumerated from a **Logview** instance through the **zonationTracks** domain object collection.

```
class Logview : public DomainObject
{
public:
  DomainObjectCollection<ZonationTrack> zonationTracks()
   const;
  …
};
```

A **ZonationTrack** can be created and then added to the **zonationTracks** domain object collection. To create a **ZonationTrack** you can use the **create** static method passing a name that has to be unique in the **zonationTracks** collection and a **Well**. This **Well** parameter means that a **ZonationTrack** can't be added to the **Logview** if there isn't a **NormalTrack** that already exists in the **Logview** and plotting **Variable** data from this parent **Well**.

```
class ZonationTrack : public Track
{
public:
  static ZonationTrack create(const QString &name,
   const Logview &logview, const Well &well);
```

```
};
```

The only public API holds by the **ZonationTrack** class is the **create** static method. This means once the **ZonationTrack** is created the only way to apply zones from a zonation stored in the Techlog database is through the **setSelectedZones** function of the **Logview** class. See "Logview display options" section on page 81 for more information on this function.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();

Workspace workspace = Session::current().currentWorkspace();

Well well = project.getWell("Well1");
Dataset dataset = well.getDataset("DATAFULL");
Variable gr = dataset.getVariable("GR");

Logview logview = Logview::create(workspace);

NormalTrack normalTrack = NormalTrack::create("NormalTrack",
logview);
LineTrackItem::create(normalTrack, gr);

ZonationTrack referenceTrack =
ZonationTrack::create("ZonationTrack", logview, well);

GlobalZonation globalZonation =
project.zonationModel().findGlobalZonation("Stratigraphy");

logview.setSelectedZones(globalZonation,
globalZonation.globalZones(ZoneFamilyType::zoneFamilyName()));

lock.release();
```

Note: The zonation and zones applied to the **Logview** through the **setSelectedZones** is applied to all **zonationTracks** in the **Logview**. For now the ability to change the current zonation set to a **ZonationTrack** isn't exposed with Ocean.

Shown below is the **ZonationTrack** created in the **Logview**:

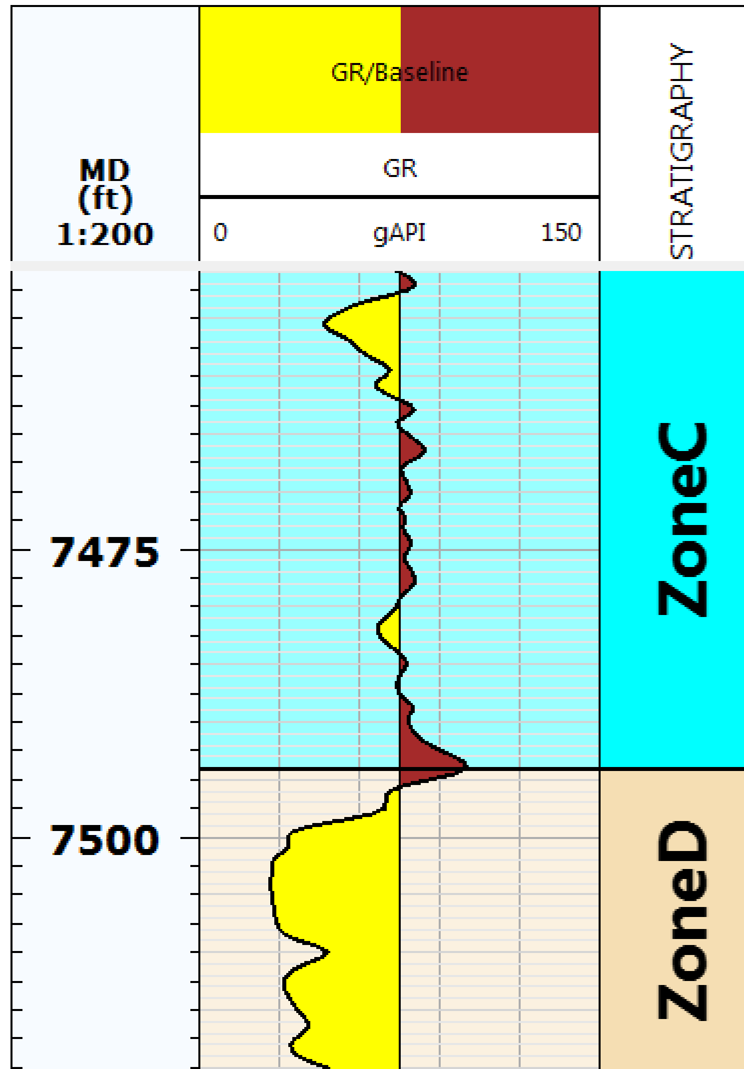


Figure 20 zonation track

TrackItem domain objects

The **NormalTrack** object does not display directly the data but **Variable** data are plotted in the **NormalTrack** through a **TrackItem** object.

```
class TrackItem : public DomainObject
{
public:
  NormalTrack normalTrack() const;
  Variable findVariable() const;

  QString headerName() const;
  void setHeaderName(const QString headerName);


  bool canDisplayHeaderInformation() const;

  void setDisplayHeaderInformation(const bool displayable);

  HeaderNameType headerNameType() const;
```

SLB-Private

```
  void setHeaderNameType(const HeaderNameType headerNameType);
  bool isUnitDisplayed() const;
  void setUnitDisplayed(const bool displayed);
  bool limitsDisplayed() const;
  void setLimitsDisplayed(const bool displayed);

  Unit displayUnit() const;
  …

 enum EventType
 {
   TrackItemMoved
 };

};
```

Only one variable can be displayed per **TrackItem** object but several **TrackItem** objects can be created into a **NormalTrack**. This variable can be retrieved from the **findVariable** method.

From the **TrackItem** domain object you can:

- add a title to the header of the **TrackItem** through **setHeaderName** function
- display the title or the variable name or both in the header passing an **HeaderNameType** enum value to **setHeaderNameType** function
- **canDisplayHeaderInformation** set to false hides all **TrackItem**'s information in the header of the track.
- show / hide **TrackItem** unit and limit values in the header
- gets the unit used to display data in the **TrackItem**

From the parent **NormalTrack**, the **trackItems** method allows you to retrieve the domain object collection of **TrackItem** that belongs to the **NormalTrack**.

A **TrackItem** object can listen to the **TrackItemMoved** event that allows to be notified when this **TrackItem** is moved to another track.

The **TrackItemMoved** signal includes an argument that gives the source and destination **Track** where has been moved the **TrackItem**. This argument is modeled in Ocean for Techlog through the **TrackItemMovedArgs** class.

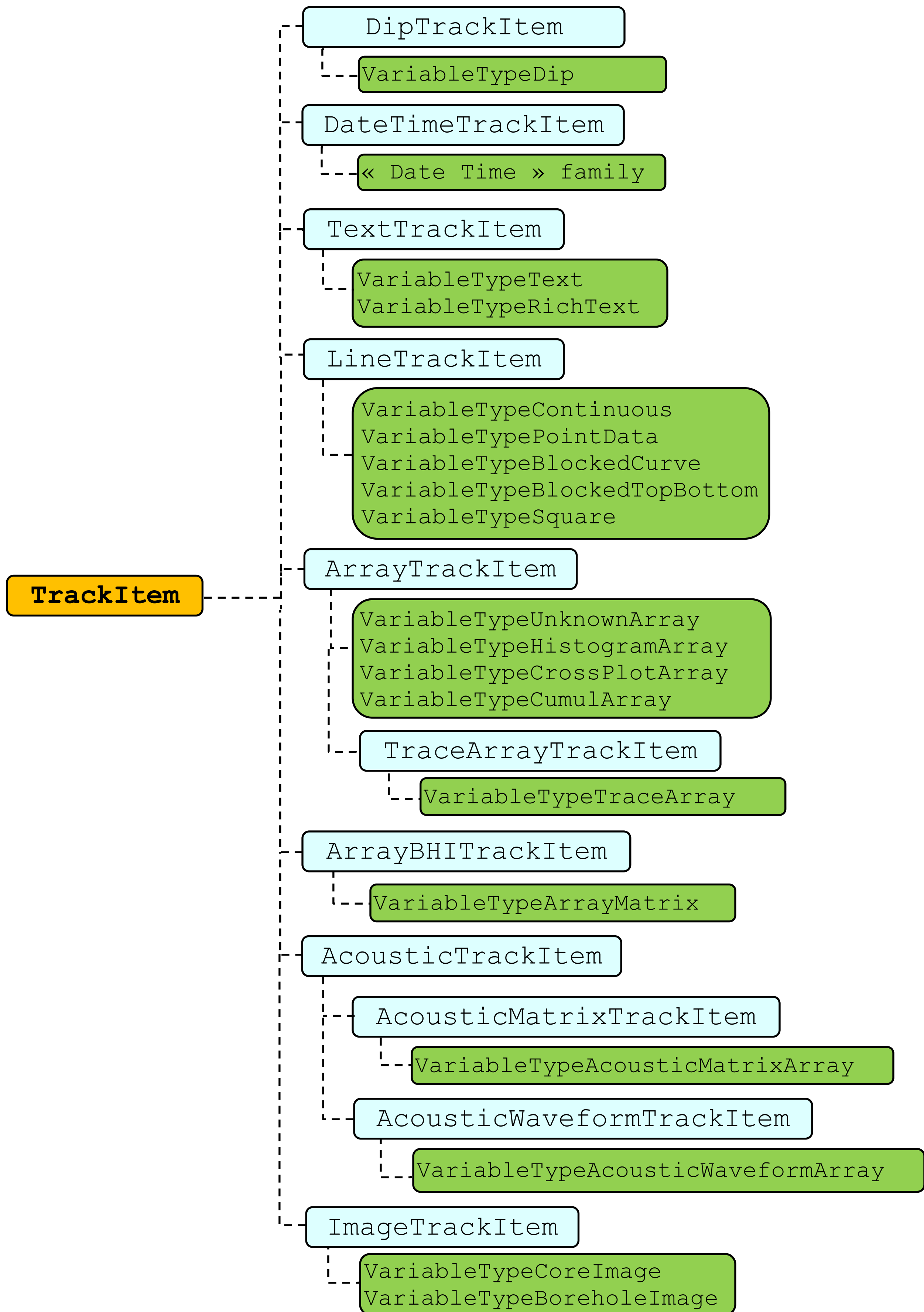
```
class TrackItemMovedArgs : public SignalArgsT<TrackItem>
{
public:
 …
 Track sourceTrack() const;
 Track destinationTrack() const;
};
```

Note: The **Track** returned by the **sourceTrack** and the **destinationTrack** can only be cast to a **NormalTrack** domain object.

TrackItem is a base class of track item derived classes used depending on **Variable** type that you intend to plot.

Note: Excepted **`BHATrackItem`** and **`WellSchematicsTrackItem`** **`create`** static methods are waiting for a **`Dataset`** object with a specific **`type`**.

The following class diagram shows all track item classes inheriting from the base class **`TrackItem`** and which class has to be used according to the variable type or dataset type that you want to display.



SLB-Private

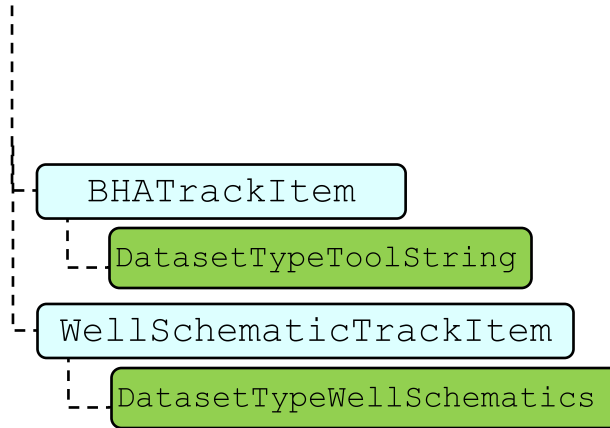


Figure 21 Track items class diagram

The derived classes hold the **`create`** static method used to plot variable in a track of the Logview. This create pattern is expecting as arguments an instance of the parent **`NormalTrack`** and an instance of the **`Variable`** that you want to display in this parent **`NormalTrack`**.

Note: Throw an exception if you try to display a variable type with the wrong **`TrackItem`** derived class. In order to avoid this plug-in exception Ocean provides in each **`TrackItem`** derived class **`tryCreate`** and **`canCreate`** static functions that check if it is the correct **`TrackItem`** derived class to display this type of variable.

The following example shows how to visualize into a **`LineTrackItem`** a single curve "Gamma Ray" with variable type set to continuous.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Well well = DomainObject::get(wellDroid).cast<Well>();

Variable gr = DomainObject::get(variableDroid).cast<Variable>();

Workspace workspace = Session::current().currentWorkspace();

// create logview
Logview logview = Logview::create(workspace);

// create track
NormalTrack track = NormalTrack::create( QLatin1String( "track 1"
), logview );

// create a track item depending on gamma ray variable type
if (LineTrackItem::tryCreate(track, gr).isNull())
{
  LineTrackItem logGR = LineTrackItem::create( track, gr);
}

lock.release();
```

`TrackItem` class holds some methods to add, remove and list **`Parameter`** domain objects along the track item.

SLB-Private

```
class TrackItem : public DomainObject
{
public:
  void addParameter(const Parameter& parameter);
  void removeParameter(const Parameter& parameter);
  const DomainObjectCollection<Parameter> parameters() const;
  …
};
```

See the "Parameter Domain Object" section in *Ocean for Techlog Developer Guide – Data&Workflow* for more information on **`Parameter`**.

See the "Workstep results display" section in in *Ocean for Techlog Developer Guide – Data&Workflow* for more information on how to display parameter bound to workstep argument for a given zone into default logview created by the AWI.

LineTrackItem Domain Object

`LineTrackItem` domain object inherits from **`TrackItem`** base class and is used to display in the Logview 1D variable (not greater that one column) with the following type values:

- continuous - **`VariableTypeContinuous`**
- point data - **`VariableTypePointData`**
- blocked curve – **`VariableTypeBlockedCurve`**
- blocked and centered - **`VariableTypeBlockedTopBottom`**
- square log - **`VariableTypeSquare`**

```
class LineTrackItem : public TrackItem
{
public:
  static LineTrackItem  create (NormalTrack track,
  const Variable &variable);

  QColor color() const ;
  void setColor(const QColor &color);


  DomainObjectCollection<Areafill> areafills() const;

  QList<Baseline> baseLines() const;
  void setBaselines(const QList<Baseline> &lines);

  void setConstantUncertainty(const float min, const float
  max, const QColor &color)
  void setVariableUncertainty(const Variable &min,
  const Variable &max, const QColor &color);

  AxisLimitType horizontalLimitType() const;
  void setHorizontalLimitType(const AxisLimitType type);
  float horizontalUserLowerLimit() const;
  float horizontalUserUpperLimit() const;
  void setHorizontalUserLimits(const float lower,
```

```
  const float upper, const Unit &unit);
  float horizontalVariableLowerLimit() const;
  float horizontalVariableUpperLimit() const;
  float horizontalFamilyLowerLimit() const;
  float horizontalFamilyUpperLimit() const;

  bool isPointsVisible() const;
  void setPointsVisible(const bool visible);
  PointType pointsType() const;
  void setPointsType(const PointType type);
  float pointsSize() const;
  void setPointsSize(const float size);


  bool isDraggable() const;
  void setDraggable(bool isDraggable);

  double paletteMinimumValue() const;
  double paletteMaximumValue() const;
  void setPaletteMinimumValue(const double min);
  void setPaletteMaximumValue(const double max);
  …

  enum EventType
  {
    LineTrackItemHorizontalAxisChanged
  };
};
```

The properties of **`LineTrackItem`** class allow you to:

- get and set the **`color`** of the curve line
- enumerate area fills that exist between this **`LineTrackItem`** and the upper and lower limits of the track (**`SingleAreafill`**) and between this **`LineTrackItem`** and a **`Baseline`** or another **`LineTrackItem`** (**`DoubleAreafill`**)
- add a **`Baseline`** to the **`LineTrackItem`** and get the list of **`baselines`** added to this LineTrackItem
- set constant or variable uncertainty values on left and righ sides of the curve displayed in the **`LineTrackItem`**
- get and set the horizontal limit type through **`AxisLimitType`** enum class
  - By default axis limit is set to variable

```
enum AxisLimitType
{
  AxisLimitTypeVariable,
  AxisLimitTypeUser,
  AxisLimitTypeFamily
};
```

- set lower and upper horizontal axis limit values with the considered **`Unit`** turning on **`horizontalLimitType`** to **`AxisLimitTypeUser`** (see "Plot axes limits and display parameters" in in *Ocean for Techlog Developer Guide – Basics*)

- The **Unit** tells the **LineTrackItem** which unit has to be considered regarding limits values passed to the function. Those values are converted from the **Unit** to current horizontal limit unit and converted values are set to user horizontal limits.

- get the family and variable limits when the **AxisLimitType** is set respectively to **AxisLimitTypeFamily** and **AxisLimitTypeVariable**
- ability through **setPointsVsible** method to turn on points display option and set a type to these points through the enum class **PointType**

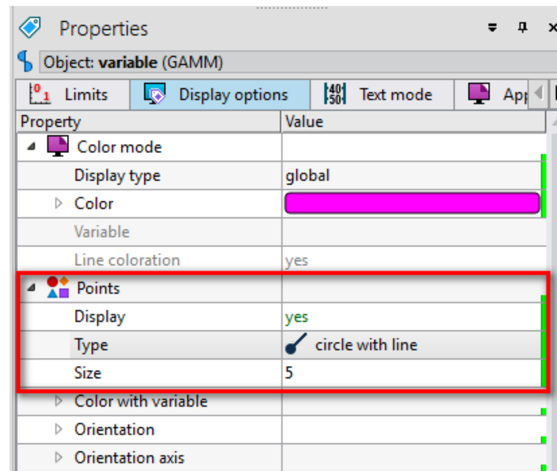


Figure 22 Display points option turned on

- you can drag or not this **LineTrackItem** from the source track to a destination track
- get and set limit values of a **palette** applied to the **LineTrackItem**

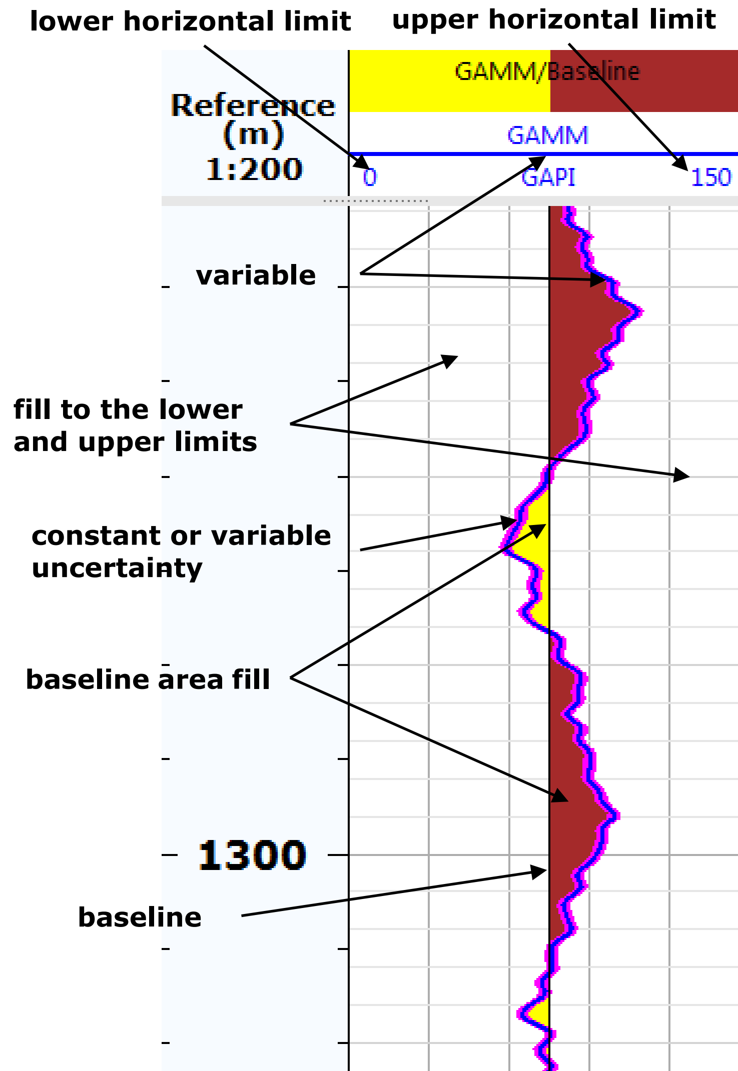


Figure 23 LineTrackItem properties

A **LineTrackItem** domain object can subscribe on the **LineTrackItemHorizontalAxisChanged** signal that is emitted when the horizontal axis limits are changed.

The **LineTrackItemHorizontalAxisChanged** signal includes an argument that gives the **LineTrackItem** sender object and then from where you can get the new horizontal axis limit values. This argument is modeled in Ocean for Techlog through the **LineTrackItemHorizontalAxisChangedArgs** class.

```
class LineTrackItemHorizontalAxisChangedArgs : public Sig-
nalArgsT<LineTrackItem>
{
};
```

ArrayTrackItem Domain Object

ArrayTrackItem domain object inherits from **TrackItem** base class and is used to display in the Logview Array variable (greater that one column) with the following type values:

- unknown array - **VariableTypeUnknownArray**
- histogram array - **VariableTypeHistogramArray**
- cross-plot array – **VariableTypeCrossPlotArray**
- cumulated array – **VariableTypeCumulArray**

```
class ArrayTrackItem : public TrackItem
{
public:
  static ArrayTrackItem  create(NormalTrack track,
  const Variable &variable);


  const ArrayTrackItemDisplayDrift displayDirection() const;
  void setDisplayDirection(const ArrayTrackItemDisplayDrift
  displayDirection);

  const ArrayTrackItemAxisTypeIndex
  horizontalAxisIndexAxisType() const;
  void setHorizontalAxisIndexAxisType(const
  ArrayTrackItemAxisTypeIndex horizontalAxisIndexAxisType);
  const float horizontalAxisIndexAxisLowerLimit() const;
  void setHorizontalAxisIndexAxisLowerLimit(const float
  horizontalAxisIndexAxisType);
  const float horizontalAxisIndexAxisUpperLimit() const;
  void setHorizontalAxisIndexAxisUpperLimit(const float
  horizontalAxisIndexAxisUpperLimit);

  const AxisLimitType verticalAxisType() const;
  void setVerticalAxisType(const AxisLimitType
  verticalAxisType);
  const float verticalAxisLowerLimit() const;

  const float verticalAxisUpperLimit() const;

  void setVerticalAxisLimits(const float verticalAxisLowerLimit,
  const float verticalAxisUpperLimit, const Unit &limitUnit);

  const bool smoothMatrixDisplay() const;
  void setSmoothMatrixDisplay(const bool smoothMatrixDisplay);
  const bool matrixDisplayFillMissingValues() const;
  void setMatrixDisplayFillMissingValues(const bool
  matrixDisplayFillMissingValues);

  void setPalette(const QString &name, StorageLevel level);

  double paletteMinimumValue() const;
  double paletteMaximumValue() const;
  void setPaletteMinimumValue(const double min);
  void setPaletteMaximumValue(const double max);

  void reversePalette();
  void restoreDefaultPalette();
  void adaptPaletteToLocalLimits();
```

```
  void adaptPaletteToGlobalLimits();
  …
};
```

The properties of **`ArrayTrackItem`** class allow you to:

- get and set the display direction through enum class values **`ArrayTrackItemDisplayDrift`**
  - the display of the variable in the track changes following the display direction value selected in the list

```
enum ArrayTrackItemDisplayDrift
{
  ArrayTrackItemDisplayDriftRowDisplay,
  ArrayTrackItemDisplayDriftCumulated,
  ArrayTrackItemDisplayDriftCumulatedSorted,
  ArrayTrackItemDisplayDriftMatrixDisplay,
  ArrayTrackItemDisplayDriftColumnDisplay
};
```

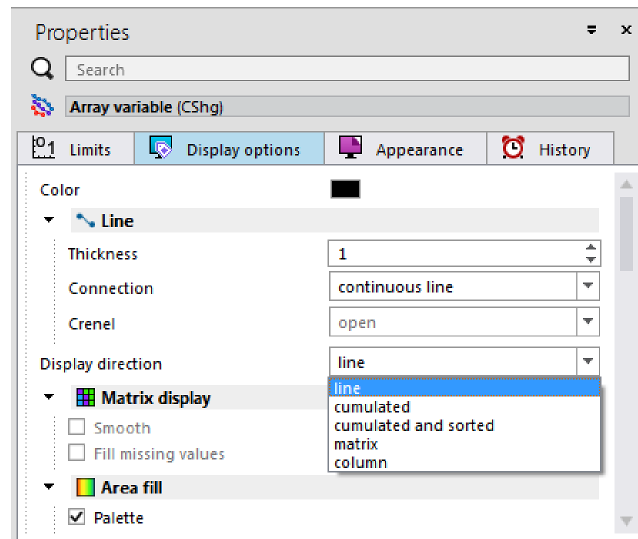


Figure 24 ArrayTrackItem display directions

- set lower and upper horizontal axis limits with **`horizontalAxisIndexAxisType`** turned to **`ArrayTrackItemAxisTypeIndexUser`**
  - this option allows you to set the lower and upper index column of the array variable that you want to see in the track
- set lower and upper vertical axis limits with **`verticalAxisType`** turned to **`AxisLimitTypeUser`**
  - this option allows you to display a range values of the variable in the track
  - this array track property is only enabled for **`ArrayTrackItemDisplayDriftRowDisplay`**,

ArrayTrackItemDisplayDriftMatrixDisplay, and **ArrayTrackItemDisplayDriftColumnDisplay**

- The **Unit** tells the **ArrayTrackItem** which unit has to be considered regarding limits values passed to the function. Those values are converted from the **Unit** to the current **Logview** reference unit and converted values are set to limits. See "Plot axes limits and display parameters" in in *Ocean for Techlog Developer Guide – Basics* .

Note: The function throws an exception if you set limits with wrong display direction value. If the display direction is set to one of the cumulated types (**ArrayTrackItemDisplayDriftCumulated** and **ArrayTrackItemDisplayDriftCummulatedSorted**) the vertical limits are set by default to the min and max variable data values.

- get and set the vertical axis scale type through **AxisScaleType** enum class values, can be linear, logarithmic or tangential
- turn on the smooth matrix display and filling missing values options
  - those properties have an effect only when diplay direction is set to **ArrayTrackItemDisplayDriftMatrixDisplay**
- set a color palette value to the array track at a storage level (Techlog, Company, Project, User, Plug-in folder)
- get and set limit values of a **palette** applied to the **ArrayTrackItem**
- reverse the color gradient of the palette
  - the min color is swapt with the max color
  - this does not change the palette min and max
- restore the "default" palette assigned to the variable displayed through the **ArrayTrackItem**
  - This API can be used for example right after the **Logview::setVerticalTopBottomLimits** call. It results to a palette with a color gradiant that best fit the area of interest.
- adapt the palette min and max boundaries to values of the variable corresponding to the area visible in the **Logview**
- adapt the palette min and max boundaries to match all values of the variable (variable min and max)

The following example shows how to plot in ArrayTrackItem a variable with type "unkown array". This variable has 100 columns, after to change the display direction to matrix display the lower and upper horizontal limits are set to display only the 10 first column values. The lower and upper vertical limits are set to display a limted range of values.

Smooth matrix display property is turned on, and a palette color stored at the Techlog level is set to **ArrayTrackItem**.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Workspace workspace = Session::current().currentWorkspace();

Logview logview = Logview::create(workspace);
```

```
NormalTrack track = NormalTrack::create("track1", logview);

Variable arrayVar =
DomainObject::get(droidArrayVariable).cast<Variable>();
ArrayTrackItem arrayTrackItem = ArrayTrackItem::create(track,
arrayVar);

arrayTrackItem.setDisplayDirection(
ArrayTrackItemDisplayDriftMatrixDisplay);
arrayTrackItem.setHorizontalAxisIndexAxisType(
ArrayTrackItemAxisTypeIndexUser);
arrayTrackItem.setHorizontalAxisIndexAxisLowerLimit(0);
arrayTrackItem.setHorizontalAxisIndexAxisUpperLimit(10);

arrayTrackItem.setVerticalAxisType(AxisLimitTypeUser);
arrayTrackItem.setVerticalAxisLimits(65, 70,
arrayVar.displayUnit());

arrayTrackItem.setSmoothMatrixDisplay(true);
arrayTrackItem.setPalette("WBSSingleDepthPalette",
StorageLevelTechlog);

lock.release();
```

Below is the display of the **`ArrayTrackItem`**:

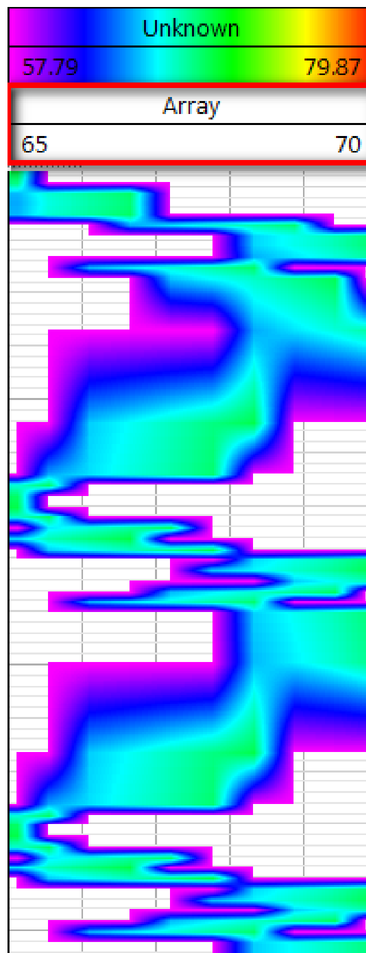


Figure 25 Display a range of values in a ArrayTrackItem

TraceArrayTrackItem Domain Object

`TraceArrayTrackItem` domain object inherits from **`ArrayTrackItem`** base class and is used to display in the Logview an array variable (greater that one column) with "trace array" type (**`VariableTypeTraceArray)`**.

```
class TraceArrayTrackItem : public ArrayTrackItem
{
public:
  static TraceArrayTrackItem create(NormalTrack track,
  const Variable &variable);

  QColor firstTraceColor() const;
  void setFirstTraceColor(const QColor &color);
  QColor lastTraceColor() const;
  void setLastTraceColor(const QColor &color);
  QColor tracesColor() const;
  void setTracesColor(const QColor &color);
  QList<bool> traceSelection() const;
  void setTraceSelection(const QList< bool > &selection);
  bool isTraceSelected(const int index) const;
```

```
  void selectTrace(const int index);
  void unselectTrace(const int index);
  void selectAllTraces();
  void unselectAllTraces();
};
```

Properties of **`TraceArrayTrackItem`** class allow you to:

- get and set the first, last and others traces colors through respectively **`firstTraceColor`**, **`lastTraceColor`** and **`tracesColor`** properties.

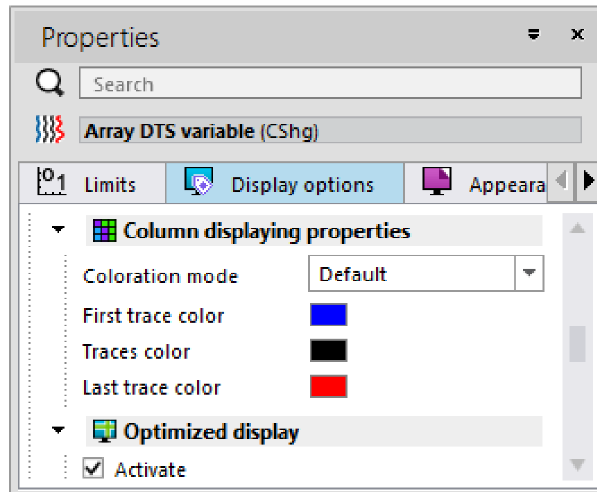


Figure 26 Trace array track item column display options in property editor

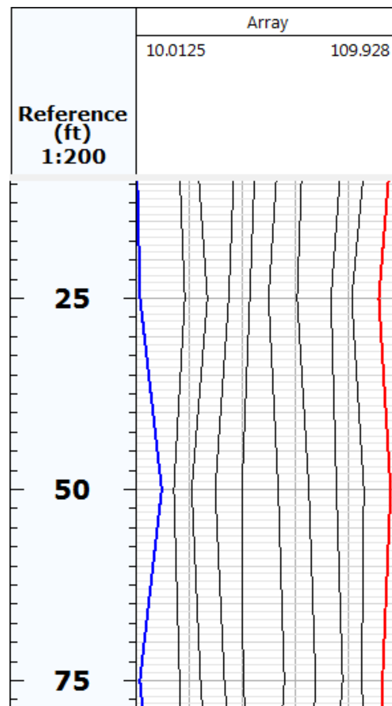


Figure 27 Trace array track item column display properties

- A "trace selection" button shows up in the Edit menu or contextual toolbar when the trace array track item is selected in the **`Logview`**. This button popups a dialog from which you can hide or display the traces of the "trace array" variable plotted through the **`TraceArrayTrackItem`**. API of the TraceArrayTrackiTem allows you to unselect/select some traces passing the column index or unselect/select all the traces.

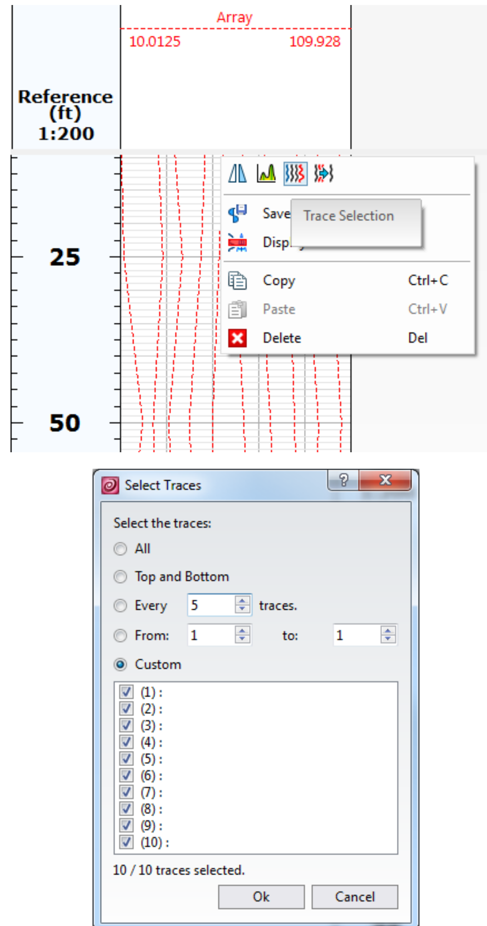


Figure 28 Trace selection

Below is an example setting the first and last trace colors respectively to blue and red and selecting traces every five traces.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Workspace workspace = Session::current().currentWorkspace();

Logview logview = Logview::create(workspace);

NormalTrack track = NormalTrack::create("track1", logview);

Variable arrayVar =
DomainObject::get(droidArrayVariable).cast<Variable>();
TraceArrayTrackItem traceArrayTrackItem =
TraceArrayTrackItem::create(track, arrayVar);
```

```
traceArrayTrackItem.setFirstTraceColor(Qt::blue);
traceArrayTrackItem.setTracesColor(Qt::black);
traceArrayTrackItem.setLastTraceColor(Qt::red);

quint64 nbTraces = arrayVar.columnCount();
for (int i = 0; i < nbTraces; i++)
{
  if (i % 5)
    traceArrayTrackItem.unselectTrace(i);
  else
    traceArrayTrackItem.selectTrace(i);
}

lock.release();
```

Below is the display of the **`TraceArrayTrackItem`**:

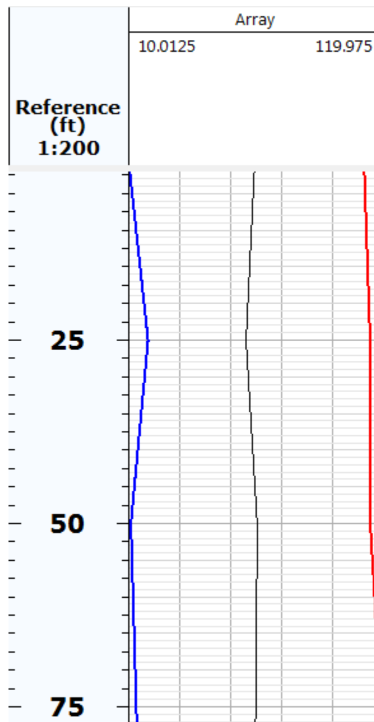


Figure 29 TraceArrayTrackItem traces selected every 5 traces

ArrayBHITrackItem Domain Object

`ArrayBHITrackItem` domain object inherits from **`TrackItem`** base class and is used to display in the Logview Array variable (greater that one column) with "matrix array" type (**`VariableTypeArrayMatrix`**).

```
class ArrayBHITrackItem : public TrackItem
{
public:
  static ArrayBHITrackItem create(NormalTrack track,
  const Variable &variable);

  void setPalette(const QString &name, const StorageLevel
  level);

  bool areaFillValuesVisible() const;
  void setAreaFillValuesVisible(const bool
  areaFillValuesVisible);
  bool areaFillTextVisible() const;
  void setAreaFillTextVisible(const bool areaFillTextVisible);

  AxisScaleType limitScaleType() const;
  void setLimitScaleType(const AxisScaleType type);
  float upperLimit() const;
  float lowerLimit() const;
  void setLimit(const float lower, const float upper);

  bool isMissingValuesFilled() const;
  void setMissingValuesFilled(const bool fill);

  double paletteMinimumValue() const;
  double paletteMaximumValue() const;
  void setPaletteMinimumValue(const double min);
  void setPaletteMaximumValue(const double max);

  void reversePalette();
  void restoreDefaultPalette();
  void adaptPaletteToLocalLimits();
  void adaptPaletteToGlobalLimits();
};
```

The properties of **`ArrayBHITrackItem`** class allow you to:

- set a color palette value to the array track at a storage level (Techlog, Company, Project, User, Plug-in folder)
- show / hide area fill text and values in the track header

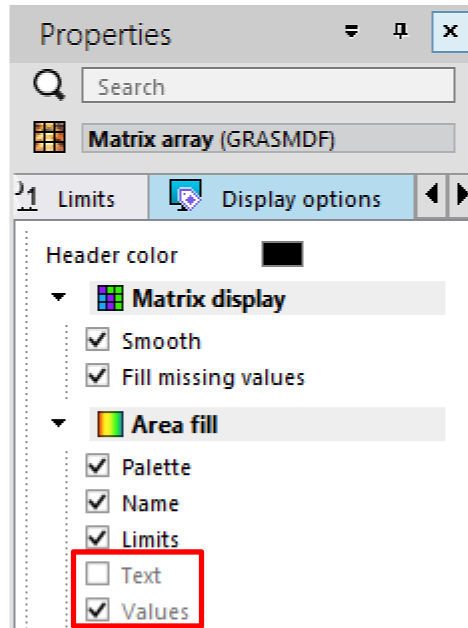


Figure 30 Areafill BHI track item display properties

- get and set the axis scale type through **`AxisScaleType`** enum class values, can be linear, logarithmic or tangential
- get and set upper and lower limit of the values that you want to display in the track item
- turn on filling missing values option
- get and set limit values of a **`palette`** applied to the **`ArrayBHITrackItem`**
- reverse the color gradient of the palette
  - the min color is swapt with the max color
  - this does not change the palette min and max
- restore the "default" palette assigned to the variable displayed through the **`ArrayBHITrackItem`**
  - This API can be used for example right after the **`Logview::setVerticalTopBottomLimits`** call. It results to a palette with a color gradiant that best fit the area of interest.
- adapt the palette min and max boundaries to values of the variable corresponding to the area visible in the **`Logview`**
- adapt the palette min and max boundaries to match all values of the variable (variable min and max)

DateTimeTrackItem Domain Object

`DateTimeTrackItem` domain object inherits from **`TrackItem`** base class and is used to display a "Date Time" family variable in the Logview as a suite of dates and hours.

This can be used to add a track in the Logview displaying a date-time reference.

Note: If one of the following prerequisites is not respected the create method throws an exception:

- the **`Variable`** column count must be equal to 1
- the **`Variable`** family must be equal to "Date Time"
- the **`Variable`** unit must be equal to "ms"
- the **`Variable`** type has be set to **`VariableTypeDateTime`**

```
class DateTimeTrackItem : public TrackItem
{
public:
  static DateTimeTrackItem create(NormalTrack track,
  const Variable &variable);

  QColor color() const;
  void setColor(const QColor &color);
  QString timeFormat() const;
  void setTimeFormat(const QString &format);
  QString dateFormat() const;
  void setDateFormat(const QString &format);
  bool isDateVisible() const;
  void setDateVisible(const bool visible);
  int dateStep() const;
  void setDateStep(const int step);
};
```

Properties of **`DateTimeTrackItem`** class allow you to:

- get and set the color of the track item
- get and set date and time format. Formats available for date time reference are:
  - "d-MMM-yyyy h:m:s.z"
  - "yyyy-MM-dd hh:mm:ss.zzz"
  - "yyyy-MM-dd hh:mm:ss"
  - "yyyy/MM/dd-hh:mm:ss"
  - "yyyy/MM/ddThh:mm:ss"
  - "dd/MM/yyyy-hh:mm:ss"
  - "dd/MM/yyyy hh:mm:ss"
  - "dd/M/yyyy hh:mm:ss"
  - "d/M/yyyy h:m:s"
  - "d/MM/yyyy hh:mm:ss"
  - "d/M/yyyy hh:mm:ss"
  - "dd/M/yyyy hh:mm:ss"
  - "dd-MMM-yy hh:mm:ss"
  - "M/d/yyyy h:m:s AP"
  - "M/d/yyyy h:m:s"
  - "M/d/yyyyThh:mm:ss.zzz"

– "M/d/yyyy hh:mm:ss.zzz"
– "hh:mm:ss dd-MMM-yyyy"
– "hh:mm:ss.dd-MMM-yyyy"
– "hh:mm:ss dd-MMM-yy"
– "h:mm:ss dd-MMM-yy"

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();
QString wellName = project.getUniqueWellName(wellNameBase);
Well well = Well::create(wellName, project);
Dataset dataset = Dataset::create("myDataset", "MD",
VariableDataFormatFloat, 1000, well);
Variable reference = dataset.findReferenceVariable();
reference.setFamily("Measured Depth");
reference.setUnit("ft");
Variable variable = Variable::create("DateTime", dataset,
VariableDataFormatFloat, VariableTypeDateTime, 1);
variable.setFamily("Date Time");
variable.setUnit("ms");

QDateTime dateTime = QDateTime::currentDateTime();

for (int i = 0 ; i < 1000 ; i++)
{
  reference.setFloatValue(i, 0, 10.0f*i);

  qint64 ms = dateTime.toMSecsSinceEpoch();
  variable.setFloatValue(i, (float)ms);
  dateTime = dateTime.addDays(1);
}

Workspace workspace = Session::current().currentWorkspace();

Logview logview = Logview::create(workspace);

NormalTrack track = NormalTrack::create("track1", logview);

DateTimeTrackItem datetimeTrackItem =
DateTimeTrackItem::create(track, variable);

datetimeTrackItem.setColor(Qt::blue);
datetimeTrackItem.setDateFormat("yyyy/MM/dd");
```

```
datetimeTrackItem.setTimeFormat("hh:mm:ss");
datetimeTrackItem.setDateStep(2);

lock.release();
```

Display of the **`DateTimeArrayTrackItem`**:

Figure 31 DateTimeTrackItem with date displayed every two steps

TextTrackItem Domain Object

`TextTrackItem` domain object inherits from **`TrackItem`** base class and is used to display text variables in the Logview (not greater that one column) with the following type values:

- text - **`VariableTypeText`**
- rich text - **`VariableTypeRichText`**

```
class TextTrackItem : public TrackItem
{
public:
  static TextTrackItem create(NormalTrack track,
  const Variable &variable);

  void setDisplayType(const TextTrackItemDisplayType
  displayType);
  TextTrackItemDisplayType displayType() const;

  void setTextSeparatorsVisible(const bool visible);
  bool textSeparatorsVisible() const;
  void setTextBackgroundVisible(const bool visible);
  bool isTextBackgroundVisible() const;
  void setTextBackgroundColor(const QColor &backgroundColor);
```

```
  QColor textBackgroundColor() const;

  void setCrenelConnectionType(const LineType type);
  LineType crenelConnectionType() const;
  void setCrenelConnectionLineStyle(const LineStyle style);
  LineStyle crenelConnectionLineStyle() const;
  void setCrenelConnectionLineThickness(const int thickness);
  int crenelConnectionLineThickness() const
};
```

The properties of **`TextTrackItem`** class allow you to:

- change the display type from text to blocked curve
  - if **`displayType`** is set to **`TextTrackItemDisplayTypeBlockedCurve`** enum value you can change crenel connection type, style and line thickness through the corresponding API

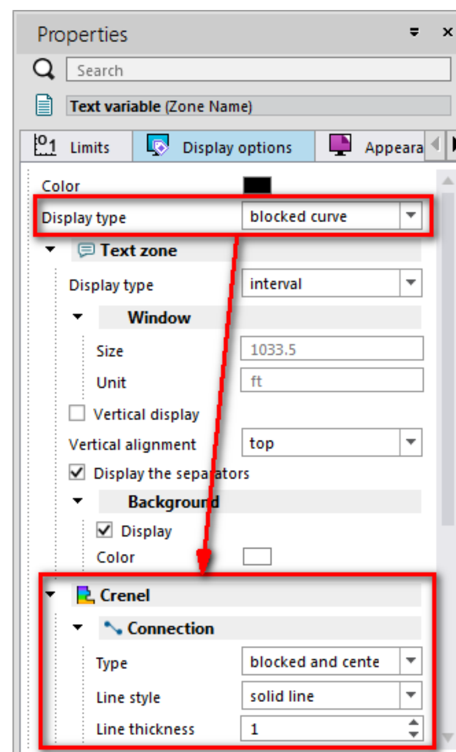


Figure 32 TextTrackItem blocked curve display options

  - if **`displayType`** is set to **`TextTrackItemDisplayTypeText`** enum value you can change the background color or hide/show the background color and the sperators between text zones

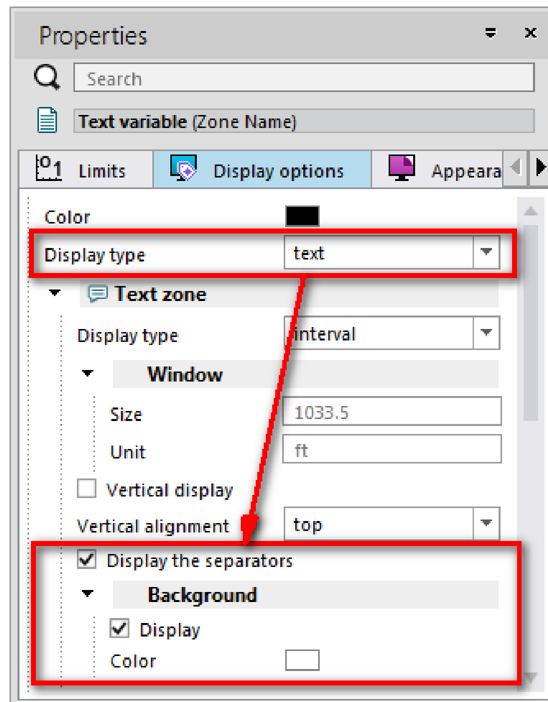


Figure 33 TextTrackItem text display options

The following example shows how to add to a **`NormalTrack`** of the **`Logview`** two **`TextTrackItem`**: one with display type set to text and the other one set to blocked curve.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Logview logview = Logview::create(workspace);

Variable textVariable =
DomainObject::get(variableDroid).cast<Variable>();

NormalTrack track1 = NormalTrack::create("track1", logview);

TextTrackItem textTrackItem1 = TextTrackItem::create(track1,
textVariable);
textTrackItem1.setTextBackgroundColor(Qt::lightGray);
textTrackItem1.setTextSeparatorsVisible(false);

NormalTrack track2 = NormalTrack::create("track2", logview);

TextTrackItem textTrackItem2 = TextTrackItem::create(track2,
textVariable);
textTrackItem2.setDisplayType(TextTrackItemDisplayTypeBlockedC
urve);
```

```
textTrackItem2.setCrenelConnectionType(LineTypeBlockedAndUpHol
e);
textTrackItem2.setCrenelConnectionLineThickness(3);

lock.release();
```

The following screenshot is the result of this code and shows the difference between text and blocked curve settings.

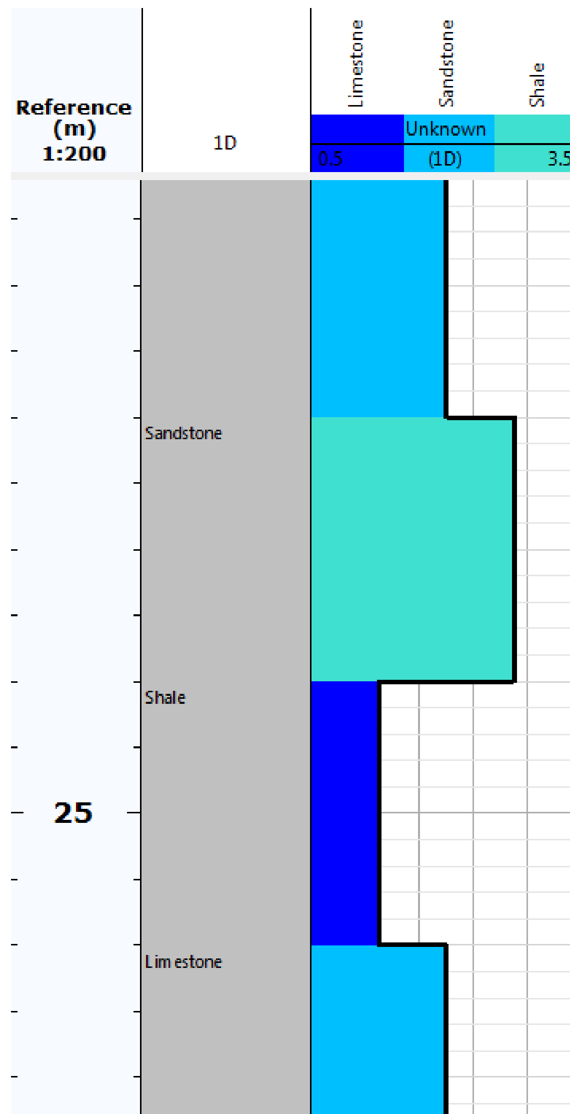


Figure 34 TextTrackItem display types

AcousticTrackItem Domain Object

`AcousticTrackItem` domain object inherits from **`TrackItem`** base class and is itself the base class that holds common methods for **`AcousticWaveformTrackItem`** and **`AcousticMatrixTrackItem`** domain objects.

```
class AcosuticTrackItem : public TrackItem
{
public:

  void setPalette(const QString &name, StorageLevel level);

};
```

The properties of **AcousticTrackItem** class allow you to:

- set a color palette value to the array track at a storage level (Techlog, Company, Project, User, Plug-in folder)

AcousticWaveformTrackItem Domain Object

AcousticWaveformTrackItem domain object inherits from **AcousticTrackItem** base class and is used to display in the Logview an array variable (greater than one column) with "multi-dimensionnal acoustic array" type (**VariableTypeAcousticWaveformArray).**

Note: Waveform number of samples (TL_WFLEN) **DataProperty** is a mandatory property to visualize multi-dimensionnal acoustic data in an **AcousticWaveformTrackItem.** The **create** method throws an exception if the TL_WFLEN property does not exist in the variable or is greater than variable column count. Other optional acoustic data properties are:

- TL_NUMRCV – number of axial receivers (mandatory to see all receivers in the list)
- TL_AZRCV – number of azimuthal receivers (mandatory to see all azimuths in the list)
- TL_WFSTART – waveform start time
- WF_SAMRATE – sonic waveform sampling rate

If you want to know more about acoustic data properties, see the "Acoustics Data Preparation" section in *Techlog online help*.

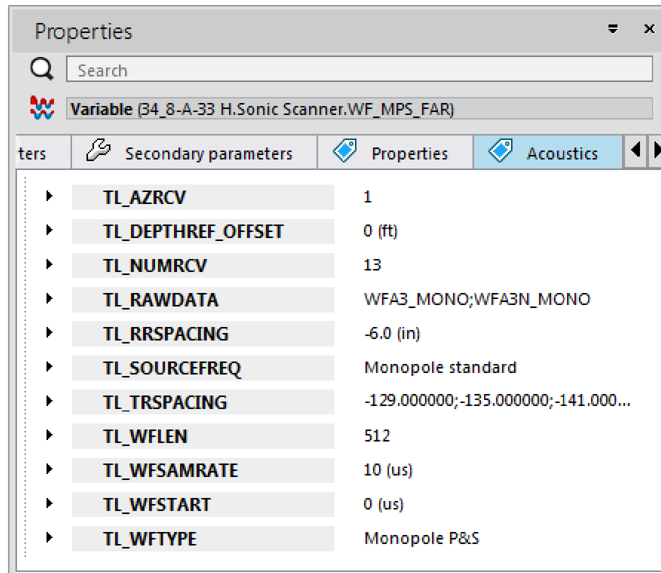


Figure 35 Acoustic properties associated to a "multi-dimensionnal acoustic array" variable

```
class AcousticWaveFormTrackItem : public AcousticTrackItem
{
public:

  static AcousticWaveFormTrackItem create(NormalTrack track,
  const Variable &variable);

  WaveDisplayMode displayMode() const;
  void setDisplayMode(const WaveDisplayMode mode);

  int currentAzimuth() const;
  void setCurrentAzimuth(const int current);
  int azimuthCount() const;
  int currentReceiver() const;
  void setCurrentReceiver(const int current);
  int receiverCount() const;
  bool isReceiverDisplayed(const int index) const;
  void setReceiverDisplay(const int index, const bool
  display);
  int receiverLineThickness(const int index) const;
  void setReceiverLineThickness(const int index, const int
  thickness);
  QColor receiverLineColor(const int index) const;
  void setReceiverLineColor(const int index, const QColor
  &color);
  QColor receiverUpperAreaFillColor(const int index) const;
  void setReceiverUpperAreaFillColor(const int index, const
  QColor &color);
  QColor receiverLowerAreaFillColor(const int index) const;
  void setReceiverLowerAreaFillColor(const int index, const
  QColor &color);
```

```
  AxisLimitType verticalLimitType() const;
  void setVerticalLimitType(const AxisLimitType type);
  float userVerticalUpperLimit() const;
  float userVerticalLowerLimit() const;
  void setUserVerticalLimit(const float lower, const float
  upper);
  AxisLimitType horizontalLimitType() const;
  void setHorizontalLimitType(const AxisLimitType type);
  float userHorizontalUpperLimit() const;
  float userHorizontalLowerLimit() const;
  void setUserHorizontalLimits(const float lower, const float
  upper);
  AxisLimitType receiverHorizontalLimitType(const int index)
  const;
  void setReceiverHorizontalLimitType(const int index, const
  AxisLimitType type);
  float receiverUserHorizontalUpperLimit(const int index)
  const;
  float receiverUserHorizontalLowerLimit(const int index)
  const;
  void setReceiverUserHorizontalLimit(const int index, const
  float lower, const float upper);

  bool isDepthIntervalsDisplayed() const;
  void setDepthIntervalsDisplayed(const bool display);
  float depthIntervalsValue() const;
  void setDepthIntervalsValue(float const value,
  Unit const& unit);

  double paletteMinimumValue() const;
  double paletteMaximumValue() const;
  void setPaletteMinimumValue(const double min);
  void setPaletteMaximumValue(const double max);

  void restoreDefaultPalette();
  void adaptPaletteToLocalLimits();
  void adaptPaletteToGlobalLimits();
  …
};
```

The properties of **`AcousticWaveformTrackItem`** class allow you to:

- get and set the **`displayMode`** through enum class **`WaveDisplayMode`** (VDL or Wiggle)

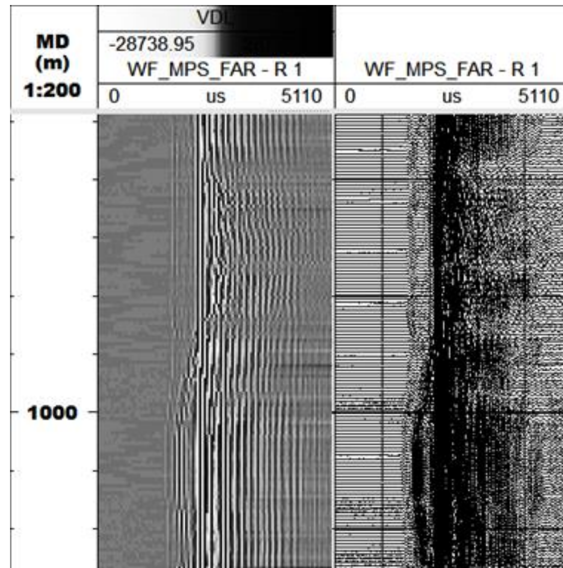


Figure 36 VDL versus Wiggle display mode

- acoustic data are displayed following which receiver and azimuth are activated or displayed in the "Receivers options" tab of the waveform properties window. The **`AcousticWaveformTrackItem`** class available the methods to:
  - get and set the current azimuth and receiver from which the data are displayed in the track item available through **`currentAzimuth`** and **`currentReceiver`** properties
  - if index value of **`currentReceiver`** property is equal to the number of receivers, then all receivers are selected and you can hide and display the receivers through **`setReceiverDisplay`** method
  - in wiggle mode change receiver line thickness and upper and lower area fill colors between receiver lines

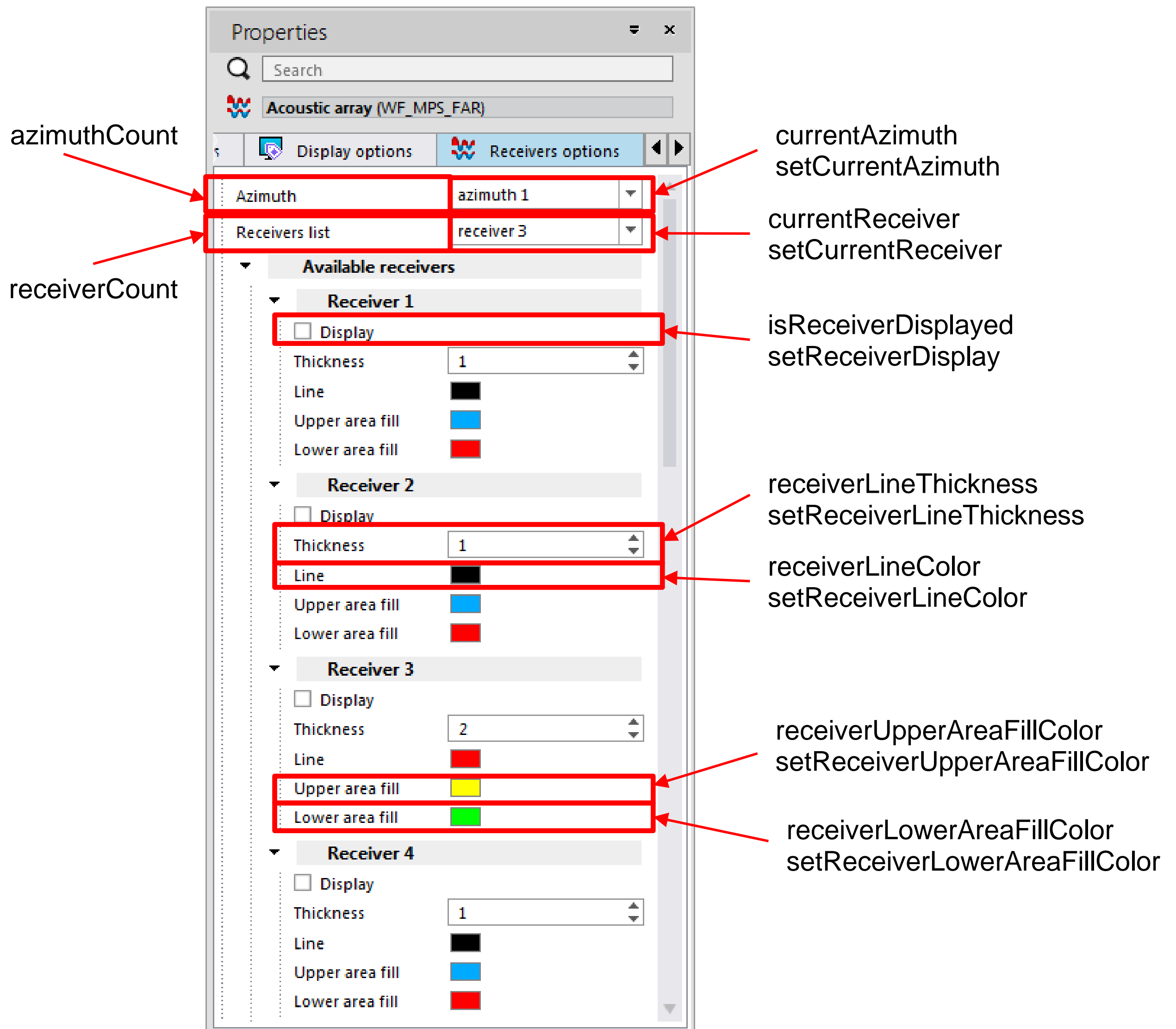


Figure 37 Azimuth and receiver API

Below is an example creating an **AcousticWaveformTrackItem** and changing the current receiver and its color properties.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Logview logview = Logview::create(workspace);

NormalTrack track = NormalTrack::create("track1", logview);

Variable vArray =
DomainObject::get(acousticVariableDroid).cast<Variable>();
if (!vArray.containsDataProperty("TL_WFLEN"))
{
```

```
  // Add a TL_WFLEN acoustic data property to the variable
(mandatory)
  DataProperty wflen_property("TL_WFLEN");
  // waveform length cannot be greater than the number of columns
of the array variable
  wflen_property.setValue(
  QString::number(vArray.columnCount()));

  vArray.addDataProperty(wflen_property);
}

AcousticWaveformTrackItem acousticWFTrackItem =
AcousticWaveformTrackItem::create(track, vArray);
acousticWFTrackItem.setDisplayMode(WaveDisplayModeWiggles);
// Iterate on the list of receivers
for (int i = 0; i < acousticWFTrackItem.receiverCount(); i++)
{
  if (i == acousticWFTrackItem.receiverCount() - 1)
  {
    // Activate the last receiver of the list
    // Note: data displayed in the Track from this receiver, even
if its display property is set to False
    acousticWFTrackItem.setCurrentReceiver(i);
    // Change line thickness of the current receiver
    acousticWFTrackItem.setReceiverLineThickness(i, 2);
    // Change line, upper and lower area fill colors of the current
receiver
    acousticWFTrackItem.setReceiverLineColor(i, Qt::red);
    acousticWFTrackItem.setReceiverUpperAreaFillColor(i,
    Qt::yellow);
    acousticWFTrackItem.setReceiverLowerAreaFillColor(i,
    Qt::green);
    break;
  }
}

lock.release();
```

Display of the **`AcousticWaveformTrackItem`**:

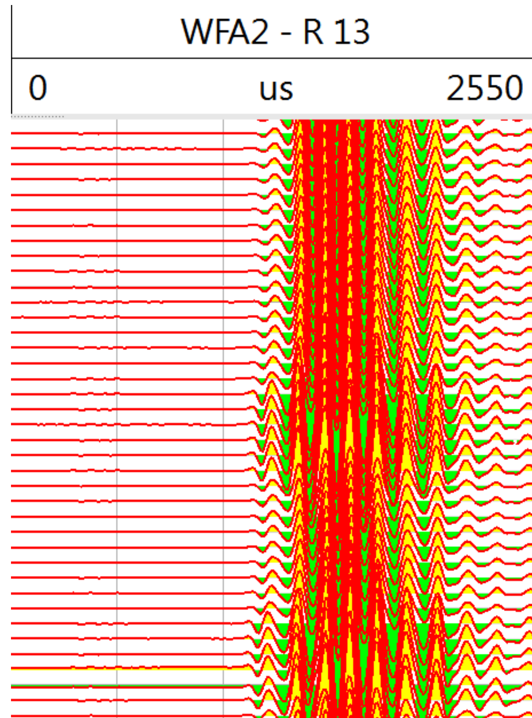


Figure 38 AcousticWaveFormTrackItem current receiver colors changed

- get and set lower and upper horizontal axis limits with **`horizontalLimitType`** turned to **`AxisLimitTypeUser`**
- get and set lower and upper vertical axis limits with **`verticalLimitType`** turned to **`AxisLimitTypeUser`**
- get and set lower and upper horizontal axis limits with **`receiverHorizontalLimitType`** turned to **`AxisLimitTypeUser`** for a given receiver index value
- enable the depth intervals display option and get/set the depth intervals value
  - The **`Unit`** tells the **`AcousticWaveformTrackItem`** which unit must be considered regarding the depth intervals value passed to the function. This value is converted from the **`Unit`** to the current **`Logview`** reference unit and the converted value is set to depth intervals parameter. See "Plot axes limits and display parameters" in *Ocean for Techlog Developer Guide – Basics*.
- get and set limit values of a **`palette`** applied to the **`AcousticWaveformTrackItem`**
- restore the "default" palette assigned to the variable displayed through the **`AcousticWaveformTrackItem`**
  - This API can be used for example right after the **`Logview::setVerticalTopBottomLimits`** call. It results to a palette with a color gradiant that best fit the area of interest.
- adapt the palette min and max boundaries to values of the variable corresponding to the area visible in the **`Logview`**

- adapt the palette min and max boundaries to match all values of the variable (variable min and max)

AcousticMatrixTrackItem Domain Object

`AcousticMatrixTrackItem` domain object inherits from **`AcousticTrackItem`** base class and is used to display in the Logview an array variable (greater than one column) with "acoustic array" type (**`VariableTypeAcousticMatrixArray).`**

Before visualizing an "acoustic array" variable through **`AcousticMatrixTrackItem`** some acoustic data properties for slowness-time projections need to be added to the variable as:

- TL_PKSL: reference to coherence peak slowness variable array name in same dataset for the slowness projection, if this property does not exist when **`AcousticMatrixTrackItem`** is created, the functions related to peak slowness are disabled (they do nothing)
- TL_DT, TL_DT2, TL_DT3: references to associate DT variable names in same dataset for the slowness projection, if these properties do not exist when **`AcousticMatrixTrackItem`** is created, the functions related to associated variables are disabled (they do nothing)
- TL_SLL: slowness lower limit
- TL_SUL: slowness upper limit

If you want to know more about acoustic data properties and how to visualize slowness projection in a Logview, see the "Acoustics Data Preparation" and "Viewing slowness projections in Logview" sections in *Techlog online help*.

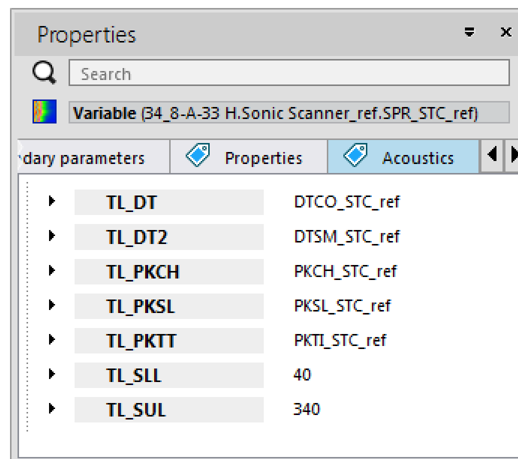


Figure 39 Acoustic properties associated to an "acoustic array" variable

```
class AcousticMatrixTrackItem : public AcousticTrackItem
{
public:

  static AcousticMatrixTrackItem create(NormalTrack track,
  const Variable &variable);

  DomainObjectCollection<Variable> associatedVariables()
```

```
  const;
  QColor associatedVariableColor(const Variable &variable);
  void setAssociatedVariableColor(const Variable &variable,
  const QColor &color);
  LineType associatedVariableConnectionType(const Variable
  &variable) const;
  void setAssociatedVariableConnectionType(const Variable
  &variable, const LineType type);
  LineStyle associatedVariableConnectionLineStyle(const
  Variable &variable) const;
  void setAssociatedVariableConnectionLineStyle(const
  Variable &variable, const LineStyle style);
  int associatedVariableConnectionLineThickness(const Variable
  &variable) const;
  void setAssociatedVariableConnectionLineThickness(const
  Variable &variable, const int thickness);

  bool isPeakSlownessDisplayed() const;
  void setPeakSlownessDisplayed(const bool displayed);
  QColor peakSlownessColor() const;
  void setPeakSlownessColor(const QColor &color);
  PointType peakSlownessPointType() const;
  void setPeakSlownessPointType(PointType type);
  int peakSlownessPointSize() const;
  void setPeakSlownessPointSize(const int size);

  double paletteMinimumValue() const;
  double paletteMaximumValue() const;
  void setPaletteMinimumValue(const double min);
  void setPaletteMaximumValue(const double max);

  void restoreDefaultPalette();
  void adaptPaletteToLocalLimits();
  void adaptPaletteToGlobalLimits();
  …
};
```

Properties of **`AcousticMatrixTrackItem`** class allow you to:

- if TL_DT acoustic properties are defined, you can iterate through the associated variables and change some properties as connection line type, style and thickness

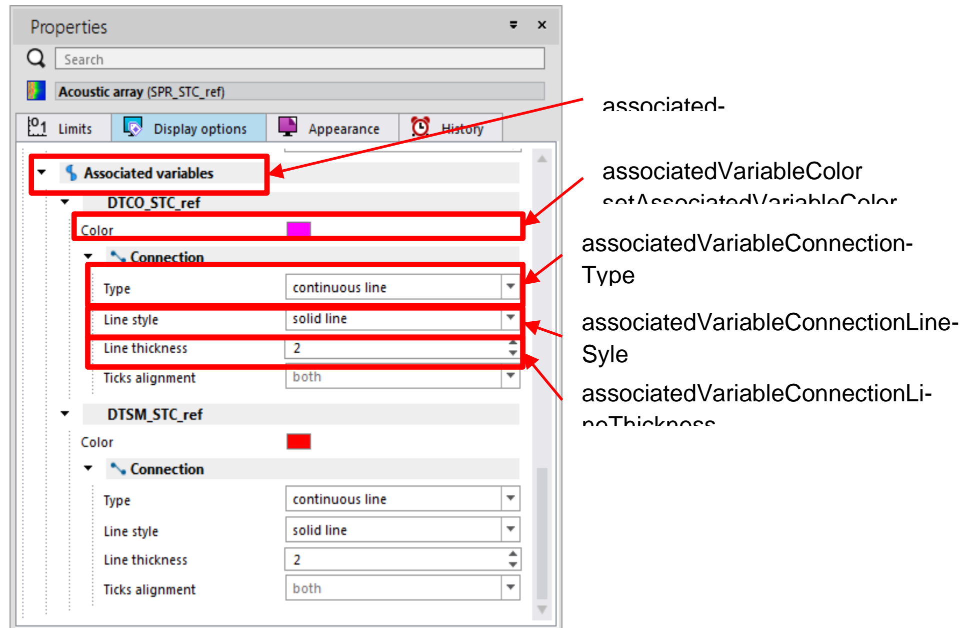


Figure 40 AcousticMatrixTrackItem associated variables display options

- if TL_PKSL acoustic property is defined, some APIs give the ability to change peak slowness display options as color, type, point type and point size

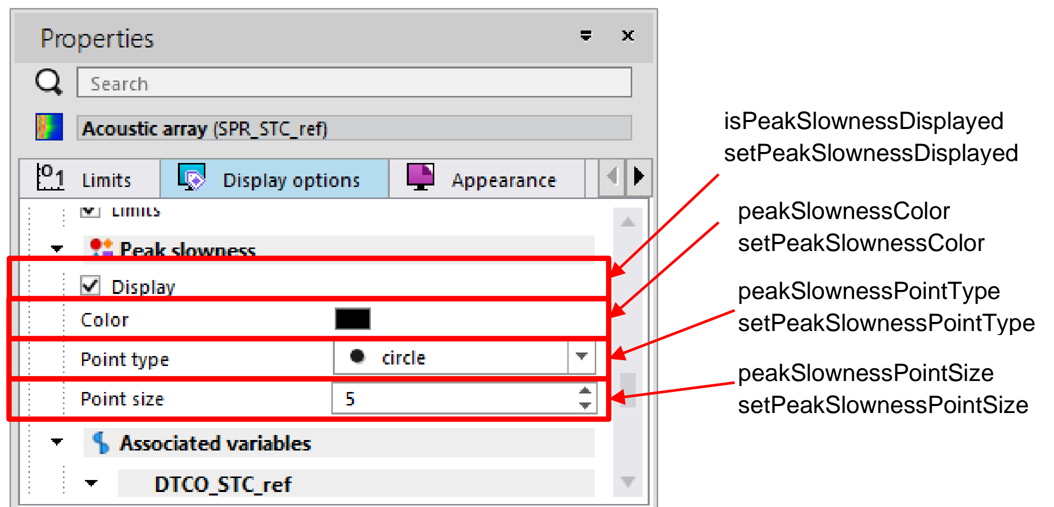


Figure 41 AcousticMatrixTrackItem peak slowness display options

The following example shows the change of the display properties of an associated variable of the "acoustic array" variable.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Logview logview = Logview::create(workspace);

NormalTrack track = NormalTrack::create("track1", logview);
```

```
Variable vArray =
DomainObject::get(acousticVariableDroid).cast<Variable>();

AcousticMatrixTrackItem acousticMatrixTrackItem =
AcousticMatrixTrackItem::create(track, vArray);

foreach (Variable var,
acousticMatrixTrackItem.associatedVariables())
{
  // set the display properties of the associated variable
  if (var.name() == "DTCO_STC_ref")
  {
    acousticMatrixTrackItem.setAssociatedVariableColor(var,
    Qt::green);
    acousticMatrixTrackItem.
    setAssociatedVariableConnectionLineStyle(var,
    LineStyleSolid);
    acousticMatrixTrackItem.
    setAssociatedVariableConnectionLineThickness(var, 2);
    acousticMatrixTrackItem.
    setAssociatedVariableConnectionType(
    var, LineTypePointToPoint);
    break;
  }
}

lock.release();
```

Display of the **`AcousticMatrixTrackItem`**:

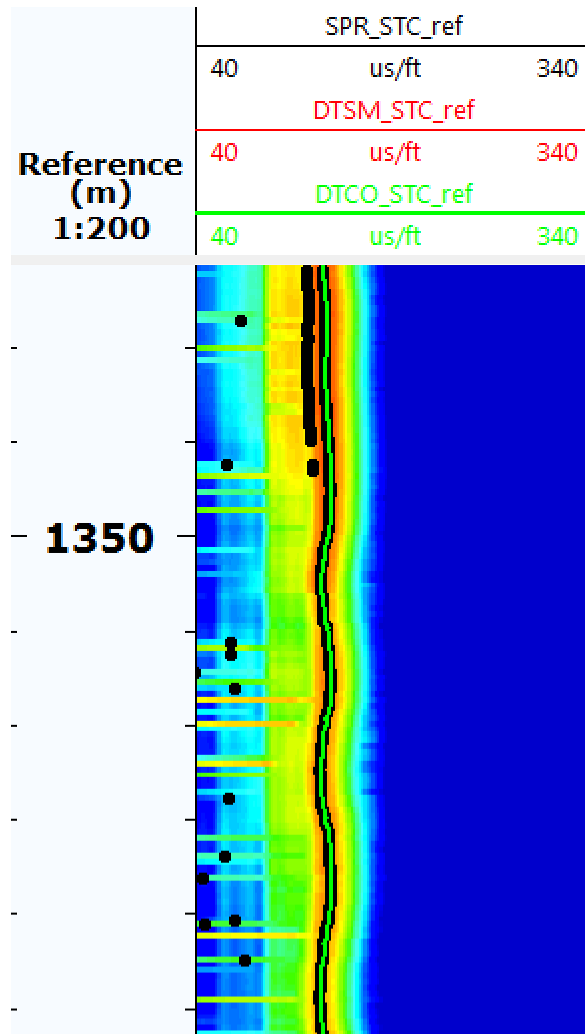


Figure 42 AcousticMatrixTrackItem associated variable display changed

DipTrackItem Domain Object

See the "DipTrackItem domain object" section in in *Ocean for Techlog Developer Guide – Geology* for more information on how to display dip variable in a track of the Logview with Ocean.

ImageTrackItem Domain Object

Image variables created in Ocean with **`ImageVariable`** domain object can be displayed in a track of the Logview through an **`ImageTrackItem`** domain object.

See "ImageVariable" section in in *Ocean for Techlog Developer Guide – Data&Workflow* for more information on how to create a core image or a borehole image variable.

```
class ImageTrackItem : public TrackItem
{
public:
  static ImageTrackItem create(NormalTrack track, ImageVariable
  variable);
  …
```

```
};
```

BHATrackItem Domain Object

The Bottom Hole Assembly (BHA) module provides information about the tools in the hole during acquisition. This allows you to visualize the distribution of sensors in time and space.

BHA is built in Techlog through the BHA builder.

To access the BHA builder, go to the **Petrophysics** tab > **TLA** group > **BHA builder** or **Drilling** tab > **Drilling analysis** group > **BHA builder**.

Please refer to Techlog documentation for more information on how to build a BHA.

After the BHA is built, the saved BHA can be display in a logview window along with other curves to visualize the measure point offsets of the different LWD (Logging While Drilling) measurements. This can be done programmatically in Ocean through **`BHATrackItem`** domain object.

```
class BHATrackItem : public TrackItem
{
public:
  static BHATrackItem create(NormalTrack track, Dataset dataset);
  …
};
```

Note: The parent **`NormalTrack`** domain object passed to the **`BHATrackItem::create`** static function must be created with an associated **`Well`**.

```
class NormalTrack : public Track
{
public:
  static NormalTrack create(const QString &name, const Logview
&logview,
  const Well &well);
  static NormalTrack create(const QString &name, const Logview
&logview,
  const Well &well, cont int position);
  …
};
```

All BHA display parameters are exposed as:

- shows and hides sensor and stabilizer labels
- sets the alignement of the BHA as left, center or right in the track
- sets the BHA image width ratio between 0 and 0.5
- sets the index of the displayed BHA (a BHA dataset can contain more than one BHA)
- set the BHA to a fixed reference value
    - The **`Unit`** tells the **`BHATrackItem`** which unit has to be considered regarding the fixed reference value passed to the function. This value is converted from the **`Unit`** to the current **`Logview`** reference unit and the converted value is set to fixed reference parameter. See “Plot axes limits and display parameters” in *Ocean for Techlog Developer Guide – Basics*.

Note: By default **`isFixedReference`** is false and the BHA image is displayed at the bottom view position of the Logview. If the user scrolls, the BHA is anchored to the Logview bottom view position. When **`isFixedReference`** is true, BHA reference is fixed at the last known bottom view position reference value.

```
class BHATrackItem : public TrackItem
{
public:
  void setSensorLabelsVisible(const bool visible);
  bool isSensorLabelsVisible() const;
  void setStabilizerLabelsVisible(const bool visible);
  bool isStabilizerLabelsVisible() const;
  void setAlignment(const BHAAlignment alignment);
  BHAAlignment alignment() const;
  void setFixedReference(const bool fixed);
  bool isFixedReference() const;
  void setFixedReferenceValue(const double value,
  const Unit &unit);
  double fixedReferenceValue() const;
  void setImageWidthRatio(const double ratio);
  double imageWidthRatio() const;
  void setCurrentBHAIndex(const int index);
  int currentBHAIndex() const;
};
```

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();
Well well = project.findWell("well");

if (well.isNull())
{
  lock.release();
  return;
}

Dataset dataset = well.findDataset("ecoscope");
Variable rssc = dataset.findVariable("RSSC");
Variable robb = dataset.findVariable("ROBB");

Dataset datasetBHA = well.findDataset("BHA");
```

```
Workspace workspace = Session::current().currentWorkspace();

Logview logview = Logview::create(workspace);

NormalTrack track1 = NormalTrack::create("track1", logview);
ArrayTrackItem::create(track1, rssc);
NormalTrack track2 = NormalTrack::create("track2", logview);
LineTrackItem::create(track2, robb);

NormalTrack track3 = NormalTrack::create("track3", logview,
well);
BHATrackItem bhaTrackItem = BHATrackItem::create(track3,
datasetBHA);

bhaTrackItem.setImageWidthRatio(0.5);

bhaTrackItem.setFixedReference(true);
// Converts value from unit to current logview reference unit
bhaTrackItem.setFixedReferenceValue(3550.92, "m");

lock.release();
```

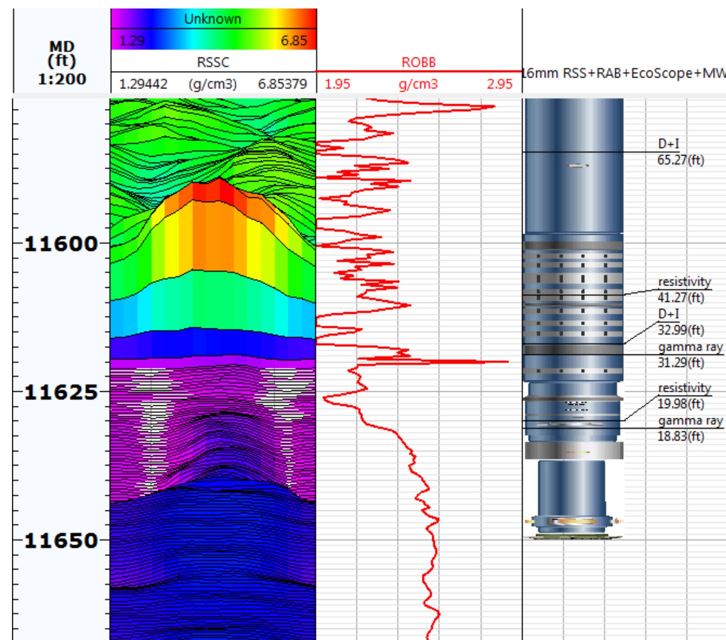


Figure 43 BHA image displayed in a track

WellSchematicTrackItem Domain Object

The Well schematic feature displays well tools and drilling action within every track type, even the reference one.

To insert a well schematic:

1. Select the track where you want to insert the well schematic.
2. Right-click and select "Well Schematic" menu item in the contextual menu. The Well Schematic builder opens and you can select objects to insert as tubular tools and downhole objects.

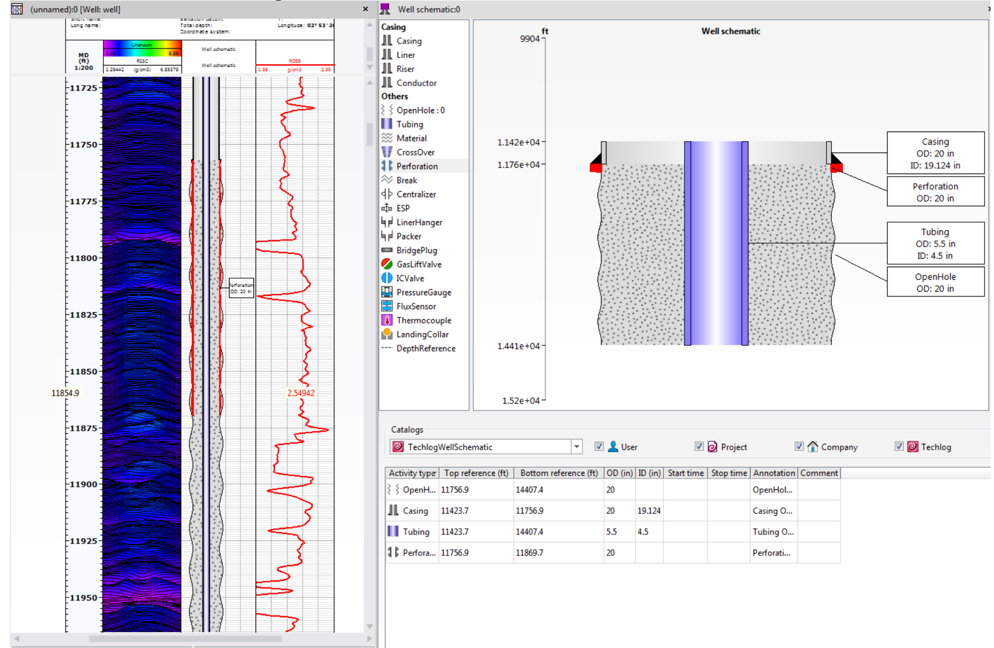


Figure 44 Well Schematic builder

3. Save the Well Schematic.
4. In the Logview select the Well Schematic, right-click and select the "Save well schematic" menu item. This creates a well schematic dataset in the project browser under the well associated to the track and containing all the schematics information.

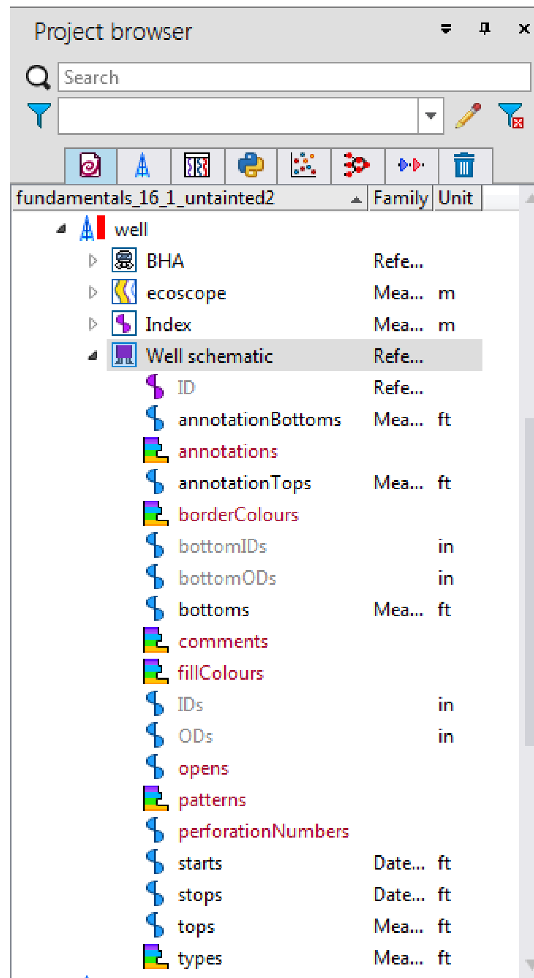


Figure 45 Well Schematic dataset

Ocean allows you to display this Well Schematic dataset in a track of the Logview through the **`WellSchematicTrackItem`** domain object.

```
class WellSchematicTrackItem : public TrackItem
{
public:
  static WellSchematicTrackItem create(NormalTrack track,
  Dataset dataset);
  …
};
```

Note: The parent **`NormalTrack`** domain object passed to the **`WellSchematicTrackItem::create`** static function must be created with an associated **`Well`**.

```
class NormalTrack : public Track
{
public:
  static NormalTrack create(const QString &name, const Logview
    &logview, const Well &well);
  static NormalTrack create(const QString &name, const Logview
```

```
    &logview, const Well &well, cont int position);
  …
};
```

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();
Well well = project.findWell("well");

if (well.isNull())
{
  lock.release();
  return;
}

Dataset dataset = well.findDataset("ecoscope");
Variable rssc = dataset.findVariable("RSSC");
Variable robb = dataset.findVariable("ROBB");

Dataset datasetWellSchematic = well.findDataset("Well
schematic");

Workspace workspace = Session::current().currentWorkspace();

Logview logview = Logview::create(workspace);

NormalTrack track1 = NormalTrack::create("track1", logview);
ArrayTrackItem::create(track1, rssc);
NormalTrack track2 = NormalTrack::create("track2", logview);
LineTrackItem::create(track2, robb);

NormalTrack track3 = NormalTrack::create("track3", logview, well,
1);

WellSchematicTrackItem wellSchematicTrackItem =
WellSchematicTrackItem::create(track3, datasetWellSchematic);

logview.setVerticalTopBottomPosition(11725, 11850, "ft");

lock.release();
```

Track template

In the **`NormalTrack`** class **`create`** static methods are availables to instanciate a **`NormalTrack`** object passing a track template saved at a storage level.

```
class NormalTrack : public Track
{
public:

  …
  static NormmalTrack create(const QString &name, const
    TrackTemplate &trackTemplate, const Logview &logview, const
    Well &well);

  static NormalTrack create(const QString &name, const
    TrackTemplate &trackTemplate, const Logview &logview, const
    Dataset &dataset);
  …
}
```

In Techlog after setting a track in the Logview, save the track as a template.

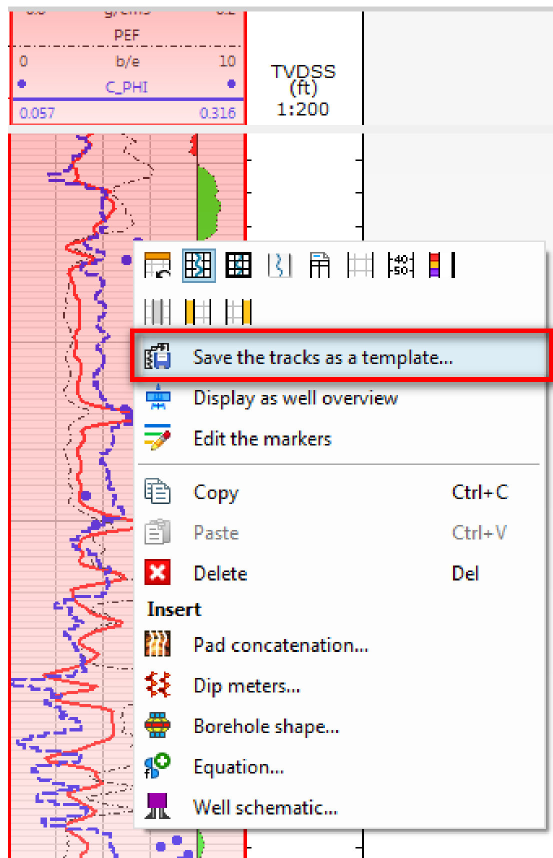


Figure 46 Track saved as a template

Note: The difference between a track and a template is that a template saves only the content of a track, whereas a track saves specific variables and the complete display.

You will specify the template name and at the level where is stored the template, modeled in Ocean with **`TrackTemplate`** class and enum class **`StorageLevel`**.

```
class TrackTemplate
{
public:
   static TrackTemplate get(const StorageLevel level, const
   QString &name);
   static bool exists(const StorageLevel level, const QString
   &name);

   StorageLevel level() const;
   QString name() const;
    …
}
```

Ocean introduces an additional level which is the plug-in level and the ability to provide track templates in the plug-in folder. At the plug-in level the layout template must be stored in a directory named "TemplatesTrack" closed to the plug-in dll in the plug-in folder.

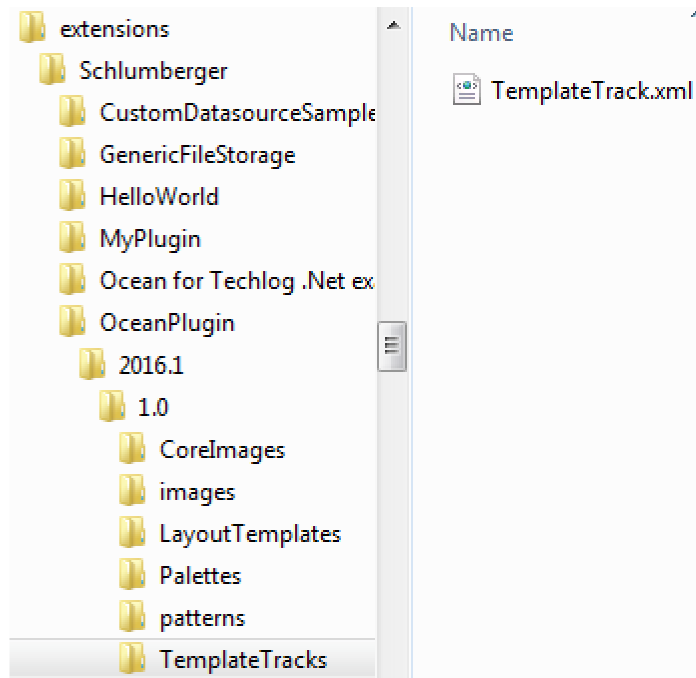


Figure 47 TemplatesTrack folder that contains track templates at the plug-in level

Ocean through the **`NormalTrack::create`** methods passing a Track**`Template`** instance (containing template name and storage level) allows you to display the logs saved in the template for a well or a dataset in a new **`NormalTrack`** added to **`Logview`**.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Well well = DomainObject::get(wellDroid).cast<Well>();

Workspace workspace = Session::current().currentWorkspace();
```

```
if (TrackTemplate::exists(StorageLevelPlugin, "TemplateTrack"))
{
  Logview logview = Logview::create(workspace);

  TrackTemplate trackTemplate =
  TrackTemplate::get(StorageLevelPlugin, "TemplateTrack");

  NormalTrack::create("track1", trackTemplate, logview, well);
}

lock.release();
```

Note: The possibility to create a **`NormalTrack`** through a template is a very convenient workaround for track in Techlog that are not exposed with Ocean as separation track or annotation track.

Area fills

There are two types of area fill available in Techlog:

- An area fill that handle only one color or pattern displayed in the track exposed in Ocean through the **`SingleAreafill`** domain object. This is the case for an area fill created between a single curve (**`LineTrackItem`**) and a limit (lower or upper) of the track.
- An area fill that handle two colors or patterns displayed in the track exposed in Ocean through the **`DoubleAreafill`** domain object. This is the case for area fills created between:
    - a **`Baseline`** and lower and upper single curve values
    - two single curves

Areafill Domain Object

The area fill objects exposed with Ocean are grouped under the **`Areafill`** base class.

```
class Areafill : public DomainObject
{
public:
  FillType fillType() const;
  void setFillType(const FillType fillType);

  QString paletteName() const;
  void setPalette(const QString &paletteName,
    const StorageLevel level);

  QString title() const;
  void setTitle(const QString &name);


  bool isHeaderVisible() const;
```

```
  void setHeaderVisible(bool visible);
};
```

You cannot create an **Areafill** object. This class holds some common properties to all area fill objects as:

- the type used to fill the area as a color, a pattern, or a palette
- The palette name used to fill the area. For **DoubleAreafill** only one palette is applied for left and right sides. This is why **paletteName** is a common property holds by **Areafill** base class.
- the title of the area fill displayed in the header of the track
- show / hide the area fill displayed in the header of the track

SingleAreafill Domain Object

A **SingleAreafill** is used to create an area fill between a **LineTrackItem** and lower or upper limit of the track.

```
class SingleAreafill : public AreaFill
{
public:
  static SingleAreafill create(LineTrackItem &lineTrackItem);

  void setLimitType(const LimitType limitType);
  void setColor(const QColor &color);
  void setPattern(const QString &patternName,
  const StorageLevel level);
};
```

The **SingleAreafill** is a domain object and therefore is instantiated through the **create** static method passing the parent **LineTrackItem** as argument of the function. Then you must set a limit type (lower or upper), a fill type and the corresponding color, pattern, or palette.

Note: The patterns and palettes cannot be stored at the plug-in level.

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Variable dens =
DomainObject::get(variableDroid).cast<Variable>();

Logview logview = Logview::create(workspace);
NormalTrack track1 = NormalTrack::create("track1", logview);
LineTrackItem lineTrackItem = LineTrackItem::create(track1,
dens);

SingleAreafill lowerAreaFill =
SingleAreafill::create(lineTrackItem);
lowerAreaFill.setTitle("Lower area fill color");
```

```
lowerAreaFill.setLimitType(LimitTypeLowerLimits);
lowerAreaFill.setFillType(FillTypeColor);
lowerAreaFill.setColor(QColor(255, 165, 0, 100));

SingleAreafill upperAreaFill =
SingleAreafill::create(lineTrackItem);
upperAreaFill.setTitle("Upper area fill pattern");
upperAreaFill.setFillType(FillTypePattern);
upperAreaFill.setLimitType(LimitTypeUpperLimits);
upperAreaFill.setPattern("psychedelic", StorageLevelUser);

NormalTrack track2 = NormalTrack::create("track2", logview);
LineTrackItem lineTrackItem2 = LineTrackItem::create(track2,
dens);

SingleAreafill lowerAreaFill2 =
SingleAreafill::create(lineTrackItem2);
lowerAreaFill2.setTitle("Lower area fill palette");
lowerAreaFill2.setLimitType(LimitTypeLowerLimits);
lowerAreaFill2.setFillType(FillTypePalette);
lowerAreaFill2.setPalette("FACIES_10_USER", StorageLevelUser);

SingleAreafill upperAreaFill2 =
SingleAreafill::create(lineTrackItem2);
upperAreaFill2.setTitle("Upper area fill pattern");
upperAreaFill2.setFillType(FillTypePattern);
upperAreaFill2.setLimitType(LimitTypeUpperLimits);
upperAreaFill2.setPattern("Shale", StorageLevelTechlog);

lock.release();
```

The screenshot below shows the result:

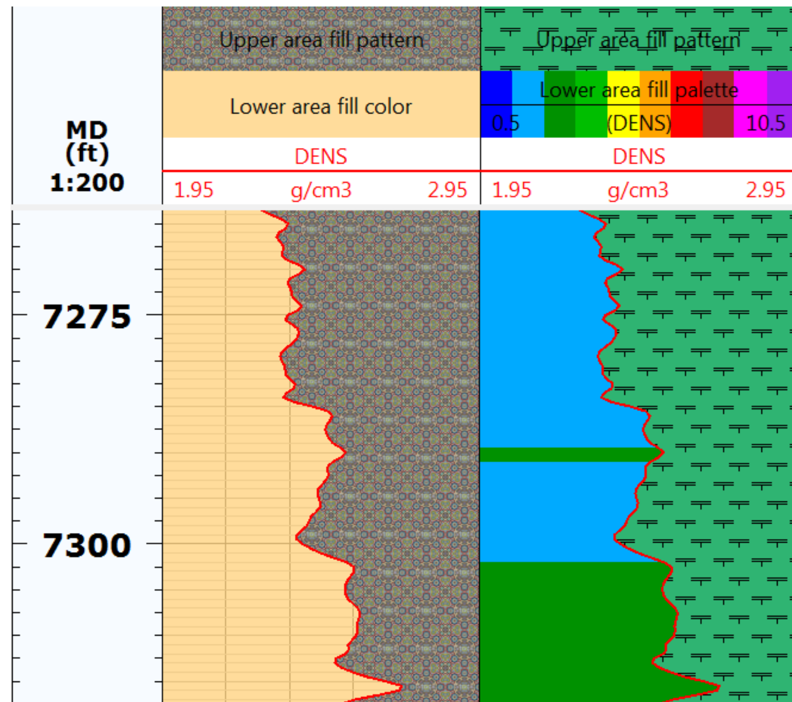


Figure 48 Single area fills with different types

DoubleAreafill Domain Object

A **`DoubleAreafill`** is used to create an area fill between two **`LineTrackItem`** or a **`LineTrackItem`** and a **`Baseline`**.

```
class DoubleAreafill : public AreaFill
{
public:
  static DoubleAreafill create(LineTrackItem &lineTrackItem,
  const Baseline &baseline);
  static DoubleAreafill create(LineTrackItem &lineTrackItem1,
  LineTrackItem &lineTrackItem2);

  void setLeftFillType(const FillType fillType);
  FillType leftFillType() const;
  QColor leftColor() const;
  void setLeftColor(const QColor &color);
  QString leftPatternName() const;
  void setLeftPattern(const QString &patternName, const
  StorageLevel level);

  void setRightFillType(const FillType fillType);
  FillType rightFillType() const;
  QColor rightColor() const;
  void setRightColor(const QColor &color);
  QString rightPatternName() const;
  void setRightPattern(const QString &patternName, const
  StorageLevel level);
};
```

The **DoubleAreafill** is a domain object and therefore is instantiated through dedicated **create** static method depending if the double area fill is created between two **LineTrackItem** or a **LineTrackItem** and a **Baseline.** Then you must set fill type and the corresponding color or pattern for right and left sides of the double area fill.

Example of **DoubleAreafill** created between two **LineTrackItem**:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Variable dens =
DomainObject::get(variableDensDroid).cast<Variable>();
Variable neut =
DomainObject::get(variableNeutDroid).cast<Variable>();

Logview logview = Logview::create(workspace);
NormalTrack track = NormalTrack::create("track1", logview);
LineTrackItem lineTrackItem1 = LineTrackItem::create(track,
dens);
LineTrackItem lineTrackItem2 = LineTrackItem::create(track,
neut);

DoubleAreafill doubleAreaFill =
DoubleAreafill::create(lineTrackItem1, lineTrackItem2);
doubleAreaFill.setTitle("My double area fill");
doubleAreaFill.setLeftFillType(FillTypeColor);
doubleAreaFill.setLeftColor(Qt::blue);
doubleAreaFill.setRightFillType(FillTypePattern);
doubleAreaFill.setRightPattern("Shale", StorageLevelTechlog);

lock.release();
```

The following screenshot shows the result where:

- Blue color (**leftColor**) is displayed when **LineTrackItem** that plots the neutron variable has values lower than **LineTrackItem** that plots the density variable.
- Shale pattern (**rightPatternName**) is displayed when **LineTrackItem** that plots the neutron variable has values greater than **LineTrackItem** that plots the density variable.

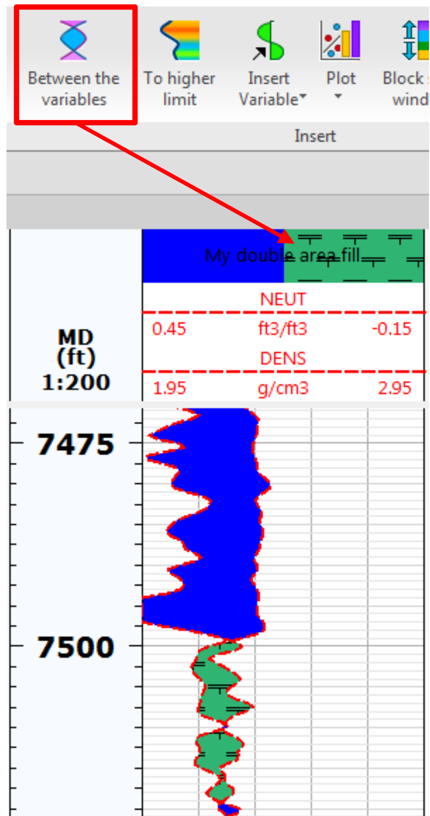


Figure 49 DoubleAreafill between two LineTrackItem's

Example of **`DoubleAreafill`** created between a **`LineTrackItem`** and a **`Baseline`**:

```
lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, lineTrackItem1);
Baseline baseline(2.50f);
lineTrackItem1.setBaselines(QList<Baseline>() << baseline);
lock.release();

lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, lineTrackItem1);
DoubleAreafill doubleAreaFill2 =
DoubleAreafill::create(lineTrackItem1, baseline);
doubleAreaFill2.setTitle("Baseline area fill");
doubleAreaFill2.setFillType(FillTypePalette);
doubleAreaFill2.setPalette("FACIES_5", StorageLevelTechlog);
lock.release();
```

The following screenshot shows the result with the same rules mentioned previously:

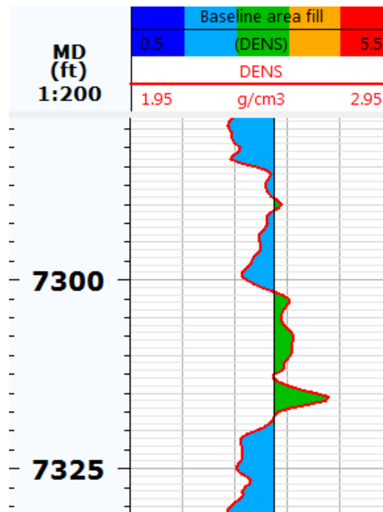


Figure 50 DoubleAreaFill between a LineTrackItem and a Baseline

Logview display options

Some **Logview** display options are available through following API:

```
class Logview : public Plot
{
public:
  …
   void setVerticalTopBottomPosition(const double top, const
  double bottom, const Unit &unit);
  void setVerticalTopBottomLimits(Well well, const double top,
  const double bottom, const Unit &unit);
  void setGlobalTopBottomLimits(const double top, const double
  bottom, const Unit &unit);
  void clearGlobalTopBottomLimits();

  bool isVertical() const;
  void setVertical();
  bool isHorizontal() const;
  void setHorizontal();

  const GlobalZonation findSelectedZonation() const;
  const QList<GlobalZone> selectedZones() const;
  void setSelectedZones(const GlobalZonation
  &selectedZonation, const QList<GlobalZone> &selectedZones);

  void setMainHeaderVisible(const bool &visibility);
  bool isMainHeaderVisible() const;
  void setWellHeaderVisible(const bool &visibility);
  bool isWellHeaderVisible() const;
  void setMainHeader(const Header &header);
  void setWellHeader(const Header &header);

  bool isFamilyPropertiesEnabled() const;
  void setFamilyPropertiesEnabled(bool state);
```

```
  LogviewVariableHeaderTitleFormat variableHeaderTitleFormat()
   const;
  void setVariableHeaderTitleFormat(const
    LogviewVariableHeaderTitleFormat &titleFormat);

  void setMouseMode(const MouseMode mouseMode);
  MouseMode mouseMode() const;
  bool canSetMouseMode(const MouseMode mouseMode) const;

  void reloadAllVariables();
  void swapWell(const Well well);

  ReturnValue<bool> canSetReferenceSpace(const Family
  &referenceSpace);
  Family referenceSpace() const;
  void setReferenceSpace(const Family &referenceSpace);

  double minimumVisibleReferenceValue(const Well well) const;
  double maximumVisibleReferenceValue(const Well well) const;

  void setCurrentDepth(double currentDepth,
  SendDepthInteractionEvent sendDepthInteractionEvent, const
  Unit &unit, const Well &well);
  double currentDepth() const;
  void adjustHorizontally();
  void adjustVertically();
  void adjustGlobally();
  void adjustToOriginalScale();
  void setAutomaticAdjustmentEnabled(const bool enabled);
  bool isAutomaticAdjustmentEnabled() const;
  bool isDepthListenerEnabled() const;
  void setDepthListenerEnabled(bool enabled);

  void setDepthScale(const float scaleDenominator);
  void setTimeScale(const LogviewScaleDistanceUnit distanceUnit,
   const int equivalence, const LogviewScaleTimeUnit timeUnit);
};
```

- **setVerticalTopBottomPosition** – adjust top and bottom view position (cannot be called before the first commit after creating the **Logview**)
- **setVerticalTopBottomLimits** – adjust top and bottoms view limits per well (cannot be called before the first commit after creating the **Logview**)
- **setGlobalTopBottomLimits** - adjust top and bottoms view limits for all wells (cannot be called before the first commit after creating the **Logview**)
- **setSelectedZones** give the ability to select some **GlobaZone** in a **GlobalZonation** and display its in the **Logview**
- ability to show/hide **Logview** headers
- enable / disable display properties related to the family applied to the variable
- get and set the format of the variable title in the **Logview** header

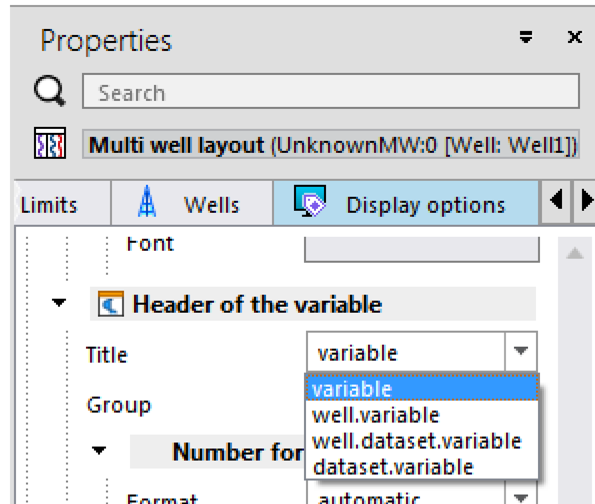


Figure 51 Format of the variable title in the header

- get and set the current **`Logview`** mouse mode (check first if the mouse mode can be applied to the Logview calling the **`canMouseMode`** function)
- set the Logview reference space for instance toTrue Vertical Depth. The list of supported reference spaces is available in **`ReferenceSpace`** namespace. The reference space has to be set at the **`Logview`** creation. The **`canSetReferenceSpace`** function returns false if the reference space can't be applied to the **`Logview`**.
- get minimum and maximum reference values visible in the plot area for the given well
- get and set the current depth to the Logview. Through the setter you can specify the current depth value unit passed to the function (converted to the reference unit if compatible) and if the **`DepthInteractionChanged`** signal of the **`Workspace`** class has to be emitted when the current depth is changed (See "DepthInteractionChanged signal" in *Ocean for Techlog Developer Guide – Basics*).
- adjust horizonatlly, vertically, or on both axes the well logs displayed in the Logview
- adjust automatically the Logview display each time a track is inserted or deleted
- change the **`Logview`** vertical scale ratio using **`setDepthScale`** or **`setTimeScale`** functions depending if the **`Logview referenceUnit`** is expressed in depth or time

Note: For **`setVerticalTopBottomPosition`**, **`setVerticalTopBottomLimits`** and **`setGlobalTopBottomLimits`** functions; the **`Unit`** tells the **`Logview`** which unit must be considered regarding position and limits values passed to functions. Those values are converted from the **`Unit`** to the **`Logview`** reference unit and converted values are set to display parameters. See "Plot axes limits and display parameters" in *Ocean for Techlog Developer Guide – Basics*.

SLB-Private

Plot group into a track

Ocean exposes the ability to create a group of plots along a **NormalTrack** of the **Logview**. The types of plots that can display **Variable** data in a **NormalTrack** of the **Logview** with Ocean are:

1. The cross-plot that allows you to compare single curve variable data at a single reference.
2. The dispersion plot

These groups of plots into a **NormalTrack** are handled by different Ocean classes grouped under the **PlotGroup** base class. The class diagram shows these different classes:

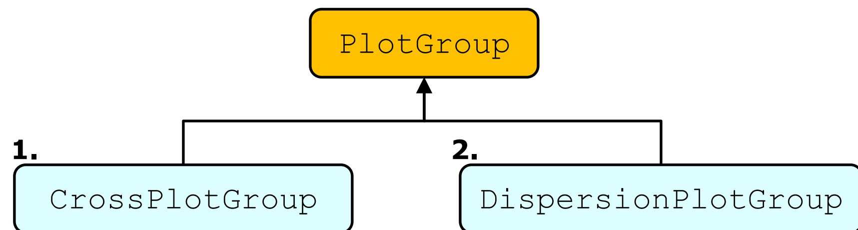


Figure 52 PlotGroup class diagram

The **PlotGroup** class holds properties common to all group of plots that you can add to a **NormalTrack** to the **Logview**.

```
class PlotGroup : public DomainObject
{
public:
  NormalTrack normalTrack() const;

  bool isLocked() const;
  void setLocked(bool);
  …
};
```

The **PlotGroup** class allows you to:

- get the parent **NormalTrack** of the **PlotGroup**
- get and set the lock state of the **PlotGroup**. If locked, the user cannot select one plot into the group to modify its properties.

CrossPlotGroup domain object

A **CrossPlotGroup** domain object is added to a **NormalTrack** through a **CrossPlotGroupDefinition** object. This class allows you through its constructor to create the definition of the **CrossPlotGroup** domain object.

```
class CrossPlotGroupDefinition
{
public:
  CrossPlotGroupDefinition(NormalTrack track, TrackItem
   trackItemX, TrackItem trackItemY);
```

```
  void setColor(TrackItem trackItemColor);
  void setDepthIntervals(float topDepth, float bottomDepth,
    float stepDepth, Unit unit);
}
```

Arguments passed to the constructor are mandatory to create a group of cross-plots along a **NormalTrack** of the **Logview**:

1) The **NormalTrack** has been created with an associated **Well**

   See “NormalTrack domain object” section on page 23 for more information on how to create a **NormalTrack** domain object.

2) **TrackItem** instances passed to the constructor are used to plot Variable data on X and Y axes of the cross-plot. Those **TrackItem**’s have to be **LineTrackItem** or **DipTrackItem**. They are already plotted in a **NormalTrack** of the same **Logview** and belong to the **Well** that is used to create the **NormalTrack** passed as first argument of the **CrossPlotGroupDefinintion** constructor.

   See “LineTrackItem Domain Object” section on page 35 for more information on how to create a **LineTrackItem** domain object.

   See “DipTrackItem Domain Object” section on page 66 for more information on how to create a **DipTrackItem** domain object.

A **NormalTrack** can’t contain more than one **CrossPlotGroup** domain object at the time. This can be checked using the **findCrossPlotGroup** function of the **NormalTrack** class.

```
class NormalTrack : public Track
{
public:
  CrossPlotGroup findCrossPlotGroup() const;
  …
};
```

Once the **CrossPlotGroupDefinition** object has been instantiated through its constructor you can create more than a cross-plot in the group using the **setDepthIntervals** function. This function allows you to set the top and bottom depth interval within cross-plots ared displayed in the group and set the step depth to control cross-plots display along the interval.

The **setColor** allows you to set a **Variable** (plotted through a **TrackItem** in the same **Logview** and same **Well**) on the color axis of the cross-plot.

The **CrossPlotGroupDefinition** object is passed to the **create** static function of the **CrossPlotGroup** class in order to create the group of cross-plots into the **NormalTrack**.

```
class CrossPlotGroup : public PlotGroup
{
public:
  static CrossPlotGroup create(const CrossPlotGroupDefinition
    &definition);

  TrackItem trackItemX() const;
  TrackItem trackItemY() const;
  TrackItem findTrackItemColor() const;
  TrackItem trackItemColor() const;
```

```
};
```

`TrackItem` objects used to plot variables on X, Y and color axes of the cross-plot can be retrieved using corresponding getters.

Shown below is an example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();
Workspace workspace = Session::current().currentWorkspace();

Well well = project.findWell("Well1");
if (well.isNull())
{
  lock.release();
  return;
}

Dataset dataset = well.findDataset("DATAFULL");
if (dataset.isNull())
{
  lock.release();
  return;
}

Variable neut = dataset.getVariable("NEUT");
Variable dens = dataset.getVariable("DENS");
Variable gr = dataset.getVariable("GR");
if ((neut.isNull()) || (dens.isNull()) || (gr.isNull()))
{
  lock.release();
  return;
}

Logview logview = Logview::create(workspace);

NormalTrack normalTrack = NormalTrack::create("track1",
logview);
LineTrackItem neutTrackItem = LineTrackItem::create(normalTrack,
neut);
LineTrackItem densTrackItem = LineTrackItem::create(normalTrack,
dens);
LineTrackItem grTrackItem = LineTrackItem::create(normalTrack,
gr);

normalTrack.setHorizontalGridDisplay(TrackGridDisplayNone);
normalTrack.setVerticalGridDisplay(TrackGridDisplayNone);
```

SLB-Private

```
NormalTrack normalTrack2 = NormalTrack::create("track2",
logview, well);

CrossPlotGroupDefinition cpGroupDef =
CrossPlotGroupDefinition(normalTrack2, neutTrackItem,
densTrackItem);
cpGroupDef.setDepthIntervals(2270.76, 2293.62, 7.62, "m");
cpGroupDef.setColor(grTrackItem);

CrossPlotGroup cpGroup = CrossPlotGroup::create(cpGroupDef);

lock.release();
```

The screenshot below shows the result of the sample code. You can note that depth values in meters passed to the **setDepthIntervals** function are converted to the **Logview** reference display unit.

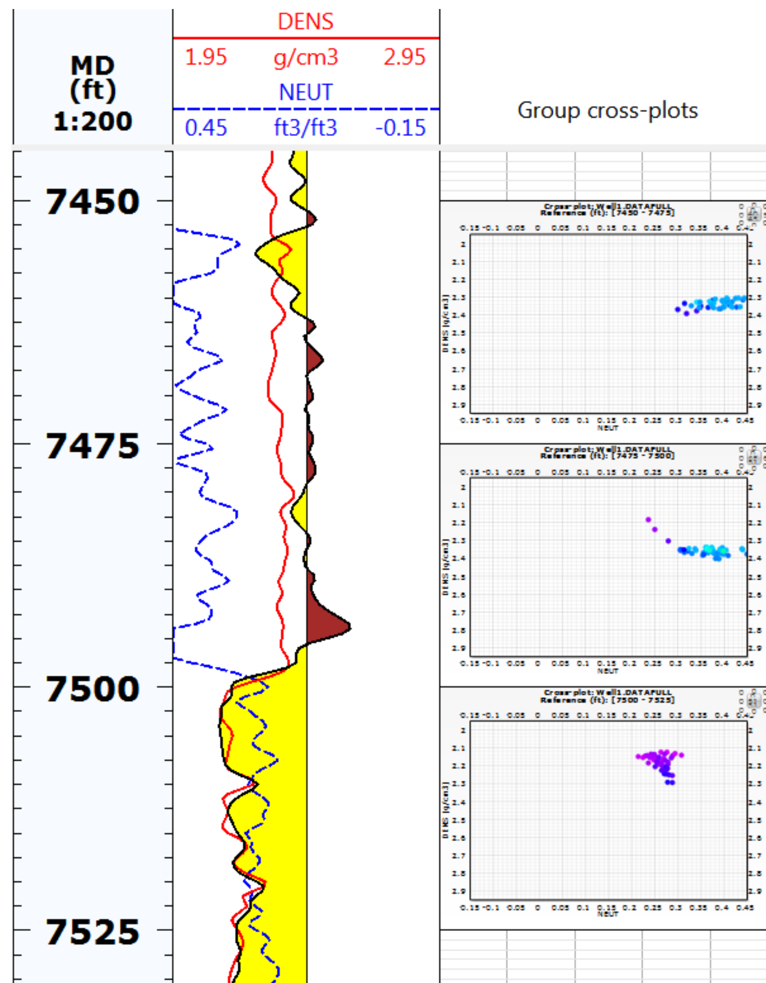


Figure 53 group of cross-plots into a track

SLB-Private

DispersionPlotGroup domain object

A **DispersionPlotGroup** domain object is added to a **NormalTrack** through a **DispersionPlotGroupDefinition** object. This class allows you through its constructor to create the definition of the **DispersionPlotGroup** domain object.

```
class DispersionPlotGroupDefinition
{
public:
  DispersionPlotGroupDefinition(NormalTrack track,
    LineTrackItem lineTrackItemX);

  void setDepthIntervals(float topDepth, float bottomDepth,
    float stepDepth, Unit unit);
}
```

Arguments passed to the constructor are mandatory to create a group of dispersion plots along a **NormalTrack** of the **Logview**:

3) The **NormalTrack** has been created with an associated **Well**

   See "NormalTrack domain object" section on page 23 for more information on how to create a **NormalTrack** domain object.

4) The **LineTrackItem** instance passed to the constructor is used to set the reference. The **LineTrackItem** is already plotted in a **NormalTrack** of the same **Logview** and belong to the **Well** that is used to create the **NormalTrack** passed as first argument of the **DispersionPlotGroupDefinintion** constructor.

   See "LineTrackItem Domain Object" section on page 35 for more information on how to create a **LineTrackItem** domain object.

A **NormalTrack** can't contain more than one **DispersionPlotGroup** domain object at the time. This can be checked using the **findDispersionPlotGroup** function of the **NormalTrack** class.

```
class NormalTrack : public Track
{
public:
  DispersionPlotGroup findDispersionPlotGroup() const;
  …
};
```

Once the **DispersionPlotGroupDefinition** object has been instantiated through its constructor you can create more than a dispersion plot in the group using the **setDepthIntervals** function. This function allows you to set the top and bottom depth interval within dispersion plots ared displayed in the group and set the step depth to control dispersion plots display along the interval.

The **DispersionPlotGroupDefinition** object is passed to the **create** static function of the **DispersionPlotGroup** class in order to create the group of dispersion plots into the **NormalTrack**.

```
class DispersionPlotGroup : public PlotGroup
{
public:
  static DispersionPlotGroup create(const
    DispersionPlotGroupDefinition &definition);
```

SLB-Private

```
  LineTrackItem lineTrackItem() const;
  void addFrequencyArray(const Variable &associatedWaveform,
    const Variable &frequencyArray);
  void removeFrequencyArray(const Variable &associatedWaveform,
    const Variable &frequencyArray);
  void addSlownessArray(const Variable &associatedWaveform,
    const Variable &slownessArray, const Variable &energyArray,
    const int order, const QColor &color, const PointType
    markersType);
  void removeSlownessArray(const Variable &associatedWaveform,
    const Variable &slownessArray);
  void addVertical2Array(const Variable &associatedWaveform,
    const Variable &y2Array, const QColor &color, const PointType
    markersType, const bool pointLinked, const bool
    pointVisible);
  void removeVertical2Array(const Variable &associatedWaveform,
    const Variable &y2Array);
  void addSlownessLine(const Variable &slownessLine, const QColor
    &color);
  void removeSlownessLine(const Variable &slownessLine);
};
```

The **`PlotGroup`** class allows you to:

- add frequency array and its associated waveform
- remove slowness array over frequency array associated to given waveform from dispersion plot
- add slowness array over frequency array associated to given waveform as a set of point in x-y axes in dispersion plot
- remove slowness array over frequency array associated to given waveform from dispersion plot
- add spectrum or snr array over frequency array associated to given waveform as a set of point or polyline in x-y2 axes in dispersion plot
- remove spectrum or snr array over frequency array associated to given waveform from dispersion plot
- add slowness line as a horizontal flat line in x-y axes in dispersion plot
- remove slowness line from dispersion plot

SLB-Private

Plots single well

The cross-plot interface has a central viewing area in which the plot is displayed. The box on the left contains the variables, filters, and charts displayed over the plot.

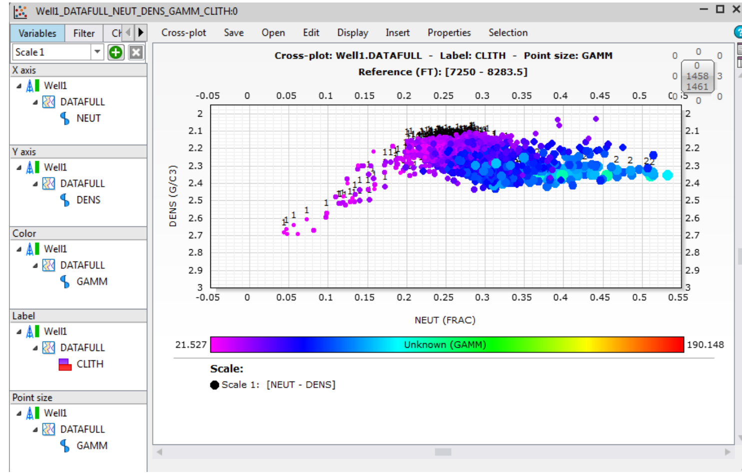


Figure 54 Cross-plot single well

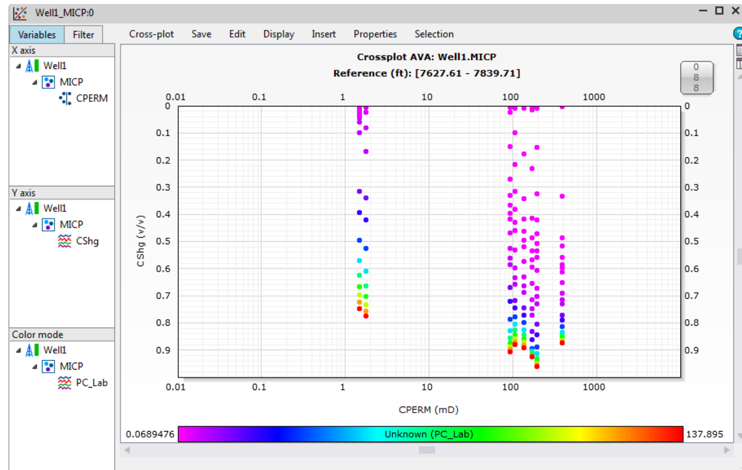


Figure 55 Cross-plot AVA single well

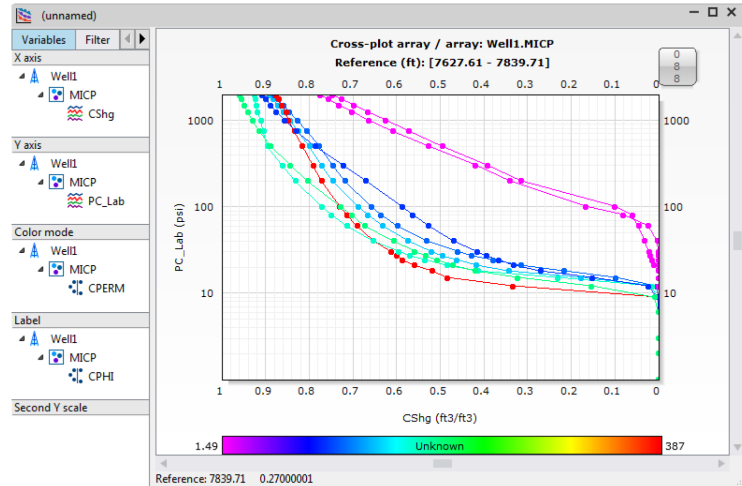


Figure 56 Cross-plot Array single well

Variables tab is where you place curves to be displayed by specifying X axis, Y axis, Color code (Label and Point Size for cross-plot single well).

Filter tab allows you to refine the display using a different variable. Most commonly, this is a qualitative variable. For example, you might wish to display data points filtering on facies, fluid code, or any classification group.

Charts tab adds predefined charts to the data. Charts added over the data points allow you to identify the lithologies and the presence of hydrocarbons.

Through Ocean domain objects you can add a cross-plot view to the Techlog workspace and setting programmatically variables on all dimensions of the variables tab and on the filter tab too. Charts tab is not available with Ocean but you can design your own charts using custom graphics.

Cross-plots single well expsosed with ocean are:

- cross-plot single well modeled through `CrossPlot` domain object
  - allows you to compare multiple measurements made at a single reference over a 2D plot
  - a single well cross-plot can handle multiple scales and multiple variables
- cross-plot Array Variable Array (AVA) modeled through `CrossPlotArrayVariableArray` domain object
  - the cross-plot AVA displays an array variable versus a simple curve
  - another array variable filters the displayed values
- cross-plot Array modeled through `CrossPlotArrayArray` domain object
  - the cross-plot array used to cross-plot an array variable versus another array variable
  - the two arrays plotted on X and Y axes need to have the same column size

The Histogram plot is a graphical display of tabulated frequencies to visualize the data distribution. The Histogram is compatible with scalar (variables) and array data.

Several variables and arrays are plotted together if their units are compatible.

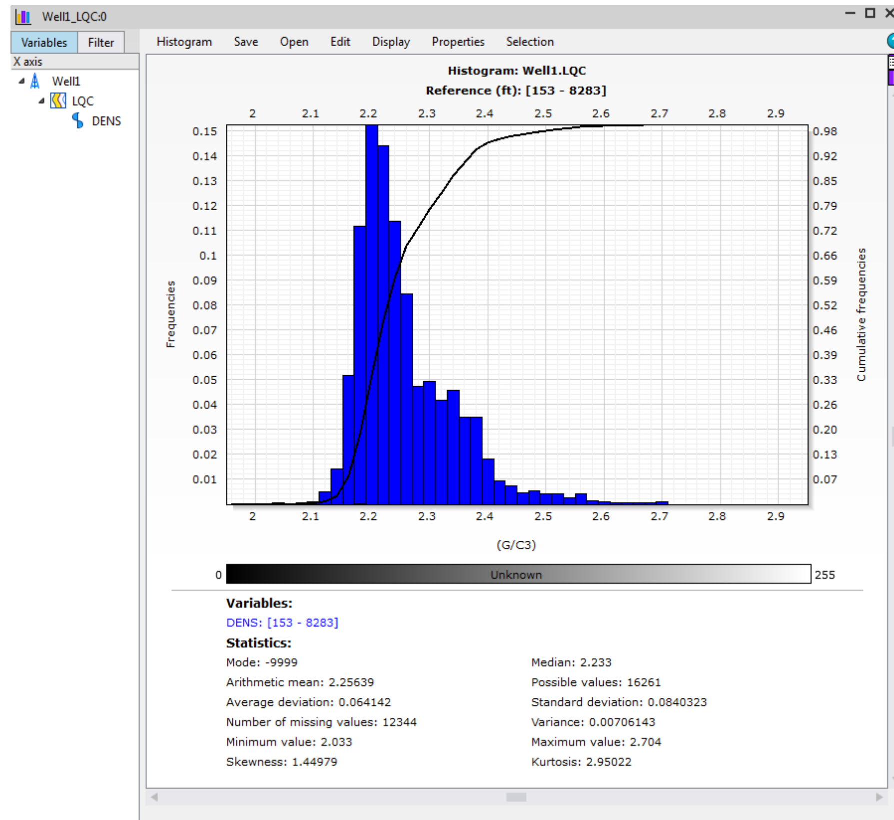


Figure 57 Histogram single well

Histogram plot is modeled in Ocean through the `Histogram` domain object.

The Line-plot is a crossplot for which one of the two plotted variables on X and Y axes must be strictly monotonic (increasing or decreasing):

- X axis: Measured depth (monotically increasing)
- Y axis: Density
- Color: Gamma Ray

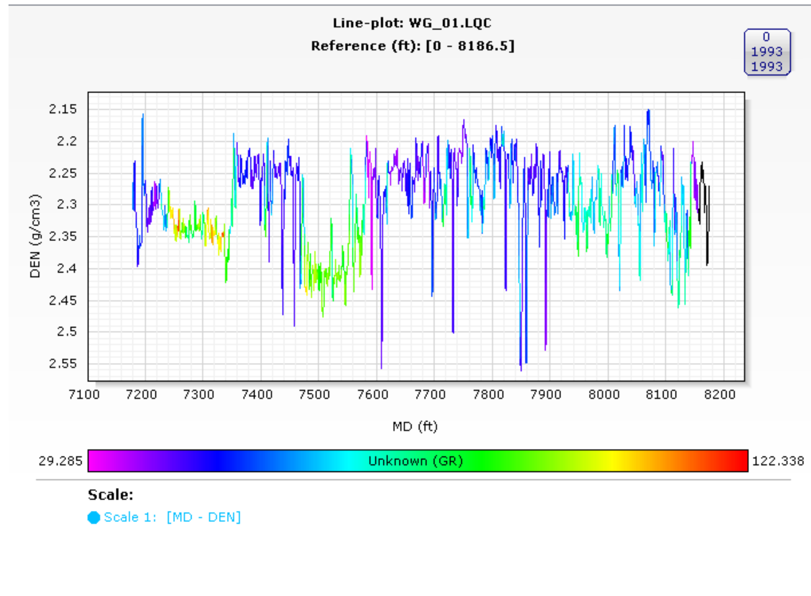


Figure 58 Line-plot

CrossPlot and PlotScale Domain Objects

The **CrossPlot** class inherits from the **Plot** class and does not support name. A **CrossPlot** object is instancied through **create** static method of the class with the parent **Workspace** object as argument. The uniqueness of a **CrossPlot** object is handled by the system. The only way to retrieve a **CrossPlot** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

A **CrossPlot** can also be added to a container plot (matrix-plot) using the dedicated **create** static method.

See the "ContainerPlot Domain Object" section on page 189 for more information on how to create a cross-plot into a container plot.

```
class CrossPlot : public Plot
{
public:
  static CrossPlot create(Workspace workspace);

  static CrossPlot create(ContainerPlotPosition
    &containerPlotPosition);

  PlotScale defaultPlotScale() const;
  DomainObjectCollection<PlotScale> plotScales() const;
  PlotScale findPlotSscale(const QString &plotScaleName) const;
  PlotScale getPlotScale(const QString &plotScaleName) const;
  …
  const Variable findVariableFilter() const;
  void setVariableFilter(const Variable &variable);
  QStringList variableFilterValues();
  void setVariableFilterValues(const QStringList &values);

  const GlobalZonation findSelectedZonation() const;
  const QList<GlobalZone> selectedZones() const;
```

```
  void setSelectedZones(const GlobalZonation
  &selectedZonation, const QList<GlobalZone> &selectedZones);

  const PlotScaleType xScaleType() const;
  void setXScaleType(const PlotScaleType plotScaleType);
  const PlotScaleType yScaleType() const;
  void setYScaleType(const PlotScaleType plotScaleType);
  QString paletteName() const;
  void setPalette(const QString &paletteName, StorageLevel
  level);

  const QString title() const;
  void setTitle(const QString &title);
  const QString subtitle() const;
  void setSubtitle(const QString &title);
  bool isSideBoxVisible() const;
  void setSideBoxVisible(bool isSideBoxVisible);

  DomainObjectCollection<Regression> regressions() const;

  void addEquation(const QString &function, const QString &name,
   const QColor &color);
  …

};
```

CrossPlot domain object is created with a default scale which can be retrieved through the **defaultPlotScale** public method. This method returns a **PlotScale** domain object. Additional scales can be added to the **CrossPlot** but by default, only one scale is used and any additional scale is independent from the others. The **plotScales** method allows you to retrieve the domain object collection of **PlotScale** that belongs to the **CrossPlot** or a PlotScale can be retrieved by its name through **findPlotScale** or **getPlotScale** functions.

Properties of **CrossPlot** class allow you to access cross-plot window display options and to do the following:

- get and set filter variable to the filter tab of the plot and select some values
- **setSelectedZones** gives the ability to select some **GlobaZone** in a **GlobalZonation** and display its in the **CrossPlot**
- get and set X and Y axes scale type through **PlotScaleType** enum class values (linear or logarithmic)
  - applied to all the **scales** in the **CrossPlot**
- set a color palette value to use in the **CrossPlot** at a storage level (Techlog, Company, Project, User, Plug-in folder)
  - setting the palette will overwrite the **Variable** color axis.
- get and set a **title** and **subtitle** to the **CrossPlot**
- hide / show the variable side box
- get the **DomainObjectCollection** of linear regressions created into the plot (see the "Regression" section on page 137)

- ability to add an equation to the plot where Y is a function of X with the **addEquation** function (X values are first variable values plot on X axis of the **defaultScale**)

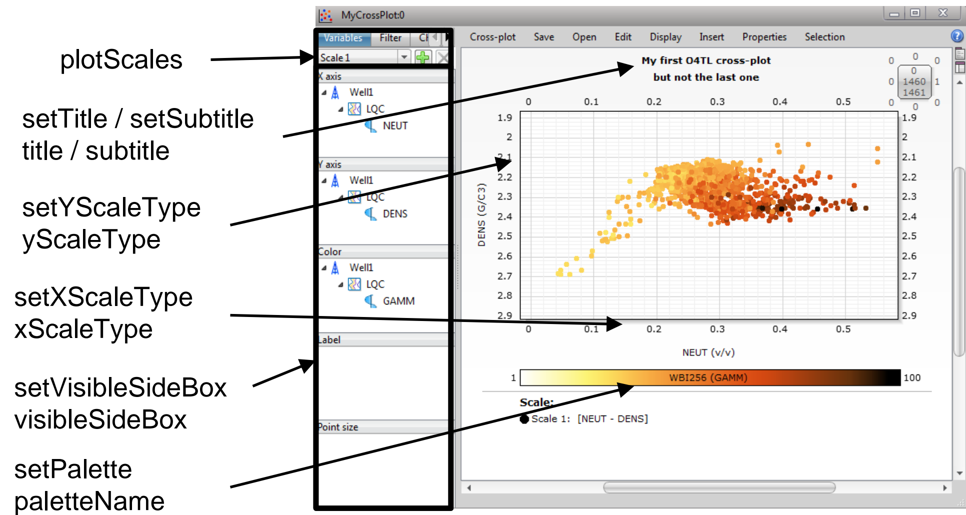


Figure 59 CrossPlot display options

Variables are set on the five dimensions of the plot through the **PlotScale** domain object.

```
class PlotScale : public DomainObject
{
public:
  static PlotScale create(const QString &name, CrossPlot
  crossplot);
  bool isDefaultScale() const;

  void addXAxisVariable(const Variable &variable);
  void trySetXAxisVariables(const QList<Variable> &variables);
  bool canAddXAxisVariable(const Variable &variable) const;
  void clearXAxisVariables();

  void addYAxisVariable(const Variable &variable);
  void trySetYAxisVariables(const QList<Variable> &variables);
  bool canAddYAxisVariable(const Variable &variable) const;
  void clearYAxisVariables();

  void setColor(const Variable &value);
  void setLabel(const Variable &value);
  void setPointSize(const Variable &value);

  DomainObjectCollection<Variable> xAxisVariables() const;
  DomainObjectCollection<Variable> yAxisVariables() const;
  const Variable findColor() const;
  const Variable findLabel() const;
  const Variable findPointSize() const;
  …
};
```

An additional scale can be added to the **CrossPlot** through **create** static method. The scale name is used as a unique identifier within a **CrossPlot**, and therefore plug-ins should always check for the existence of a **PlotScale** with a particular name in the same **CrossPlot** before creating a new one.

Note: CrossPlot default scale cannot be erased. Before to remove a **PlotScale** domain object check if it is the **CrossPlot** default scale through **isDefaultScale** public method. Removing the default scale will throw an exception.

Several variables can be added to a scale axis (X and Y axis). However Techlog can reject a variable added to X or Y axis for the reasons listed below:

- variable unit isn't compatible with the unit of the variables already added to the axis
- variable isn't from the same well than variables already added to the axis
- there is already more than one variable already added to the other axis

The **canAdd** and **trySet** functions allow to check if the variable respect those rules before to add it to a scale axis.

Varaibles added to X and Y axes can be retrieved respectively from the **xAxisVariables** and **yAxisVariables** domain object collections.

Find pattern allows for color, label and point size dimensions to retrieve the **Variable** instance. The method return a null object if no **Variable** is set to the dimension.

Example cross-plotting variable on X, Y and color axes:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Dataset dataset =
DomainObject::get(datasetDroid).cast<Dataset>();

Variable neutron = dataset.variables().get("NEUT");
Variable density = dataset.variables().get("DENS");
Variable gammaRay = dataset.variables().get("GAMM");

// Cross-plotting on three dimensions X=Neut, Y=Dens, Color=GR
CrossPlot crossplot = CrossPlot::create(workspace);
crossplot.setWindowTitle("My Cross-Plot");
PlotScale scale = crossplot.defaultPlotScale();
scale.addXAxisVariable(neutron);
scale.addYAxisVariable(density);
scale.setColor(gammaRay);

lock.release();
```

```
class PlotScale : public DomainObject
{
public:
  …

  void setMarkerType(const PointType PointType);
  PointType markerType() const;
  void setMarkerSize(const int PointType);
  int markerSize() const;
  void setVariableColors(const QList<QColor> variableColors);
  QList<QColor> variableColors() const;

  void setAutoAdjustXY(const bool variable);
  bool isAutoAdjustXY() const;

  double horizontalUserLowerLimit() const;
  double horizontalUserUpperLimit() const;
  double verticalUserLowerLimit() const;
  double verticalUserUpperLimit() const;
  double horizontalFamilyLowerLimit() const;
  double horizontalFamilyUpperLimit() const;
  double verticalFamilyLowerLimit() const;
  double verticalFamilyUpperLimit() const;
  double horizontalVariableLowerLimit() const;
  double horizontalVariableUpperLimit() const;
  double verticalVariableLowerLimit() const;
  double verticalVariableUpperLimit() const;
  void setHorizontalUserLimits(double horizontalUserLowerLimit,
  double horizontalUserUpperLimit,const Unit &unit);
  void setVerticalUserLimits(double verticalUserLowerLimit,
  double verticalUserUpperLimit,const Unit &unit);

  void setVerticalLimitType(AxisLimitType type);
  void setHorizontalLimitType(AxisLimitType type);

  Unit horizontalUnit() const;
  Unit verticalUnit() const;

  QString axisYLegend() const;
  void setAxisYLegend(const QString &legend);
  QString axisXLegend() const;
  void setAxisXLegend(const QString &legend);
};
```

The properties of **`PlotScale`** class allow you to access scale display options and to do the following:

- get and set point size, type and color plotted in the scale
- get variable and family upper and lower limits for X and Y axes
- get and set user upper and lower limits for X and Y axes passing the considered unit for limit values
    - The **`Unit`** tells the plot which unit has to be considered regarding limits values passed to the function. Those values are converted from the Unit to

the plot axis unit and converted values are set to limits. See “Plot axes limits and display parameters” in *Ocean for Techlog Developer Guide – Basics*.

- change limit types to variable, family or user
- get display unit of variables set to horizontal and vertical axes of the scale (see “Display unit” in *Ocean for Techlog Developer Guide – Basics*)
- get and set X and Y axes legends

The following example shows how to create a new **PlotScale** and modify point size, type and color and adjust X and Y limits automatically:

```
Lock lock2 = LOCK_CREATE_AND_ACQUIRE_ALL(lock2);

Variable dens = dataset.variables().get("DENS");
Variable neut = dataset.variables().get("NEUT");
Variable pore_eo = dataset.variables().get("PORE_EO");

PlotScale scale2 = crossplot.findPlotScale("scale 2");
if (scale2.isNull())
  scale2 = PlotScale::create("scale 2", crossplot);

if (scale2.canAddXAxisVariable(pore_eo))
  scale2.addXAxisVariable(pore_eo);
if (scale2.canAddXAxisVariable(neut))
  scale2.addXAxisVariable(neut);
if (scale2.canAddYAxisVariable(dens))
  scale2.addYAxisVariable(dens);

// Point size, type and color
scale2.setMarkerSize(8);
scale2.setMarkerType(PointTypeDiamond);
// set 2 colors which is the size of the 2 combinations of variables
plotted on the scale
scale2.setVariableColors(QList<QColor>() << Qt::red <<
Qt::blue);

// X and Y adjusted automatically
scale2.setAutoAdjustXY(true);

lock2.release();
```

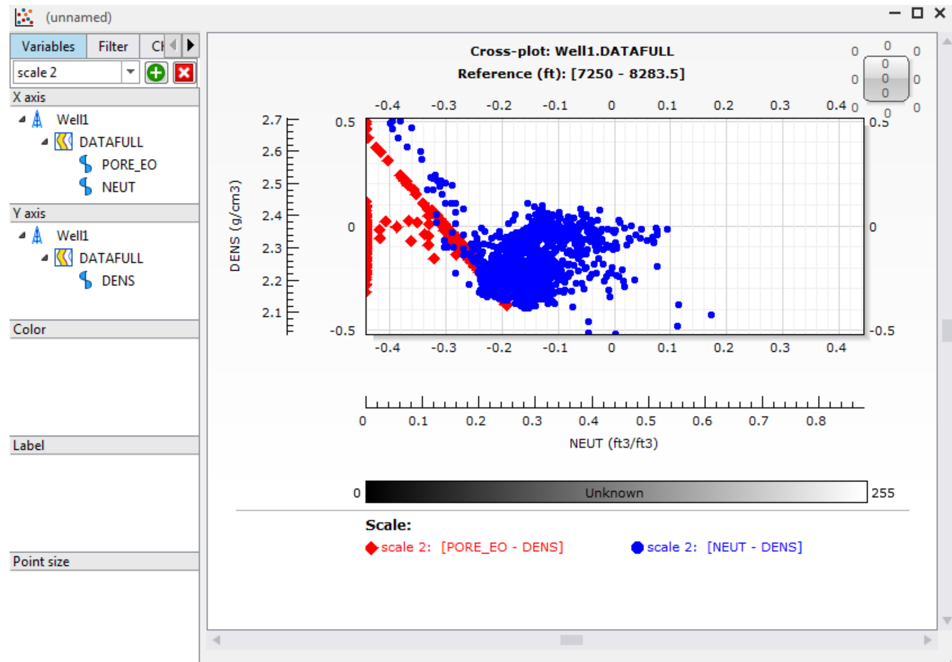


Figure 60 PlotScale display options

In the **CrossPlot** class a create static method is available to instantiate a **CrossPlot** object passing the parent **Workspace** and a **CrossPlotTemplateDataBinding** instance.

A **CrossPlot** can also be created by template into a container plot (matrix-plot) using the dedicated **create** static method.

See the "ContainerPlot Domain Object" section on page 189 for more information on how to create a **CrossPlot** by template into a container plot.

```
class CrossPlot : public Plot
{
public:
  …
  static CrossPlot create(Workspace workspace, const
   CrossPlotTemplateDataBinding &crossPlotTemplateDataBinding);

  static CrossPlot create(const ContainerPlotPosition
    &containerPlotPosition, const CrossPlotTemplateDataBinding
    &crossPlotTemplateDataBinding) ;
  …
}
```

The **CrossPlotTemplateDataBinding** contains the layout template saved at a storage level and the well or the dataset to apply to this template.

```
class CrossPlotTemplateDataBinding
{
public:
  CrossPlotTemplateDataBinding(const CrossPlotTemplate
  &crossplotTemplate, const Well &well);

  CrossPlotTemplateDataBinding(const CrossPlotTemplate
```

```
  &crossplotTemplate, const Dataset &dataset);

}
```

In Techlog after setting a cross-plot for a well, you can save the cross-plot as a template. As in the following screenshot where a cross-plot that shows the Bulk Density values (Y axis) Neutron Porosity values (X axis), Gamma Ray values (color of the points) and a chart to ease the facies identification is saved as a template.

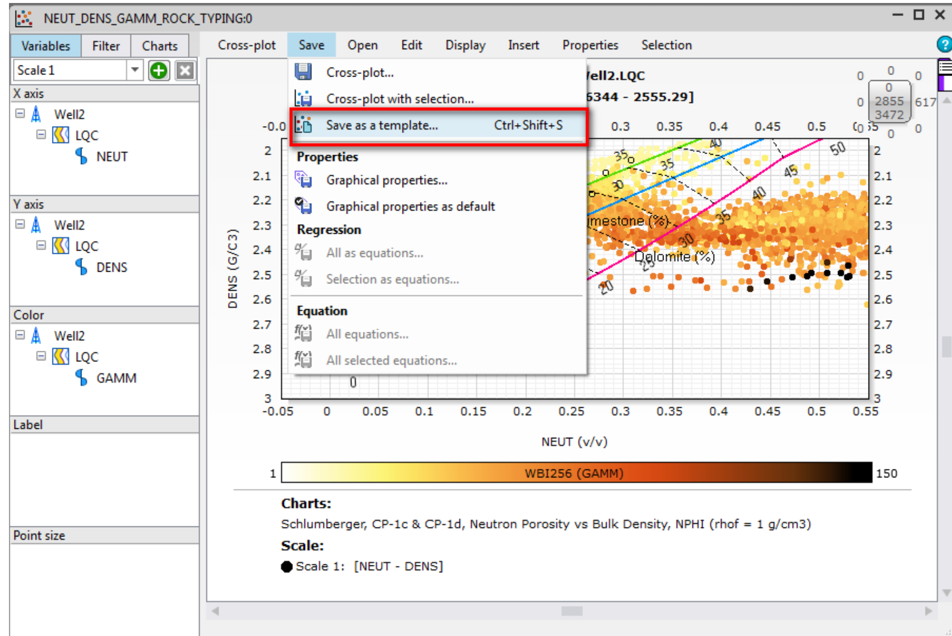


Figure 61 Cross-plot saved as a template

Note: The difference between a cross-plot and a template is that a template saves only the content of a cross-plot, whereas a cross-plot saves specific variables and the complete display.

You specify the template name and at the level where is stored the template and pass the information through the **CrossPlotTemplate** argument of the function.

```
class CrossPlotTemplate
{
public:
  static CrossPlotTemplate get(const StorageLevel level, const
  QString &name);
  static bool exists(const StorageLevel level, const QString
  &name);

  StorageLevel level() const;
  QString name() const;
   …
}
```

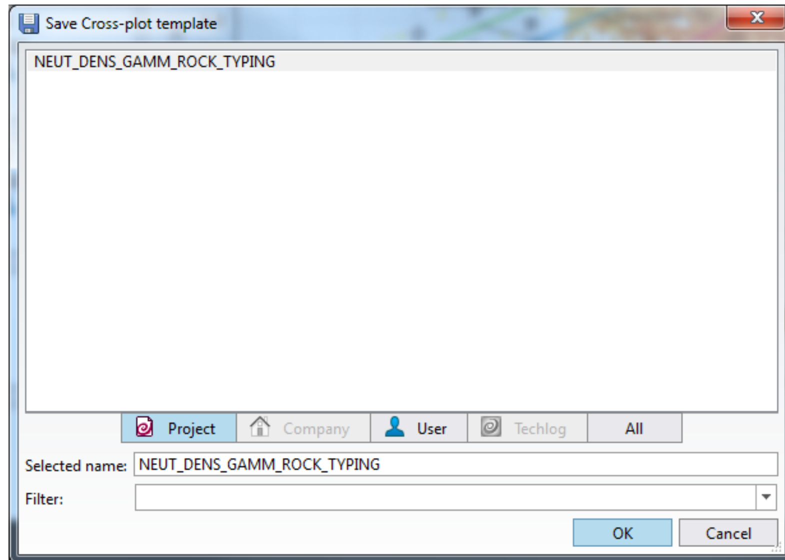


Figure 62 Techlog storage levels

The cross-plot template is saved to the Project browser under Cross-plots. You can retrieve a cross-plot template in the project later, double-click on the template and decide to apply the cross-plot template as a well template or as a dataset template.

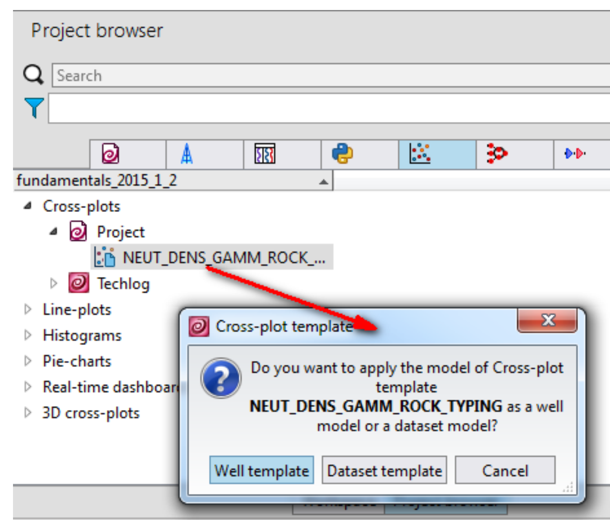


Figure 63 Cross-plot template stored in the project browser at the project level

Ocean introduces an additional level which is the plug-in level and the ability to provide cross-plot templates in the plug-in folder. At the plug-in level the cross-plot template must be stored in a directory named "CrossPlotTemplates" closed to the plug-in dll in the plug-in folder.

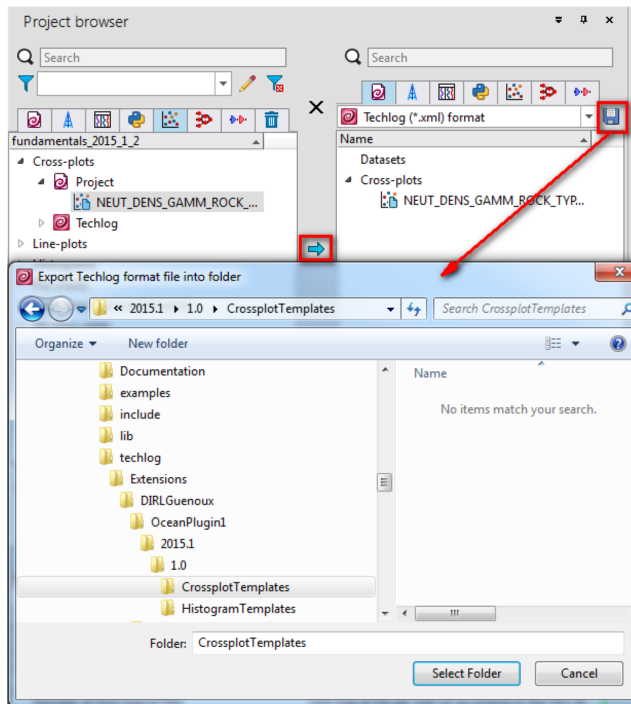


Figure 64 Export template to CrossplotTemplates folder at the plug-in level

The following example shows how to open a cross-plot template stored at the plug-in level.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Project project = Session::current().mainProject();
Well well = project.wells().find("Well1");
if (well.isNull())
{
  lock.release();
  return;
}
Dataset dataset = well.datasets().find("LQC");
if (dataset.isNull())
{
  lock.release();
  return;
}

if (!CrossPlotTemplate::exists(StorageLevelPlugin,
"NEUT_DENS_GAMM_ROCK_TYPING"))
{
  lock.release();
```

```
  return;
}

CrossPlotTemplate crossplotTemplate =
CrossPlotTemplate::get(StorageLevelPlugin,
"NEUT_DENS_GAMM_ROCK_TYPING");

CrossPlotTemplateDataBinding templateDataBinding =
CrossPlotTemplateDataBinding(crossplotTemplate, dataset);

CrossPlot crossplot = CrossPlot::create(workspace,
templateDataBinding);

lock.release();
```

LinePlot Domain Object

The **LinePlot** class inherits from the **Plot** class and does not support name. A **LinePlot** object is instancied through **create** static method of the class with the parent **Workspace** object as argument. The uniqueness of a **LinePlot** object is handled by the system. The only way to retrieve a **LinePlot** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

A **LinePlot** can also be added to a container plot (matrix-plot) using the dedicated **create** static method.

See the “ContainerPlot Domain Object” section on page 189 for more information on how to create a line plot into a container plot.

```
class LinePlot : public Plot
{
public:
  static LinePlot create(Workspace workspace);

  static LinePlot create(Workspace workspace, const
   LinePlotTemplateDataBinding &linePlotTemplateDataBinding);

  PlotScale defaultScale() const;
  DomainObjectCollection<PlotScale> scales() const;

  const GlobalZonation findSelectedZonation() const;
  const QList<GlobalZone> selectedZones() const;
  void setSelectedZones(const GlobalZonation
  &selectedZonation, const QList<GlobalZone> &selectedZones);

  const QString title() const;
  void setTitle(const QString &title);
  const QString subtitle() const;
  void setSubtitle(const QString &title);
  bool isSideBoxVisible() const;
  void setSideBoxVisible(bool isSideBoxVisible);
```

```
  bool isLegendBoxVisible() const;
  void setLegendBoxVisible(bool visible);
  bool isColorScaleVisible() const;
  void setColorScaleVisible(bool visible);

  void setReferenceLimitsEnabled(bool enabled);
  bool referenceLimitsEnabled() const;
  void setReferenceLimits(double top, double bottom);
  double referenceTopLimit() const;
  double referenceBottomLimit() const;

  PlotScaleType xScaleType() const;
  void setXScaleType(const PlotScaleType plotScaleType);
  PlotScaleType yScaleType() const;
  void setYScaleType(const PlotScaleType plotScaleType);

  DomainObjectCollection<Regression> regressions() const;
  QString paletteName() const;
  void setPalette(const QString &paletteName, StorageLevel
  level);
  …

};
```

The **LinePlot** shares with the **CrossPlot** domain object the **PlotScale** domain object in order to plot variables on different scales.

See the "CrossPlot and PlotScale Domain Objects" section on page 93 for more information on how to retrieve the default plot scale or create additional scales to the **LinePlot**.

The properties of **LinePlot** class allow you to access line-plot window display options and to do:

- **setSelectedZones** gives the ability to select some **GlobaZone** in a **GlobalZonation** and display its in the **LinePlot**
- get and set a **title** and **subtitle** to the **CrossPlot**
- hide/show the variable side box
- hide/show the legend box in the bottom section of the line plot
- hide/show the color scale in the bottom section of the line plot
- ability to constraint the display in the plot setting top and bottom dataset reference limits
- get and set the horizontal and vertical axes scale type to logarithmic or linear
- get the **DomainObjectCollection** of linear regressions created into the plot (see the "Regression" section on page 137)
- get and set a palette to the plot

Note: The reference limits are those from the dataset that contains the variable plotted on X axis.

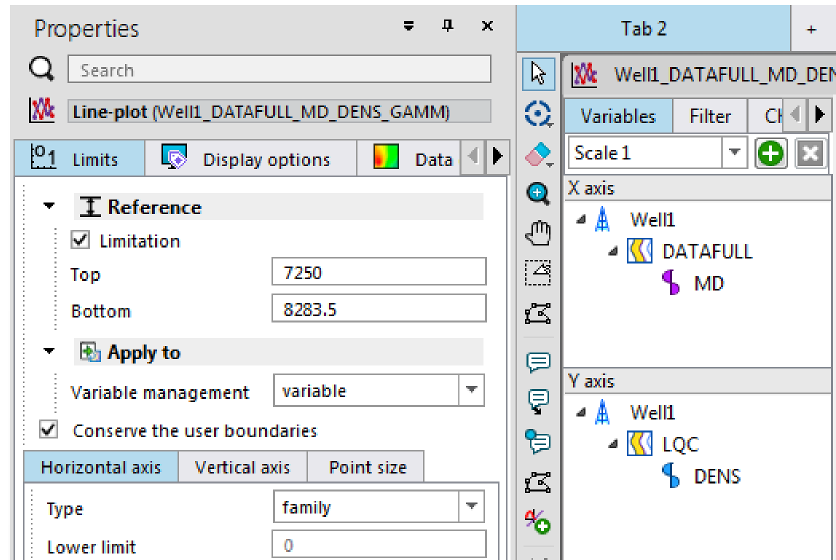


Figure 65 Reference limits are from DATAFULL dataset

As you can see in the sample code below the way to create a **LinePlot** and add variables to the **PlotScale** domain object is the same as for **CrossPlot**. The only difference is one of the variable on X or Y axis has to be monotically increasing as the Measured Depth variable in this example.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Project project = Session::current().mainProject();

Well well = project.findWell("Well1");

Dataset dataset = well.findDataset("DATAFULL");

Variable measuredDepth = dataset.findVariable("MD");
Variable density = dataset.findVariable("DENS");
Variable gammaRay = dataset.findVariable("GAMM");

LinePlot lineplot = LinePlot::create(workspace);
lineplot.setWindowTitle("My Line-Plot");
lineplot.setSideBoxVisible(true);
PlotScale scale = lineplot.defaultScale();
scale.addXAxisVariable(measuredDepth);
scale.addYAxisVariable(density);
scale.setColor(gammaRay);
lock.release();

lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
```

```
Variable neutron = dataset.findVariable("NEUT");

PlotScale scale2 = lineplot.scales().find("scale 2");
if (scale2.isNull())
  scale2 = PlotScale::create("scale 2", lineplot);

scale2.addXAxisVariable(measuredDepth);
scale2.addYAxisVariable(neutron);

// Point size, type and color
scale2.setMarkerSize(8);
scale2.setMarkerType(PointTypeDiamond);
scale2.setVariableColors(QList<QColor>() << Qt::red);

// X and Y adjusted automatically
scale2.setAutoAdjustXY(true);

lock.release();
```

The following screenshot shows the resulting **`LinePlot`**:

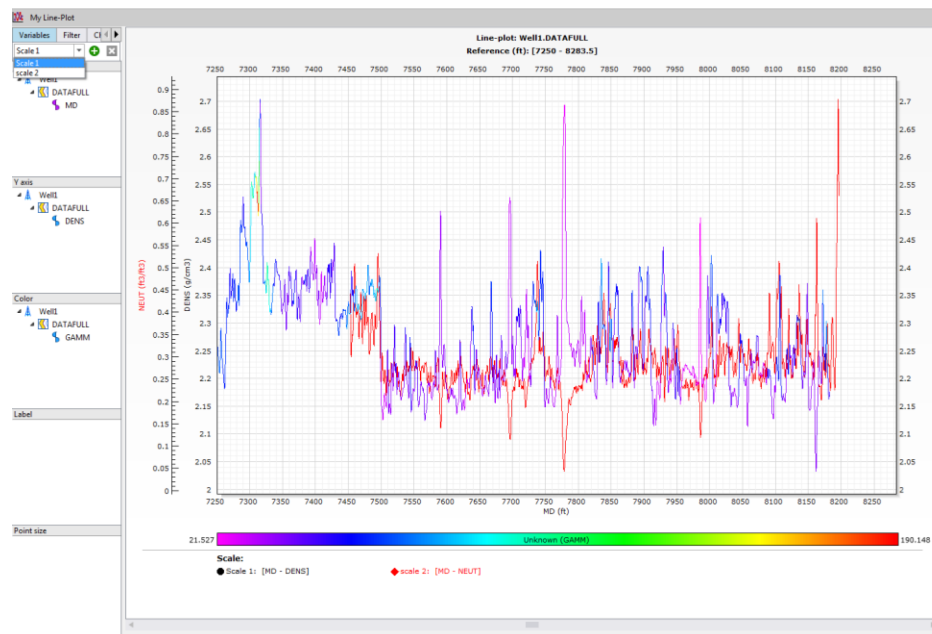


Figure 66 LinePlot with variables plotted on two scales

In the **`LinePlot`** class a create static method is available to instantiate a **`LinePlot`** object passing the parent **`Workspace`** and a **`LinePlotTemplateDataBinding`** instance.

A **`LinePlot`** can also be created by template into a container plot (matrix-plot) using the dedicated **`create`** static method.

See the "ContainerPlot Domain Object" section on page 189 for more information on how to create a **`LinePlot`** by template into a container plot.

SLB-Private

```
class LinePlot : public Plot
{
public:
  …
  static LinePlot create(Workspace workspace, const
   LinePlotTemplateDataBinding &linePlotTemplateDataBinding);

  static LinePlot create(const ContainerPlotPosition
    &containerPlotPosition, const LinePlotTemplateDataBinding
    &linePlotTemplateDataBinding) ;
  …
}
```

The **LinePlotTemplateDataBinding** contains the layout template saved at a storage level and the well or the dataset to apply to this template.

```
class LinePlotTemplateDataBinding
{
public:
  LinePlotTemplateDataBinding(const LinePlotTemplate
  &lineplotTemplate, const Well &well);

  LinePlotTemplateDataBinding(const LinePlotTemplate
  &lineplotTemplate, const Dataset &dataset);
}
```

You will specify the template name and at the level where is stored the template, modeled in Ocean with **LinePlotTemplate** class and enum class **StorageLevel**.

```
class LinePlotTemplate
{
public:
  static LinePlotTemplate get(const StorageLevel level, const
  QString &name);
  static bool exists(const StorageLevel level, const QString
  &name);

  StorageLevel level() const;
  QString name() const;
   …
}
```

Ocean introduces an additional level which is the plug-in level and the ability to provide line-plot templates in the plug-in folder. At the plug-in level the line-plot template must be stored in a directory named "LinePlotTemplates" closed to the plug-in dll in the plug-in folder.

CrossPlotArrayVariableArray Domain Object

The **CrossPlotArrayVariableArray** class inherits from the **Plot** class and does not support name. A **CrossPlotArrayVariableArray** object is instancied through **create** static method of the class with the parent **Workspace** object as argument. The uniqueness of a **CrossPlotArrayVariableArray** object is handled by the system. The only way to retrieve a **CrossPlotArrayVariableArray** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

A **CrossPlotArrayVariableArray** can also be added to a container plot (matrix-plot) using the dedicated **create** static method.

See the “ContainerPlot Domain Object” section on page 189 for more information on how to create a cross-plot AVA into a container plot.

```
class CrossPlotArrayVariableArray : public Plot
{
public:
  static CrossPlotArrayVariableArray create(Workspace
  workspace);

  static CrossPlotArrayVariableArray create(const
  ContainerPlotPosition &containerPlotPosition);

  void setXAxis(const Variable &value);
  void setYAxis(const Variable &value);
  void setColorMode(const Variable &value);

  const Variable findXAxis() const;
  const Variable findYAxis() const;
  const Variable findColorMode() const;

  …

};
```

Once a **CrossPlotArrayVariableArray** domain object is created then you can plot variables on X, Y and color axes.

Note: Throw an exception if not at least an array variable on X or Y axis. The variable on the X axis is a single curve **Variable** (**columnCount** equals to 1), you must define an array for the Y axis (**columnCount** must be greater than 1). To color the points, set an array variable on the **colorMode** axis.

Find pattern enables each dimension to retrieve the **Variable** instance. The method return a null object if no **Variable** is set to the dimension.

The following example shows cross-plotting permeability (single curve) versus core mercury saturation (array) coloring with capillary pressure array:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Dataset dataset =
DomainObject::get(datasetDroid).cast<Dataset>();

Variable cshg = dataset.variables().find("CShg");
Variable cperm = dataset.variables().find("CPERM");
Variable pclab = dataset.variables().find("PC_Lab");

CrossPlotArrayVariableArray crossplotAVA =
CrossPlotArrayVariableArray::create(workspace);
```

```
Droid crossPlotAVADroid = crossplotAVA.droid();

crossplotAVA.setXAxis(cperm);
crossplotAVA.setYAxis(cshg);
crossplotAVA.setColorMode(pclab);

lock.release();
```

The following screenshot shows the display of the **CrossPlotArrayVariableArray**:

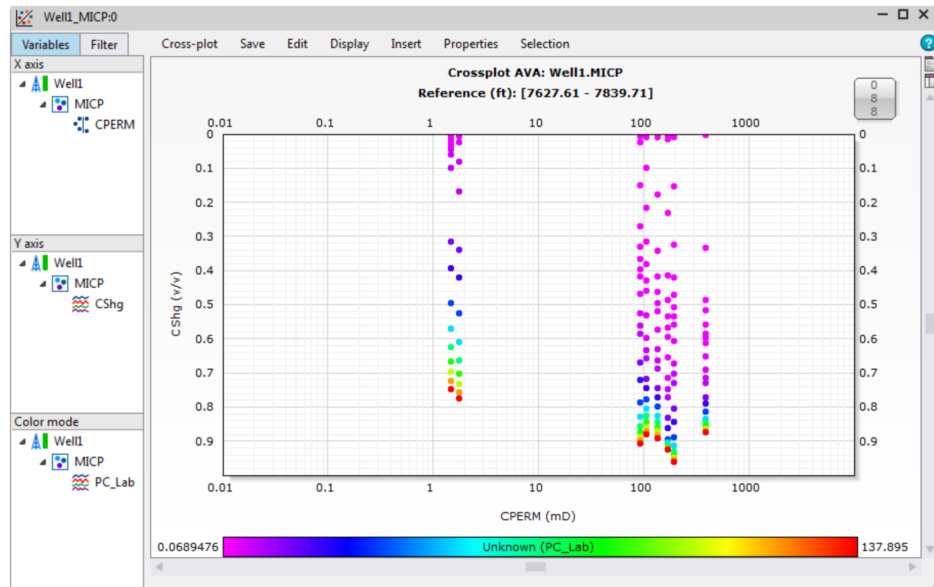


Figure 67 Cross-plot AVA single well

```
class CrossPlotArrayVariableArray : public Plot
{
public:
  …
  const Variable findVariableFilter() const;
  void setVariableFilter(const Variable &variable);

  const GlobalZonation findSelectedZonation() const;
  const QList<GlobalZone> selectedZones() const;
  void setSelectedZones(const GlobalZonation
  &selectedZonation, const QList<GlobalZone> &selectedZones);

  const QString title() const;
  void setTitle(const QString &title);
  const QString subtitle() const;
  void setSubtitle(const QString &title);

  AxisLimitType axisXLimitType() const;
  void setAxisXLimitType(const AxisLimitType &limitType);
  double axisXLowerLimit() const;
  double axisXUpperLimit() const;
```

```
  void setHorizontalUserLimits(const double lowerLimit, const
  double upperLimit, const Unit &unit);

  AxisLimitType axisYLimitType() const;
  void setAxisYLimitType(const AxisLimitType &limitType);
  double axisYLowerLimit() const;
  double axisYUpperLimit() const;
  void setVerticalUserLimits(const double lowerLimit, const
  double upperLimit, const Unit &unit);

  Unit horizontalUnit() const;
  Unit verticalUnit() const;

  QString axisYLegend() const;
  void setAxisYLegend(const QString &legend);
  QString axisXLegend() const;
  void setAxisXLegend(const QString &legend);

  bool isGridDisplayed() const;
  void setGridDisplayed(const bool &display);
  QColor backgroundColor() const;
  void setBackgroundColor(const QColor &color);

  DomainObjectCollection<Regression> regressions() const;

};
```

The properties of **CrossPlotArrayVariableArray** class allow you to access cross-plot AVA display options and to do the following:

- get and set filter variable to the filter tab of the plot
- **setSelectedZones** gives the ability to select some **GlobaZone** in a **GlobalZonation** and display its in the **Logview**
- get and set a **title** and **subtitle** to the **CrossPlotArrayVariableArray**
- get and set X and Y axes limit type through **AxisLimitType** enum class values (user, variable or family)
- get and set X and Y axes limit values
    - you can call the setter methods only if limit type is set to **AxisLimitTypeUser**, otherwise the function will throw
- The **Unit** tells the plot which unit must be considered regarding limits values passed to the function. Those values are converted from the Unit to the plot axis unit and converted values are set to limits. See "Plot axes limits and display parameters" in *Ocean for Techlog Developer Guide – Basics*.
- get display unit of variables set to horizontal and vertical axes of the **CrossplotArrayVariableArray** (see "Display unit" in *Ocean for Techlog Developer Guide – Basics*)
- get and set text legend on X and Y axes
- change the chart area background color and hide or show the grid
- get the **DomainObjectCollection** of linear regressions created into the plot (see the "Regression" section on page 137)

The following example shows how to modify some **CrossPlotArrayVariableArray** display options:

```
Lock lock2 = LOCK_CREATE_AND_ACQUIRE_ALL(lock2);

CrossPlotArrayVariableArray crossPlotAVA =
DomainObject::get(crossPlotAVADroid).cast<CrossPlotArrayVariab
leArray>();

crossPlotAVA.setAxisYLimitType(AxisLimitTypeUser);
crossPlotAVA.setVerticalUserLimits(0, 0.96, cshg.unit());

crossPlotAVA.setGridDisplayed(true);
crossPlotAVA.setBackgroundColor(Qt::lightGray);

crossPlotAVA.setTitle("core mercury saturation curves versus
permeability values");
crossPlotAVA.setSubtitle("colored by capillary pressure array
values");

crossPlotAVA.setAxisXLegend(QString("%1 (%2)").
arg("Permeability").arg(cperm.unit()));
crossPlotAVA.setAxisYLegend(QString("%1 (%2)").
arg("Mercury saturation").arg(cshg.unit()));

lock2.release();
```

CrossPlotArrayArray Domain Object

The **CrossPlotArrayArray** class inherits from the **Plot** class and does not support name. A **CrossPlotArrayArray** object is instancied through **create** static method of the class with the parent **Workspace** object as argument. The uniqueness of a **CrossPlotArrayArray** object is handled by the system. The only way to retrieve a **CrossPlotArrayArray** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

A **CrossPlotArrayArray** can also be added to a container plot (matrix-plot) using the dedicated create static method.

See the "ContainerPlot Domain Object" section on page 189 for more information on how to create a cross-plot array into a container plot.

```
class CrossPlotArrayArray : public Plot
{
public:
  static CrossPlotArrayArray create(Workspace workspace);

  static CrossPlotArrayArray create(const
  ContainerPlotPosition &containerPlotPosition);

  Variable findXAxis() const;
```

```
  void setXAxis(Variable variable);
  ReturnValue<bool> trySetXAxis(Variable variable);

  Variable findYAxis() const;
  void setYAxis(Variable variable);
  ReturnValue<bool> trySetYAxis(Variable variable);

  Variable findColor() const;
  void setColor(Variable variable);
  ReturnValue<bool> trySetColor(Variable variable);

  Variable findLabel() const;
  void setLabel(Variable variable);
  ReturnValue<bool> trySetLabel(Variable variable);

  …

};
```

Once a **`CrossPlotArrayArray`** domain object is created then you can plot variables on X, Y, color and label dimensions.

X and Y axes are expecting variables with the same columns size, however color and label dimensions expect single curve variables.

The "try" functions allow to check if the variable can be "set" to each dimension. Setting an invalid variable to the dimension throws a plug-in exception.

Note: In the Crossplot array vs array, the label is not displayed in the area but in the status bar. The status bar in the Crossplot array vs array is located under the scrollbar at the bottom of the plot. If you want to see the label value associated to a row of the plotted array, move the mouse above the line/points.

The following example shows cross-plotting core mercury saturation (array) versus capillary pressure array coloring with permeability (single curve) and labelling with core porosity (single curve):

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Well well = Session::current().mainProject().findWell("Well1");
if (well.isNull())
{
  lock.release();
  return;
}

Dataset dataset = well.findDataset("MICP");
if (dataset.isNull())
{
  lock.release();
  return;
}
```

```
Variable cshg = dataset.getVariable("CShg");
Variable pclab = dataset.getVariable("PC_Lab");
Variable cperm = dataset.getVariable("CPERM");
Variable cphi = dataset.getVariable("CPHI");

Workspace workspace = Session::current().currentWorkspace();

CrossPlotArrayArray crossplotArray =
CrossPlotArrayArray::create(workspace);

if (crossplotArray.trySetXAxis(cshg))
  crossplotArray.setXAxis(cshg);

if (crossplotArray.trySetYAxis(pclab))
  crossplotArray.setYAxis(pclab);

if (crossplotArray.trySetColor(cperm))
  crossplotArray.setColor(cperm);

if (crossplotArray.trySetLabel(cphi))
  crossplotArray.setLabel(cphi);

crossplotArray.setSideBoxVisible(true);
crossplotArray.setVerticalLimitType(AxisLimitTypeUser);
crossplotArray.setVerticalUserLimits(pclab.minimumValue(),
pclab.maximumValue(), pclab.unit());

lock.release();
```

In the **`CrossPlotArrayArray`** you have the ability to set an array **`Variable`** to a Y2 axis that will display the variable values as graduations on the second Y axis of the plot.

You can also set variables to X, Y and Color dimensions of a second scale of the plot following those rules that can be checked through the corresponding "try" functions:

- variable on the X axis of the second scale has a compatible display unit with the variable set to the X axis of the first scale
- variable on the Y axis of the second scale has a compatible display unit with the variable set to the Y axis of the first scale
- variable on the color axis of the second scale has a compatible display unit with the variable set to the color axis of the first scale
- variable on X and Y axes of the second scale are array variables with the same column size
- variable on the color axis of the second scale is a single curve variable

```
class CrossPlotArrayArray : public Plot
{
public:
  Variable findY2Axis() const;
```

```
  void setY2Axis(Variable variable);
  ReturnValue<bool> trySetY2Axis(Variable variable);

  Variable findXAxisSecondScale() const;
  void setXAxisSecondScale(Variable variable);
  ReturnValue<bool> trySetXAxisSecondScale(Variable variable);

  Variable findYAxisSecondScale() const;
  void setYAxisSecondScale(Variable variable);
  ReturnValue<bool> trySetYAxisSecondScale(Variable variable);

  Variable findYAxisSecondScale() const;
  void setYAxisSecondScale(Variable variable);
  ReturnValue<bool> trySetYAxisSecondScale(Variable variable);

  …
};
```

The default display options can be changed through API's available below.

```
class CrossPlotArrayArray : public Plot
{
public:
  QString title() const;
  void setTitle(const QString &title);
  QString subtitle() const;
  void setSubtitle(const QString &title);

  bool isSideBoxVisible() const;
  void setSideBoxVisible(bool isSideBoxVisible);

  GlobalZonation findSelectedZonation() const;
  QList<GlobalZone> selectedZones() const;
  void setSelectedZones(const GlobalZonation
  &selectedZonation, const QList<GlobalZone> &selectedZones);

  const Variable findVariableFilter() const;
  void setVariableFilter(Variable variable);
  QStringList variableFilterValues() const;
  void setVariableFilterValues(const QStringList &values);

  ColorType colorType() const;
  void setColorType(const ColorType colorType);

  QString axisYLegend() const;
  void setAxisYLegend(const QString &legend);
  QString axisXLegend() const;
  void setAxisXLegend(const QString &legend);

  void setHorizontalUserLimits(double horizontalUserLowerLimit,
    double horizontalUserUpperLimit, const Unit &unit);
  double horizontalUserLowerLimit() const;
  double horizontalUserUpperLimit() const;
  Unit horizontalUnit() const;
```

SLB-Private

```
  void setVerticalUserLimits(double verticalUserLowerLimit,
    double verticalUserUpperLimit, const Unit &unit);
  double verticalUserLowerLimit() const;
  double verticalUserUpperLimit() const;
  Unit verticalUnit() const;

  void setHorizontalLimitType(AxisLimitType type);
  AxisLimitType horizontalLimitType() const;
  void setVerticalLimitType(AxisLimitType type);
  AxisLimitType verticalLimitType() const;

  DomainObjectCollection<Regression> regressions() const;
  …
};
```

The properties of **CrossPlotArrayArray** class allow you:

- get and set a **title** and **subtitle** to the **CrossPlotArrayArray**
- hide/show the variable side box
- **setSelectedZones** gives the ability to select some **GlobaZone** in a **GlobalZonation** and display its in the **Logview**
- get and set filter variable to the filter tab of the plot
- get and set the color type to color the scale by palette, by well or by zone

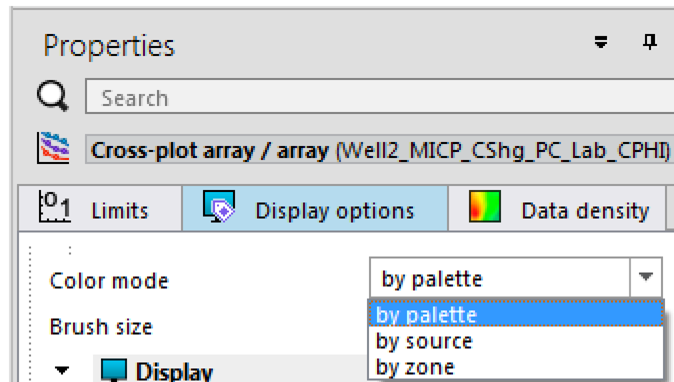


Figure 68 Scale color type

- get and set X and Y axes legends
- get and set user upper and lower limits for X and Y axes passing the considered unit for limit values
  - The **Unit** tells the plot which unit has to be considered regarding limits values passed to the function. Those values are converted from the Unit to the plot axis unit and converted values are set to limits. See "Plot axes limits and display parameters" in *Ocean for Techlog Developer Guide – Basics*.
- change X and Y axes limit types to variable, family or user
- get the **DomainObjectCollection** of linear regressions created into the plot (see the "Regression" section on page 137)

The following screenshot shows the display of the **CrossPlotArrayArray**:

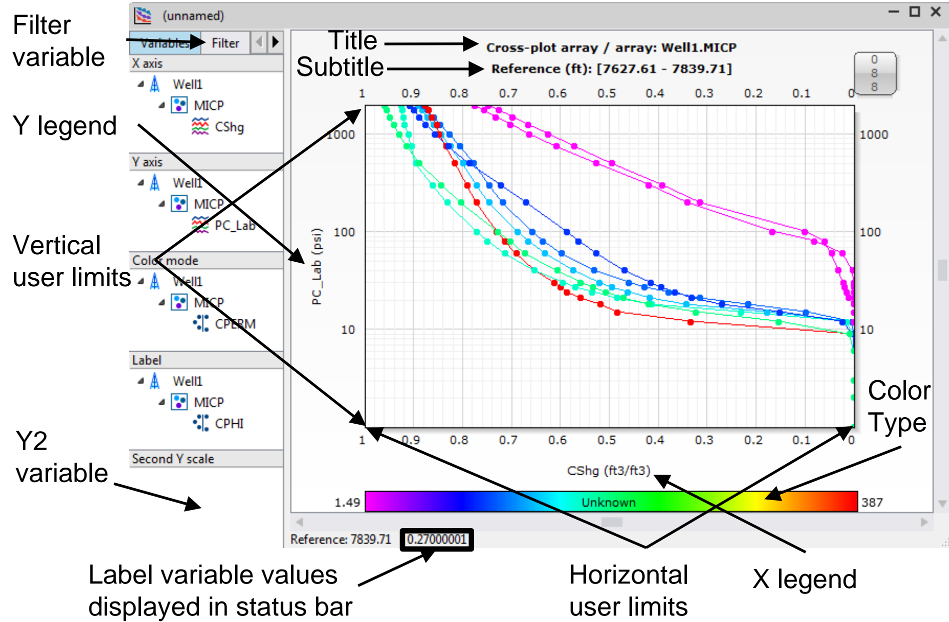


Figure 69 Cross-plot array single well

Histogram Domain Object

The **Histogram** class inherits from the **Plot** class and does not support name. A **Histogram** object is instancied through **create** static method of the class with the parent **Workspace** object as argument. The uniqueness of a **Histogram** object is handled by the system. The only way to retrieve a **Histogram** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

A **Histogram** can also be added to a container plot (matrix-plot) using the dedicated **create** static method.

See the "ContainerPlot Domain Object" section on page 189 for more information on how to create a histogram into a container plot.

```
class Histogram : public Plot
{
public:
  static Histogram create(Workspace workspace);

  static Histogram create(const ContainerPlotPosition
  &containerPlotPosition);

  DomainObjectCollection<Variable> variables() const;
  void addVariable(const Variable &variable);
  bool canAddVariable(const Variable &variable);
  void clearVariables();
  …
};
```

Once a **Histogram** domain object is created, you can add variables through the **addVariable** method.

SLB-Private

Note: Throw an exception if the variable added has not a compatible unit with the **`variables`** present in the **`Histogram`** or comes from a non compatible **`Dataset`**. To check if the variable can be added to the **`Histogram`** you can use the **`canAddVariable`** method.

The following example shows where two gamma ray variables with respectively "gAPI" and "api" units are added to the Histogram.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Workspace workspace = Session::current().currentWorkspace();
Dataset dataset =
DomainObject::get(datasetDroid).cast<Dataset>();

Variable gr = dataset.variables().get("GR");
Variable gr_2 = dataset.variables().get("GR_2");

Histogram histogram = Histogram::create(workspace);
if (histogram.canAddVariable(gr))
  histogram.addVariable(gr);
if (histogram.canAddVariable(gr_2))
  histogram.addVariable(gr_2);

lock.release();
```

The following screenshot shows gamma ray variables values as cumulated with a cumulated curve and type set as "blocked closed".

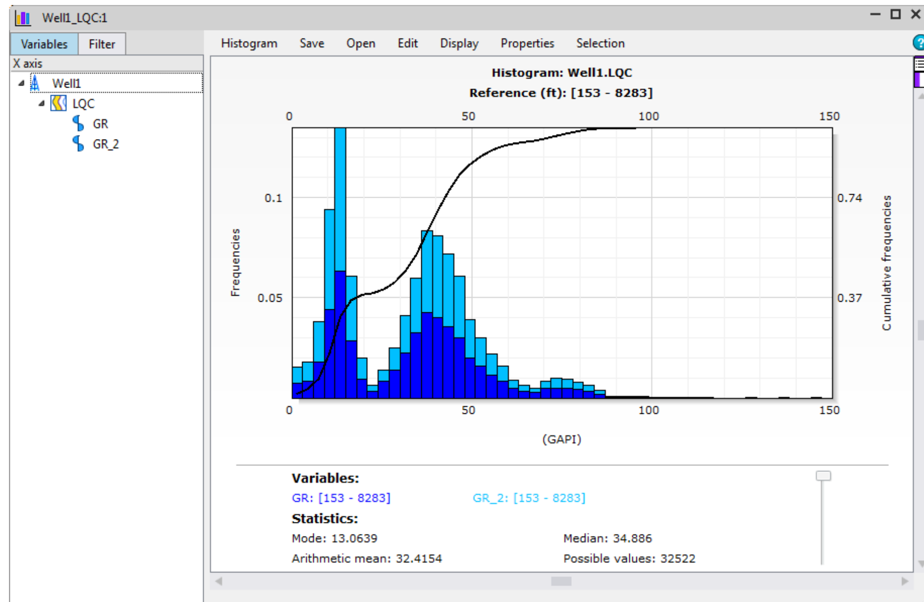


Figure 70 Histogram single well

The default display options can be changed through API's available below.

```
class Histogram : public Plot
{
public:
  QString title() const;
```

```
void setTitle(const QString &title);
QString subtitle() const;
void setSubtitle(const QString &title);

bool isSideBoxVisible() const;
void setSideBoxVisible(bool isSideBoxVisible);

QList<QColor> variableColors() const;
void setVariableColors(const QList<QColor> variableColors);

GlobalZonation findSelectedZonation() const;
QList<GlobalZone> selectedZones() const;
void setSelectedZones(const GlobalZonation
&selectedZonation, const QList<GlobalZone> &selectedZones);

Variable findVariableFilter() const;
void setVariableFilter(Variable variable);
QStringList variableFilterValues();
void setVariableFilterValues(const QStringList &values);

PlotScaleType xScaleType() const;
void setXScaleType(const PlotScaleType plotScaleType);

bool horizontalAxisInverted() const;
void setHorizontalAxisInverted(bool horizontalAxisInverted);

QString paletteName() const;
void setPalette(const QString &paletteName, StorageLevel
level);

bool isReferenceLimitationEnabled() const;
void setReferenceLimitationEnabled(const bool enabled);
void setReferenceInterval(const double topLimitation, const
double bottomLimitation);
double topReferenceLimit() const;
double bottomReferenceLimit() const;

double horizontalUserLowerLimit() const;
double horizontalUserUpperLimit() const;
double verticalUserLowerLimit() const;
double verticalUserUpperLimit() const;
void setHorizontalUserLimits(const double
horizontalUserLowerLimit, const double
horizontalUserUpperLimit, const Unit &unit);
void setVerticalUserLimits(const double
verticalUserLowerLimit, const double
verticalUserUpperLimit);

Unit horizontalUnit() const;

HistogramCurvesDisplay curvesDisplay();
void setCurvesDisplay(const HistogramCurvesDisplay
curvesDisplay);
```

```
  bool isNormalized();
  void setNormalized(const bool isNormalized);
  bool isCumulated();
  void setCumulated(const bool isCumulated);
  bool isFilled();
  void setFilled(const bool fill);

  uint numberOfBins();
  void setNumberOfBins(const uint numberOfBins);

  HistogramStyle style();
  void setStyle(const HistogramStyle style);

  void setXAxisGraduations(const double xAxisGraduations);
  double xAxisGraduations()
  void setYAxisGraduations(const double yAxisGraduations);
  double yAxisGraduations()
  …
};
```

The properties of **Histogram** class allow you to access histogram limits and display options property editor tabs and to do the following:

- get and set a **title** and **subtitle** to the **Histogram**
- hide/show the variable side box
- get and set colors for each variables displayed in the **Histogram**
- **setSelectedZones** gives the ability to select some **GlobaZone** in a **GlobalZonation** and display its in the **Logview**
- get and set filter variable to the filter tab of the plot and select some values
- get and set X and Y axes scale type through **PlotScaleType** enum class values (linear or logarithmic)invert horizontal axis
- set a color palette value to use in the **Histogram** at a storage level (Techlog, Company, Project, User, Plug-in folder)
  - setting the palette will overwrite **variableColors**
- enable reference limitation option and set top and bottom reference limits
- get and set user upper and lower limits for X and Y axes
  - The **Unit** for X axis tells the plot which unit has to be considered regarding limits values passed to the function. Those values are converted from the Unit to the plot axis unit and converted values are set to limits. See "Plot axes limits and display parameters" in *Ocean for Techlog Developer Guide – Basics*.
- get the display unit of the first variable added to the **Histogram** (see "Display unit" in *Ocean for Techlog Developer Guide – Basics*)
- get and set curves display through **HistogramCurvesDisplay** enum class values (variables, cumulated or both)
- enable/disable normalized, filled and cumulated options

- to disable **cumulated** display option, you must set first histogram **style** to **HistogramStyleContinuousLine**

- change number of bins
- get and set Histogram **style** through **HistogramStyle** enum class values (blocked closed, blocked open or continuous line)
- get and set X and Y axes graduation intervals

The following example shows where variable limits and colors are changed and data are displayed filtering on a facies variable value.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Workspace workspace = Session::current().currentWorkspace();
Dataset dataset =
DomainObject::get(datasetDroid).cast<Dataset>();

Variable gr = dataset.variables().get("GR");
Variable gr_2 = dataset.variables().get("GR_2");

Histogram histogram = Histogram::create(workspace);

if (histogram.canAddVariable(gr))
  histogram.addVariable(gr);
if (histogram.canAddVariable(gr_2))
  histogram.addVariable(gr_2);

histogram.setHorizontalUserLimits(0, 100, gr.unit());
histogram.setVariableColors(QList<QColor>()  << Qt::blue <<
Qt::red);
Variable facies = dataset.variables().find("FACIES04");
if (!facies.isNull())
{
  histogram.setVariableFilter(facies);
  histogram.setVariableFilterValues(QStringList() << "1");
  histogram.setSideBoxVisible(true);
}

lock.release();
```

The following screenshot shows the display result.

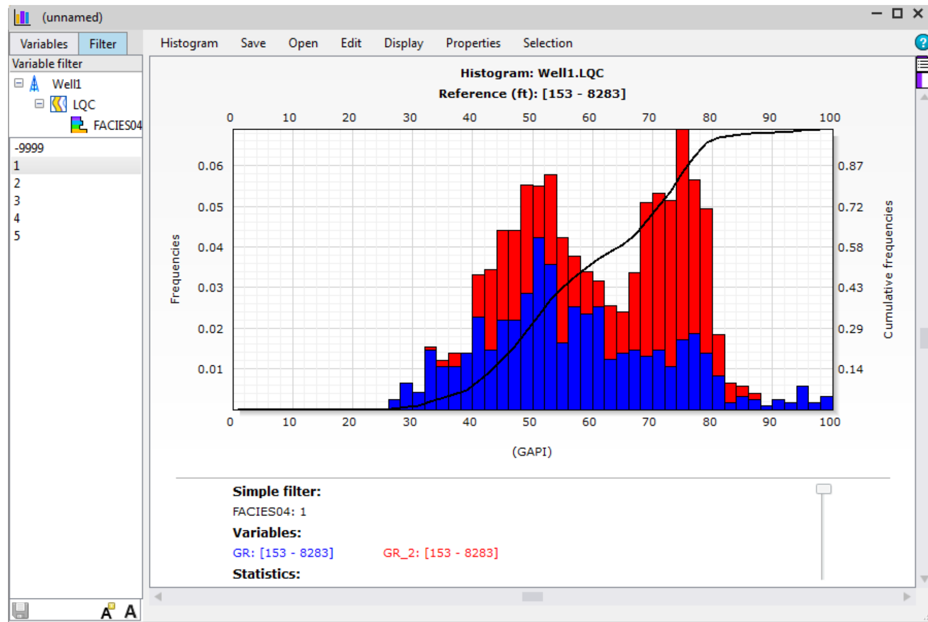


Figure 71 Histogram display options

In the **Histogram** class a create static method is available to instantiate a **Histogram** object passing the parent **Workspace** and a **HistogramTemplateDataBinding** instance.

A **Histogram** can also be created by template into a container plot (matrix-plot) using the dedicated **create** static method.

See the "ContainerPlot Domain Object" section on page 189 for more information on how to create a **Histogram** by template into a container plot.

```
class Histogram : public Plot
{
public:
  …
  static Histogram create(Workspace workspace, const
  HistogramTemplateDataBinding &histogramTemplateDataBinding);

  static Histogram create(const ContainerPlotPosition
    &containerPlotPosition, const HistogramTemplateDataBinding
    &histogramTemplateDataBinding) ;
  …
}
```

The **HistogramTemplateDataBinding** contains the layout template saved at a storage level and the well or the dataset to apply to this template.

```
class HistogramTemplateDataBinding
{
public:
  HistogramTemplateDataBinding(const HistogramTemplate
  &histogramTemplate, const Well &well);

  HistogramTemplateDataBinding(const HistogramTemplate
  &histogramTemplate, const Dataset &dataset);
```

```
}
```

In Techlog after setting a histogram for a well, you can save the histogram as a template.

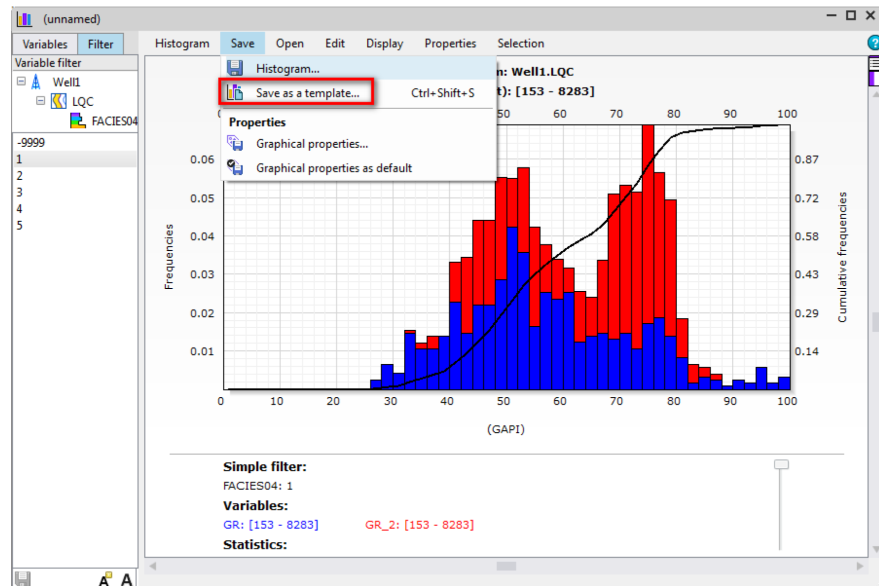


Figure 72 Histogram saved as a template

Note: A template saves only the content of a histogram, whereas a histogram saves specific variables and the complete display.

You will specify the template name and at the level where is stored the template, modeled in Ocean with **`HistogramTemplate`** class and enum class **`StorageLevel`**.

```
class HsitogramTemplate
{
public:
   static HistogramTemplate get(const StorageLevel level, const
   QString &name);
   static bool exists(const StorageLevel level, const QString
   &name);

   StorageLevel level() const;
   QString name() const;
    …
}
```

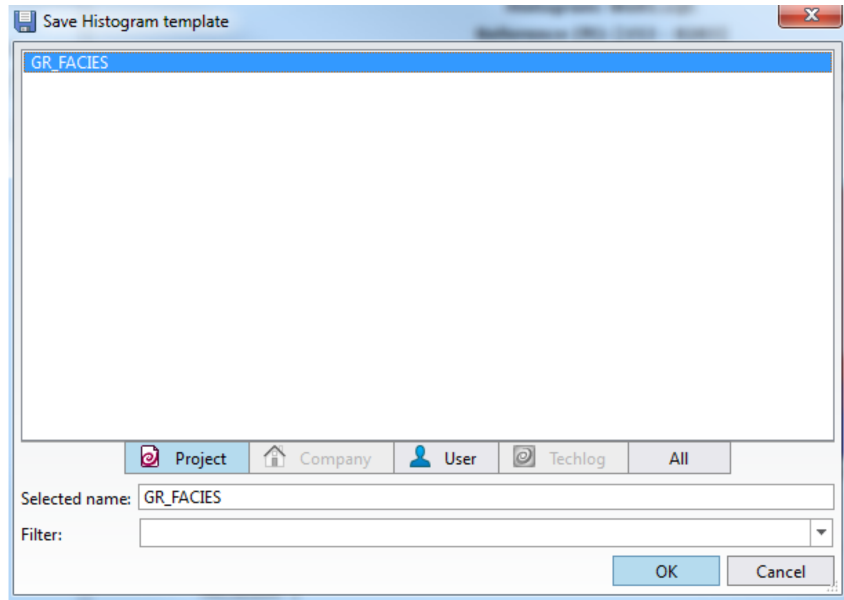


Figure 73 Techlog storage levels

The histogram template is saved to the Project browser under Histograms. You can retrieve a histogram template in the project at a later time, double click on the template and decide to apply the histogram template as well template or as a dataset template.

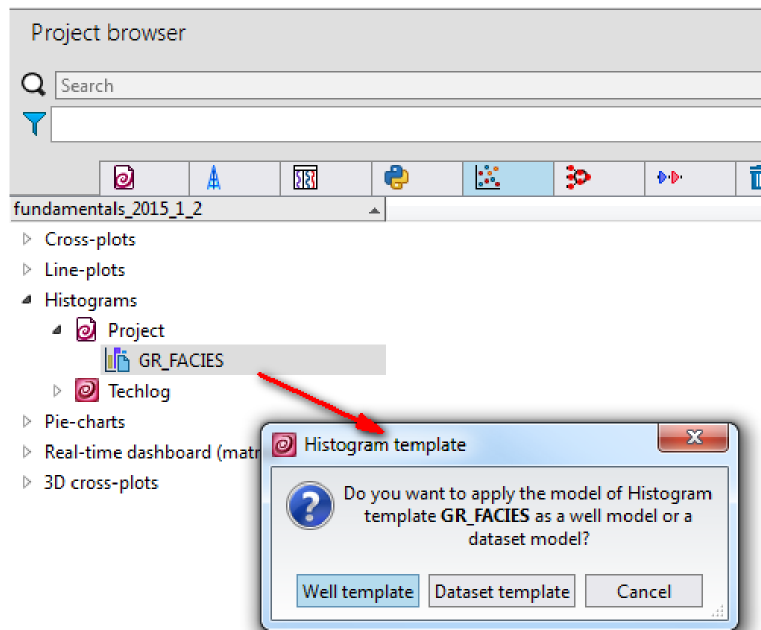


Figure 74 Histogram template stored in the project browser at the project level

Ocean introduces an additional level which is the plug-in level and the ability to provide histogram templates in the plug-in folder. At the plug-in level, the histogram template must be stored in a directory named "HistogramTemplates" closed to the plug-in dll in the plug-in folder.

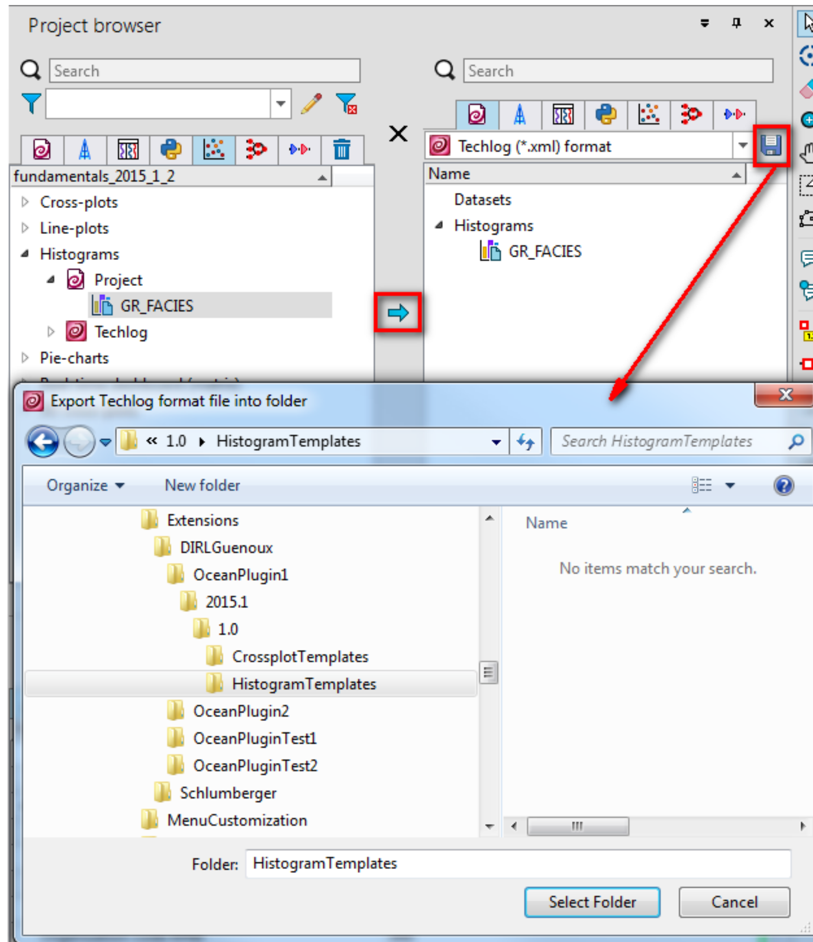


Figure 75 Export template to HistogramTemplates folder at the plug-in level

The following example shows how to open a histogram template stored at the plug-in level.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Project project = Session::current().mainProject();
Well well = project.findWell("Well2");
if (well.isNull())
{
  lock.release();
  return;
}

Dataset dataset = well.findDataset("LQC");
if (dataset.isNull())
{
  lock.release();
  return;
```

```
}

if (!HistogramTemplate::exists(StorageLevelPlugin,
"GR_FACIES"))
{
  lock.release();
  return;
}

HistogramTemplate histogramTemplate =
HistogramTemplate::get(StorageLevelPlugin, "GR_FACIES");

HistogramTemplateDataBinding templateDataBinding =
HistogramTemplateDataBinding(histogramTemplate, dataset);

Histogram histogram = Histogram::create(workspace,
templateDataBinding);

lock.release();
```

Plots multi well

Today only one plot multi-well is exposed with Ocean. This is the multi-well crossplot.

A Crossplot compares multiple measurements made at a single time or location along 2 or more axes. A multi-well crossplot compares data coming from more than one well, at a time.

1. In Techlog, select **Plot** tab > **Multi-well** group > **Cross-plot** action item.

   The Input(s) (family, alias or variable) selection window opens. This window allows you to select the mnemonic (family name, alias, or variable name) to be used within the method.

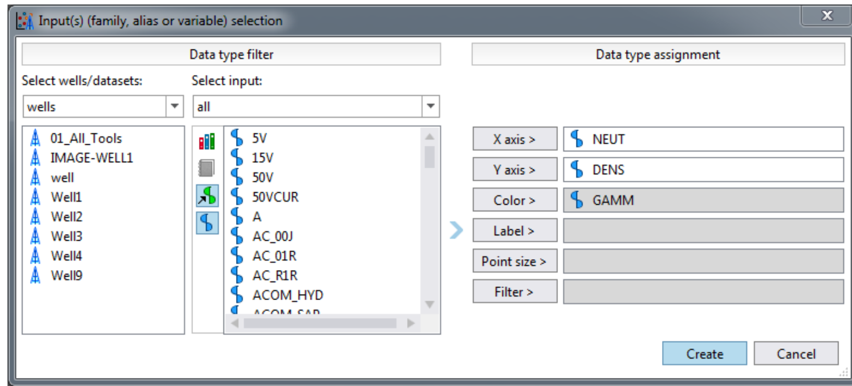


Figure 76 Inputs selection window

Note: In this window, a white cell in the Data type assignment column indicates that this input is mandatory; a gray cell indicates that the input is optional.

2. Select the family, alias, or variable to display and click **Create**.
3. In the plot, drag the datasets into the variable side box or the display area of the plot.

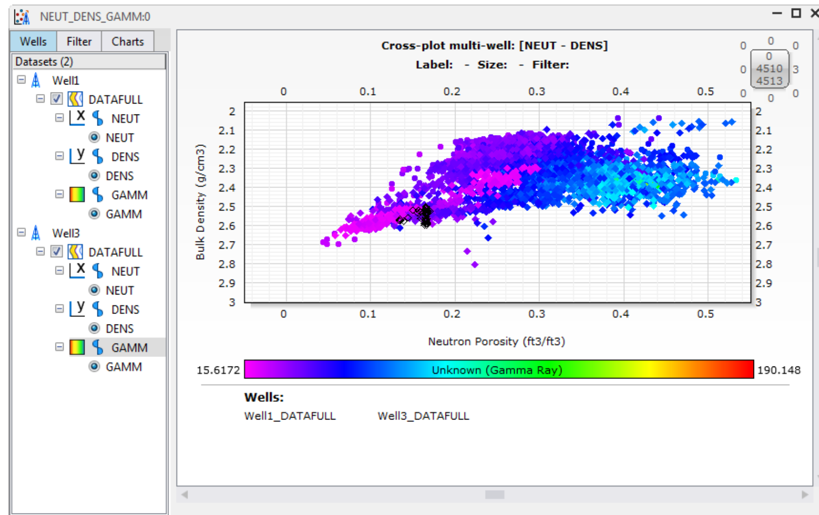


Figure 77 Multi-well crossplot

MultiWellCrossPlot Domain Object

The multi-well crossplot is modeled in Ocean with the **`MultiWellCrossPlot`** domain object.

The **`MultiWellCrossPlot`** class inherits from the **`Plot`** class and does not support name. A **`MultiWellCrossPlot`** object is instancied through **`create`** static method of the class with the parent **`Workspace`** object as argument. The uniqueness of a **`MultiWellCrossPlot`** object is handled by the system. The only way to retrieve a **`MultiWellCrossPlot`** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

You can also add a **`MultiWellCrossPlot`** to a container plot (matrix-plot) using the dedicated **`create`** static methods.

See the "ContainerPlot Domain Object" section on page 189 for more information on how to create a multi-well crossplot into a container plot.

```
class MultiWellCrossPlot : public Plot
{
public:
  static MultiWellCrossPlot create(Workspace workspace);

  static MultiWellCrossPlot create(const ContainerPlotPosition
    &containerPlotPosition);

  void setXAxis(AssignmentType assignmentType, const QString
  &assignmentValue);
  void setYAxis(AssignmentType assignmentType, const QString
  &assignmentValue);
  void setColor(AssignmentType assignmentType, const QString
  &assignmentValue);
  void setLabel(AssignmentType assignmentType, const QString
  &assignmentValue);
  void setPointSize(AssignmentType assignmentType, const QString
  &assignmentValue);
  void setFilter(AssignmentType assignmentType, const QString
  &assignmentValue);

  void refuseInterpolatedInputs();
  void acceptInterpolatedInputs();

  DomainObjectCollection<Dataset> datasets() const;
  void addDataset(const Dataset &dataset);
  bool canAddDataset(const Dataset &dataset);
  void clearDatasets();
  …
};
```

For each dimension (X, Y, Color, Label and Point size) you have to define the assignement rule for datasets added to the **`MultiWellCrossPlot`**. This is passed to setters through the **`AssignmentType`** enum value (Family, Alias or Variable) and the assignement value (Family name, Alias name or Variable name).

You can refuse or accept to search by interpolation in all datasets of the well for variables that answer to assignement rules set on each dimension of the **MultiWellCrossPlot**. The default value is set to accept.

When a **MultiWellCrossPlot** is created programmatically with Ocean the default behavior is to not show to the end user the Inputs selection window before to open the plot.

The user can at any time open the selection window clicking on **Cross-plot (multi-well)** contextual tab > **Cross-plot** group > **Family change** action item and changes assignment types set to each dimension of the plot.

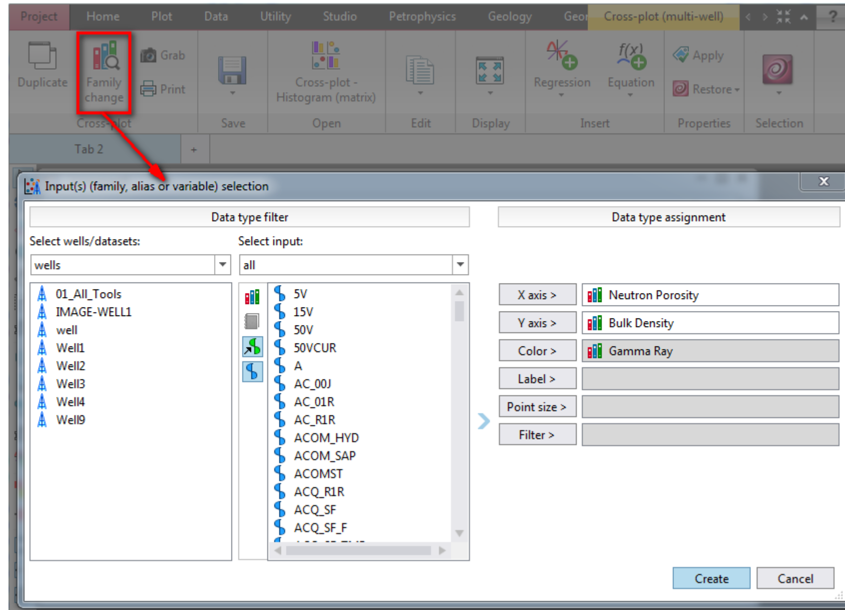


Figure 78 Assignment types set to MultiWellCrossPlot dimensions

The last step is to add dataset to the **MultiWellCrossPlot**. The best practice is to call **canAddDataset** method before **addDataset** in order to avoid throwing an exception.

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

MultiWellCrossPlot multiWellCrossPlot =
MultiWellCrossPlot::create(workspace);

multiWellCrossPlot.setXAxis(AssignmentTypeVariable, "NEUT");
multiWellCrossPlot.setYAxis(AssignmentTypeVariable, "DENS");
multiWellCrossPlot.setColor(AssignmentTypeVariable, "GAMM");

multiWellCrossPlot.refuseInterpolatedInputs();

Project project = Session::current().mainProject();
Well well1 = project.getWell("Well1");
```

SLB-Private

```
Dataset dataset1 = well1.findDataset("DATAFULL");

if (multiWellCrossPlot.canAddDataset(dataset1))
  multiWellCrossPlot.addDataset(dataset1);

Well well2 = project.getWell("Well3");
Dataset dataset2 = well2.findDataset("DATAFULL");

if (multiWellCrossPlot.canAddDataset(dataset2))
  multiWellCrossPlot.addDataset(dataset2);

multiWellCrossPlot.setSideBoxVisible(true);


lock.release();
```

The resulting multi-well crossplot:

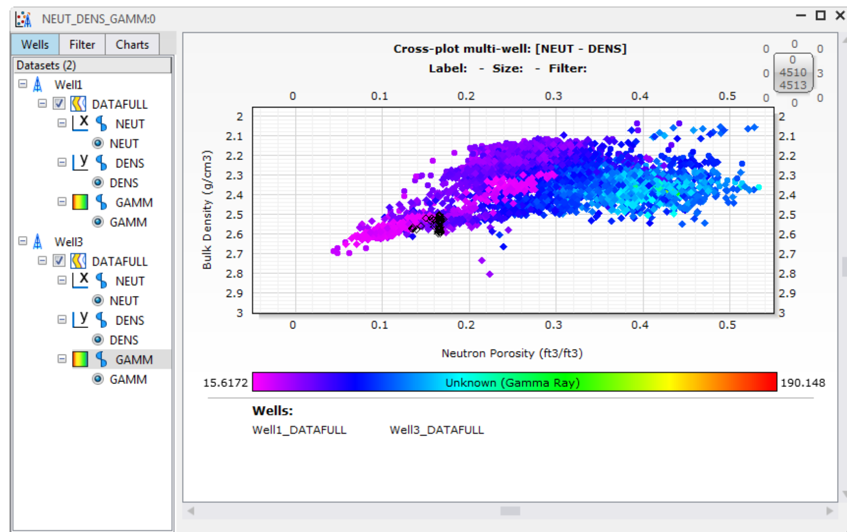


Figure 79 Multi-well crossplot

The default display options can be changed through API's available below.

```
class MultiWellCrossPlot : public Plot
{
public:
  QString title() const;
  void setTitle(const QString &title);
  QString subtitle() const;
  void setSubtitle(const QString &title);

  bool isSideBoxVisible() const;
  void setSideBoxVisible(bool isSideBoxVisible);

  GlobalZonation findSelectedZonation() const;
  QList<GlobalZone> selectedZones() const;
  void setSelectedZones(const GlobalZonation
  &selectedZonation, const QList<GlobalZone> &selectedZones);
```

```
  QStringList variableFilterValues() const;
  void setVariableFilterValues(const QStringList &values);

  double horizontalUserLowerLimit() const;
  double horizontalUserUpperLimit() const;
  double verticalUserLowerLimit() const;
  double verticalUserUpperLimit() const;
  void setHorizontalUserLimits(const double
    horizontalUserLowerLimit, const double
    horizontalUserUpperLimit);
  void setVerticalUserLimits(const double
    verticalUserLowerLimit, const double
    verticalUserUpperLimit);

  void setHorizontalLimitType(AxisLimitType type);
  AxisLimitType horizontalLimitType() const;
  void setVerticalLimitType(AxisLimitType type);
  AxisLimitType verticalLimitType() const;

  bool isHorizontalAxisInverted() const;
  bool horizontalAxisInverted() const;
  void setHorizontalAxisInverted(bool horizontalAxisInverted);
  bool isVerticalAxisInverted() const;
  bool verticalAxisInverted() const;
  void setVerticalAxisInverted(bool verticalAxisInverted);

  bool isDatasetEnabled(const Dataset & dataset) const;

  enum EventType
  {
    SelectedDatasetChanged
  };
  …
};
```

The properties of **`MultiWellCrossPlot`** class allow you:

- get and set a title and subtitle to the **`MultiWellCrossPlot`**
- hide/show the variable side box
- **`setSelectedZones`** gives the ability to select some **`GlobaZone`** in a **`GlobalZonation`** and display its in the **`Logview`**
- get and set filter values to the filter tab of the plot
- get and set user upper and lower limits for horizontal and vertical plot axes
- change horizontal and vertical axes limit type to variable, family or user
- invert vertical and horizontal axes
- get the dataset states used by the multiwell crossplot through the **`isDatasetEnabled`** function

The **`SelectedDatasetChanged`** signal is emitted when a dataset is selected or unselected in the “Variables” side box of the plot

To include the signal argument and declare the slot receiver in the activity header file:

```
#include "tsdkmultiwellcrossplotselecteddatasetchangedargs.h"
```

```
private slots:
  void onMultiWellCrossPlotSelectedDatasetChanged(
    const Slb::Ocean::Techlog::
    MultiWellCrossPlotSelectedDatasetChangedArgs &args);
```

MultiWellCrossPlot domain object can subscribe to this signal as follow:

```
void
activity::createMultiWellCrossPlotSignals(MultiWellCrossPlot
multiWellCrossPlot)
{
  multiWellCrossPlot.connect(
MultiWellCrossPlot::SelectedDatasetChanged, this,
SLOT(onMultiWellCrossPlotSelectedDatasetChanged(const
Slb::Ocean::Techlog::MultiWellCrossPlotSelectedDatasetChangedA
rgs&)));
}
```

SelectedDatasetChanged Signal

The **SelectedDatasetChanged** signal includes an argument that gives the **Dataset** domain object that has been enabled / disabled in the "Variables" side box of the plot. This argument is modeled in Ocean for Techlog through the **MultiWellCrossPlotSelectedDatasetChangedArgs** class.

```
class MultiWellCrossPlotSelectedDatasetChangedArgs :
public SignalArgsT<MultiWellCrossPlot>
{
public:
 …
 Dataset dataset() const;
 bool isEnabled() const;
};
```

The slot handler accesses the needed information from the signal argument.

```
void activity::onMultiWellCrossPlotSelectedDatasetChanged(const
Slb::Ocean::Techlog::MultiWellCrossPlotSelectedDatasetChangedA
rgs &args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  Dataset dataset = args.dataset();

  qWarning() << "Dataset " << dataset.well().name() << "."
  << dataset.name() << " is enabled = " << args.isEnabled();
```

```
  lock.release();
}
```

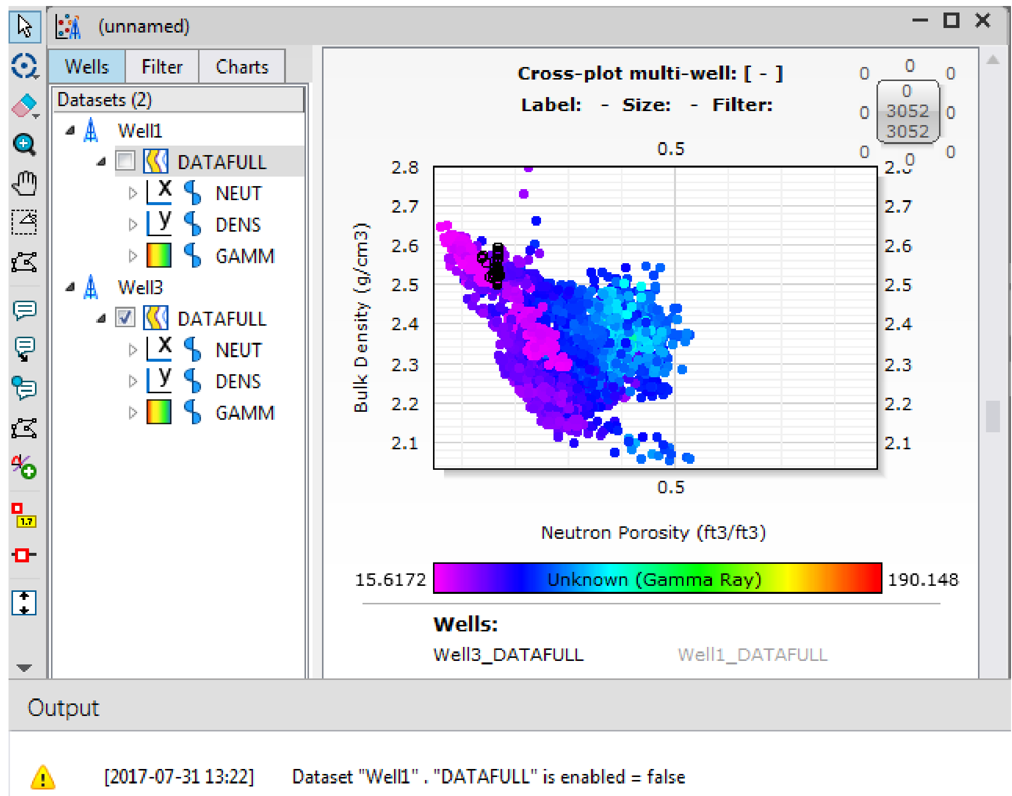


Figure 80 Selected dataset changed signal

Advanced plotting

The **Advanced plotting** group in the **Plot** tab contains viewers that allow to locate wellbores within their spatial position as the **3D Field Viewer** cuurently exposed with the Ocean framework.

The **3D Field Viewer** provides a 3 dimensional well display environment. You can also display continuous variables, wellbore images and maps.

To display a well within the **3D Field Viewer**, it is mandatory to have an index dataset containing variables with the following families: X Offset, Y Offset, True Vertical Depth Sub Sea and Measured Depth.

See "Reference dataset" section in the *Techlog Help Center*.

To plot multiple wells within the same 3D viewer, it is mandatory to define the X and Y well properties.

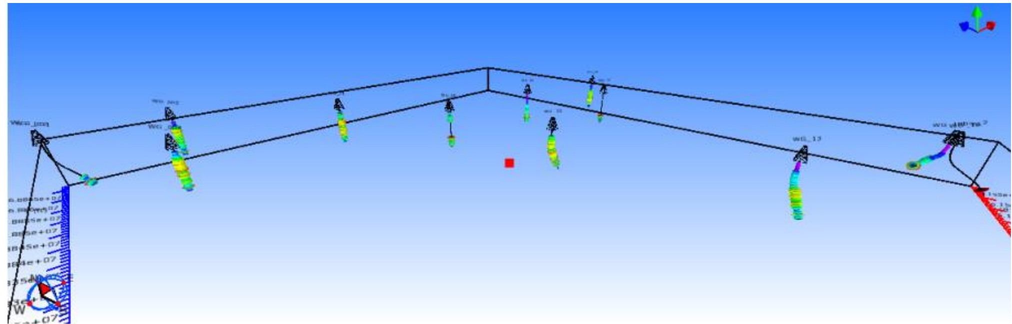


Figure 81 Inputs selection window

Field3DView Domain Object

The 3D Field Viewer is modeled in Ocean with the **Field3DView** domain object.

The **Field3DView** class inherits from the **Plot** class and does not support name. A **Field3DView** object is instancied through **create** static method of the class with the parent **Workspace** object as argument. The uniqueness of a **Field3DView** object is handled by the system. The only way to retrieve a **Field3DView** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

```
class Field3DView : public Plot
{
public:
  static Field3DView create(Workspace workspace);
  static Field3DView tryCreate(Workspace workspace);

  ReturnValue<bool> tryAddVariable(const Variable &variable);
  DomainObjectCollection<Variable> variables() const;

  bool isDepthListenerEnabled() const;
  void setDepthListenerEnabled(bool enabled);
  bool isDepthInteractionEnabled() const;
  void setDepthInteractionEnabled(bool enabled);
};
```

The **Field3DView** class provides a **tryCreate** function that can be used to instantiate a **Field3DView** domain object in a degraded mode without a 3D View

license feature available or activated into Techlog. Otherwise the plug-in developer can use the **create** static function to instantiate the **Field3DView** domain object that force him to declare the 3D View feature as a dependency of his plug-in through the **PluginInformation::addFeatureDependency** function.

See "Declare plug-in information" section in *Ocean for Techlog Developer Guide - Basics* for more information on how to add a feature dependency to your plug-in.

Once the **Field3DView** domain object is created you try to add a variable to the 3D viewer using the **tryAddVariable** function. This function can return false for reasons described in the "Advanced plotting" section on page 133.

The example below shows how to add all "GAMM" variables from "LQC" datasets in the project to a **Field3DView** through the **tryAddVariable** function. A warning message returns to the end-user for which reason the variable can't be added to the 3D field viewer.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();
Workspace workspace = Session::current().currentWorkspace();

Field3DView field3DView = Field3DView::create(workspace);

foreach(Well well, project.wells())
{
  Dataset dataset = well.findDataset("LQC");
  if (dataset.isNull())
    continue;

  Variable gr = dataset.findVariable("GAMM");
  if (gr.isNull())
    continue;

  bool variableAdded = field3DView.tryAddVariable(gr);
  if (!variableAdded)
  {
    qWarning() << "Variable " << well.name() << "." <<
dataset.name() << ".GAMM can't be added to the 3D Field Viewer for
following reason(s):";

    if (!well.datasets().contains("Index"))
      qWarning() << "\t - " << well.name()
      << " doesn't contain an Index dataset";
    else
    {
      Dataset index = well.getDataset("Index");
      bool xOffset = false, yOffset = false, tvdss = false, md =
false;
      foreach(Variable variable, index.variables())
```

```
      {
        if (variable.family() == TechlogFamily::getXOffset())
          xOffset = true;
        else if (variable.family() ==
           TechlogFamily::getYOffset())
          yOffset = true;
        else if (variable.family() ==
           TechlogFamily::getTrueVerticalDepthSubSea())
          tvdss= true;
        else if (variable.family() ==
           TechlogFamily::getMeasuredDepth())
          md = true;
      }

      if (!xOffset)
        qWarning() << "\t - " << well.name()
        << ".Index doesn't contain a X Offset variable";
      if (!yOffset)
        qWarning() << "\t - " << well.name()
        << ".Index doesn't contain a Y Offset variable";
      if (!tvdss)
        qWarning() << "\t - " << well.name()
        << ".Index doesn't contain a TVDSS variable";
      if (!md)
        qWarning() << "\t - " << well.name()
        << ".Index doesn't contain a MD variable";
     }

     DataProperty x = well.findDataProperty("X");
     DataProperty y = well.findDataProperty("Y");
     if (x.isEmpty() || x.value().isEmpty())
       qWarning() << "\t - " << well.name()
       << "doesn't contain a valid X data property";
     if (y.isEmpty() || y.value().isEmpty())
       qWarning() << "\t - " << well.name()
       << "doesn't contain a valid Y data property";
  }
}

lock.release();
```

The screenshot below shows gamma ray variables for different wells displayed in the 3D Field Viewer.

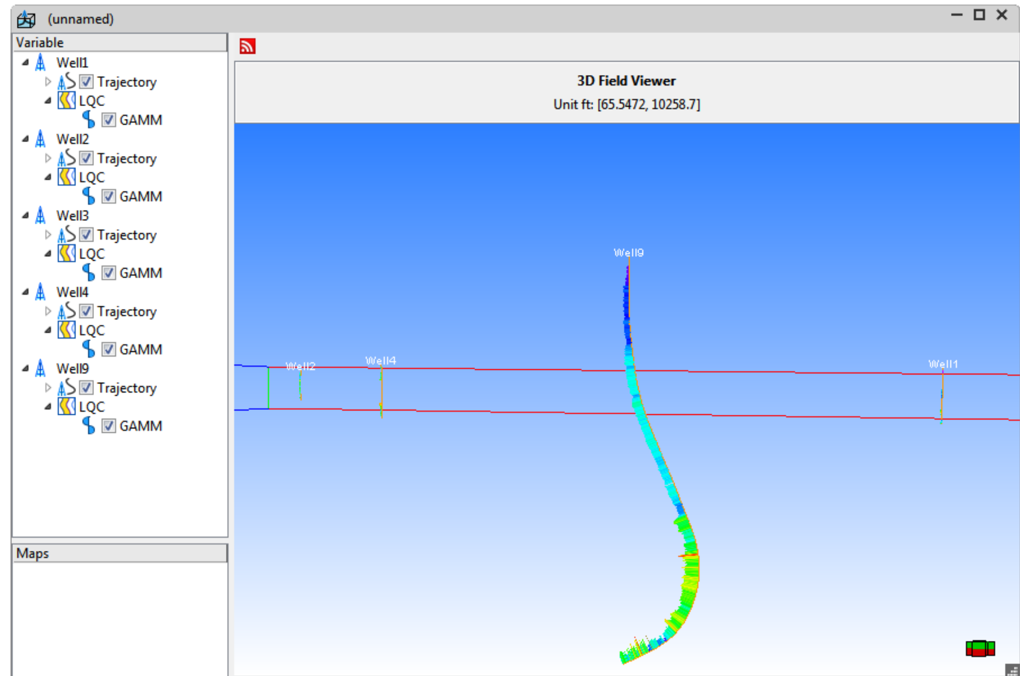


Figure 82 3D Field Viewer with GAMM variables from different wells

The Field3DView allows you to:

- enable / disable the depth listener of the viewer
- listen to depth interaction from the **`Logview`**

Regression

To compute a regression from the displayed data, either by zone and filter or selection, Techlog uses the Linear regression method. This is a statistic approach to model the relationship between one response variable Y and one or more explanatory variables X. The linear regression focuses on the conditional probability distribution of Y given X.

There are two main objectives of linear regression. If the objective is a prediction, you can use the linear regression to fit a predictive model to an observed data set of Y and X values. After developing such a model, if an additional value of X is given without its value of Y, you can use the fitted model to make a prediction of the value of Y. It helps identifying the trends along the wells to get an idea of the behavior of given parameters on a given field.

Natively in Techlog the end-user has the ability to add a regression to a plot through the **Insert** menu > **Regression**.

The available regression options are shown in the screenshot below:

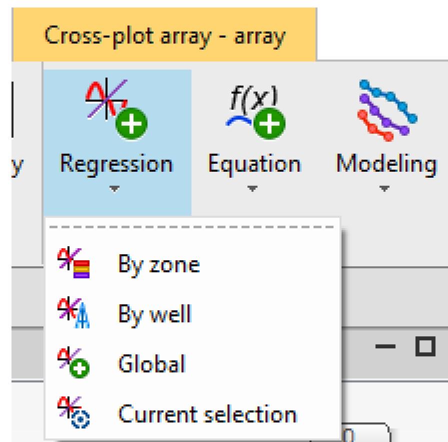


Figure 83 Regression options

Only the "Global" regression option is exposed with Ocean modeled with the **`Regression`** domain object. This object allows to add a regression taking into consideration all the displayed data.

Regression Domain Object

You can add a **`Regression`** to **`CrossPlot`**, **`LinePlot`**, **`CrossPlotArrayVariableArray`** and **`CrossPlotArrayArray`** through the **`create`** static functions below:

```
class Regression : public DomainObject
{
public:
  static Regression create(const QString &name, CrossPlot &plot);
  static Regression create(const QString &name, LinePlot &plot);
  static Regression create(const QString &name,
  CrossPlotArrayVariableArray &plot);
  static Regression create(const QString &name,
  CrossPlotArrayArray &plot);
  …
};
```

In each of those plots a **regressions** function returns the **DomainObjectCollection** of **Regression** created into the plot.

```
class CrossPlot : public Plot
{
public:
  DomainObjectCollection<Regression> regressions() const;
  …
};
```

Once the **Regression** domain object has been created you can change the following parameters related to the linear regression computation:

```
class Regression : public DomainObject
{
public:
  void setType(RegressionType type);
  RegressionType type() const;
  void setPolynomialOrder(int order);
  int polynomialOrder() const;
  void setExplanatoryVariablesTransformationType
  (TransformationType type);
  TransformationType explanatoryVariablesTransformationType()
  const;

  void setColor(const QColor &color);
  QColor color() const;
  void setThickness(int thickness);
  int thickness() const;
  …
};
```

Those properties are:

- Regression type that determines the independent and dependent variables (major axis by default) handled by the **RegressionType** enum values where:
  - X function of Y and Y function of X: predicts one variable from the other. In the regression model, you assume that the predictor variable (for example, the X variable in Y/X regression) has no error and that the uncertainty is in the predicted variable. In this case, you minimize the distance between the data points and the regression line in a direction parallel to the X axis (for X/Y regression) or parallel to the Y axis (for the Y/X case).
  - Major axis: recognizes the uncertainties for both the X and Y data points. If neither X or Y variables are related to each other, do not privilege the linear regressions X/Y or Y/X. The desired fit line in such a case is a straight line drawn by the sum of the squared distances between each data point and the target line is at a minimum.
  - Reduced major axis: the solution is a straight line drawn so that the sum of the rectangular areas defined by the distances parallel to the vertical and horizontal axes between the straight line and each data point is a minimum. You try to minimize the sum of the absolute value of the measured distance products.
  - Quantiles: the quantile regression estimates the relation of independent variables and specific quantile of the dependent variable. The quantile

regression ensures that x percent of the samples in the Crossplot is below the X quantile regression. You can apply and explain any transformation to the explanatory variables. The quantile is set from the Quantile value property in the Properties window of the regression.

  - Swanson: the Swanson's regression is the result of the combination of three quantile regressions (q10, q50 and q90) weighted by the Swanson's rule. Y = 0.3 * q10(x)+0.4&q50(x)+0.30*q90(x). Swanson's regression is a conditional mean estimator invariant to monotonic transformations of the dependent variable. When building a Porosity/Permeability model with a logarithmic transformation, the arithmetic mean conditioned to the independent variables are always underestimated. With the Swanson's method, you get a more robust estimate of the mean for upscaling and permeability models.

To summarize, you should use the Major axis regression and Reduced major axis regression when the error rate in X exceeds one-third of the error rate in Y (McArdle 1988). You can consider each of them as a line of calibration rather than a line of prediction. The choice whether to apply one method rather than the other is controlled by the fact that with the Reduced major axis regression, the results are optimized when the units of each axis are different from each other. The use of either method on standardized data (data is centered by removal of the mean and normalized by its standard deviation) gives similar results. We recommend that you use Reduced major axis regression when units for each axis are different. You must use the classical linear regression X/Y or Y/X method when trying to predict one variable from another. In the regression model, the predictor variable (for example, the X variable in Y/X regression) is considered to have no error and all the uncertainty is considered to reside in the predicted variable. Therefore, each model provides an output predicted curve that is as close as possible to the mean of the observations of the predicted variable used to build the fit-line in each case.

- Explanatory variables properties that concern the independent variable (X axis by default)
  - Transformation type can be set to Log10, Exp or none value through the **`TransformationType`** enum class
  - Polynomial order can be set to an integer value going from 1 to 5
- get and set color and thickness properties of the linear regression

SLB-Private

Customize plots

Through Ocean you can add some containers called graphic scenes into a track of the Logview and the charting area of any Techlog native or custom plots.

See the "Custom plots" section on page 178 for more information on custom plot.

Some graphic items can be added to those containers. Each of the graphic items and also the container itself are domain objects and can subscribe to some mouse events that allow you to add custom user interactions with the plot.

The graphics container is modeled in Ocean through the **`GraphicsScene`** domain object and is used to display two types of graphic items within the chart area of the plot that are:

- **`ChartParameter`** domain object that allows you to add parametric equation to the plot
- **`GraphicsItem`** domain object base class that allows you to draw:
  - Shapes as ellipse, line, rectangle and triangle through the corresponding domain objects that are **`Ellipse2d`**, **`Line2d`**, **`Rectangle2d`** and **`Triangle2d`** derived from **`Shape`** base class.
  - Points list **`as`** points cloud and polyline through the corresponding domain objects that are **`PointsCloud`** and **`Polyline2d`** derived from **`PointsList`** base class.
  - Labels through **`MarkerLabel`** domain object.

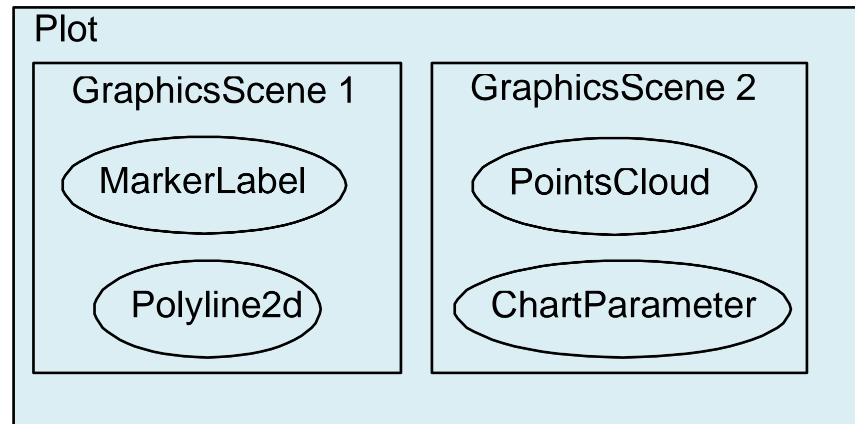


Figure 84 GraphicsScene containing graphics items

Mouse bar, tool bar and context menu of native and custom plots can also be extended through Ocean adding custom actions. This is available through **`Action`** class in Ocean.

GraphicsScene Domain Object

You can add a **`GraphicsScene`** to any plot derived from **`Plot`** base class or to a **`Track`** object of the **`Logview`** to draw something.

Create a GraphicsScene in a Plot

GraphicsScene domain object can be enumerated from a **Plot** instance (or an object derived from **Plot** base class) through **graphicsScenes** domain object collection. A **GraphicsScene** object can be retrieved by its name from the parent **Plot** domain object through **findGraphicsScene** and **getGraphicsScene** functions.

```
class Plot : public DomainObject
{
public:
  DomainObjectCollection<GraphicsScene> graphicsScenes()
  const;
  GraphicsScene findGraphicsScene(const QString
  &graphicsSceneName) const;
  GraphicsScene getGraphicsScene(const QString
  &graphicsSceneName) const;
  …
};
```

In this domain object collection a **GraphicsScene** must be unique by its name so it is recommended to check for graphics scene name existence in parent **Plot** before creating it.

```
class GraphicsScene : public DomainObject
{
public:

  static GraphicsScene create(const QString &name, Plot plot);

  …
};
```

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
crossplotDroid);

CrossPlot crossPlot =
DomainObject::get(crossplotDroid).cast<CrossPlot>();

GraphicsScene graphicsScene =
crossPlot.graphicsScenes().get("MyGraphicsScene");

lock.release();
```

Create a GraphicsScene in a NormalTrack of a Logview

GraphicsScene domain object can be enumerated from a **NormalTrack** instance through **graphicsScenes** domain object collection.

A **GraphicsScene** object can be retrieved by its name from the parent **NormalTrack** domain object through **findGraphicsScene** and **getGraphicsScene** functions.

```
class NormalTrack : public Track
```

```
{
public:
  DomainObjectCollection<GraphicsScene> graphicsScenes()
  const;
  GraphicsScene findGraphicsScene(const QString
  &graphicsSceneName) const;
  GraphicsScene getGraphicsScene(const QString
  &graphicsSceneName) const;
  …
};
```

In this domain object collection a **GraphicsScene** must be unique by its name so this is recommended to check for graphics scene name existence in parent **NormalTrack** before creating it. Additionally, you must pass a **Dataset** instance to the **create** method. This **Dataset** gives to the **GraphicsScene** its size on Y axis corresponding to the top and bottom depth of the **Dataset** displayed into the **NormalTrack**.

```
class GraphicsScene : public DomainObject
{
public:
  static GraphicsScene create(const QString &name, NormalTrack
  track, Dataset dataset);

  const Unit xUnit() const;
  void setXUnit(const Unit &unit);


  double horizontalUserLowerLimit() const;
  double horizontalUserUpperLimit() const;
  void setHorizontalUserLimits(double horizontalUserLowerLimit,
  double horizontalUserUpperLimit);

  …
};
```

Once the **GraphicsScene** object has been instancied to draw something into the **NormalTrack** you first need to indicate to the **GraphicsScene** what are the X limits and unit of the drawing area.

Note: if X limits and unit properties aren't passed to the **GraphicsScene** object, the range of values of the **GraphicsScene** in the **NormalTrack** is going from 0 to 1.

Example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Variable gr = DomainObject::get(variableDroid).cast<Variable>();

Workspace workspace = Session::current().currentWorkspace();

// create logview
Logview logview = Logview::create(workspace);

// create track
```

```
NormalTrack track = NormalTrack::create( QLatin1String( "track 1" ), logview );

// create a track item depending on gamma ray variable type
LineTrackItem trackItem = LineTrackItem::create( track, gr);

// Create the graphicsScene
GraphicsScene graphics = GraphicsScene::create("MyGraphicsScene", track, gr.dataset());

// set the unit, lower and upper limits of the variable to the graphicsScene
graphics.setXUnit(gr.unit());
graphics.setHorizontalUserLimits(trackItem.horizontalUserLowerLimit(), trackItem.horizontalUserUpperLimit());

lock.release();
```

GraphicsScene display options

The default display options can be changed through API's available below.

```
class GraphicsScene : public DomainObject
{
public:
  void sendToBack(const GraphicsItem &itemToMove);
  void bringToFront(const GraphicsItem &itemToMove);
  void sendBehind(const GraphicsItem &itemToMove, const GraphicsItem &referenceItem);
  void bringInFrontOf(const GraphicsItem &itemToMove, const GraphicsItem &referenceItem);
  void sendBackwards(const GraphicsItem &itemToMove);
  void bringForward(const GraphicsItem &itemToMove);

  bool isHorizontalAxisInverted() const;
  void setHorizontalAxisInverted(bool inverted);
  bool canDisplayHeaderInformation() const;
  void setDisplayHeaderInformation(const bool displayable);
   …
};
```

The properties of **`GraphicsScene`** class allow you:

- organize the graphics items display within the **`GraphicScene`** through a set of functions as **`sendToBack`**, **`bringToFront`**, **`sendBehind`** …

  See the "GraphicsItem Domain Objects" section 155 for more information on how to create graphics items in a **`GraphicsScene`**.

- set the horizontal axis to inverted in case the **`GraphicScene`** is added to a Logview track that display variable values with the axis set to inverted

- throw a plug-in exception if the parent of **GraphicScene** isn't a **Track**

- show / hide the **GraphicScene** in the header of the track
  - not taken in account if the parent of **GraphicsScene** isn't a **Track**

Domain object collections

GraphicsScene domain object returns **DomainObjectCollection** of graphic items in the scene through some different public methods. Those methods are splitted by graphic item types.

```
class GraphicsScene : public DomainObject
{
public:
  …
  const DomainObjectCollection<GraphicsItem> items()
  const;
  const DomainObjectCollection<MarkerLabel> markerLabels()
  const;
  const DomainObjectCollection<ChartParameter>
  chartParameters() const;
  const DomainObjectCollection<PointsCloud> pointsClouds()
  const;
  const DomainObjectCollection<Polyline2d> polylines() const;
  const DomainObjectCollection<Shape> shapes() const;
};
```

GraphicsScene signals

A **GraphicsScene** domain object can subscribe on some mouse events.

```
class GraphicsScene : public DomainObject
{
public:
 …
 enum EventType
 {
   MouseClickPressed,
   MouseClickReleased,
   MouseDoubleClick,
   MouseMoving,
   PolygonSelectionChanged,
   KeyPressed
 };
}
```

The signals are emitted whenever:

- **MouseClickPressed** – a left click is performed in the graphics scene
- **MouseClickReleased** – a left click is released in the graphics scene

- **MouseClickDoubleClick** – a left double click is performed in the graphics scene
- **MouseMoving** – the mouse is hovering over the graphics scene
- **PolygonSelectionChanged** – shapes are selected through "Polygon type selection" tool enabled in plot mousebar
- **KeyPressed** – a key is pressed while the graphics scene has the focus

The following shows how to include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkgraphicssceneinteractionargs.h"
#include "tsdkpolygonselectionchangedargs.h"
```

```
private slots:
  void onMouseClickPressed(
  const Slb::Ocean::Techlog::GraphicsSceneInteractionArgs
&args);
  void onMouseDoubleClick(
  const Slb::Ocean::Techlog::GraphicsSceneInteractionArgs
&args);
  void onMouseClickReleased(
  const Slb::Ocean::Techlog::GraphicsSceneInteractionArgs
&args);
  void onMouseMoving(
  const Slb::Ocean::Techlog::GraphicsSceneInteractionArgs
&args);
  void onKeyPressed(
  const Slb::Ocean::Techlog::GraphicsSceneInteractionArgs
&args);
  void onPolygonSelectionChanged(
  const Slb::Ocean::Techlog::PolygonSelectionChangedArgs
&args);
```

A **GraphicsScene** domain object will connect to the signals as follows:

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
crossplotDroid);

CrossPlot crossPlot =
DomainObject::get(crossplotDroid).cast<CrossPlot>();

GraphicsScene graphicsScene =
crossPlot.graphicsScenes().get("MyGraphicsScene");

lock.release();

lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, graphicsScene);
```

```
graphicsScene.connect(GraphicsScene::MouseClickPressed, this,
SLOT(onMouseClickPressed(const
Slb::Ocean::Techlog::GraphicsSceneInteractionArgs&)));
graphicsScene.connect(GraphicsScene::MouseDoubleClick, this,
SLOT(onMouseDoubleClick(const
Slb::Ocean::Techlog::GraphicsSceneInteractionArgs&)));
graphicsScene.connect(GraphicsScene::MouseClickReleased, this,
SLOT(onMouseClickReleased(const
Slb::Ocean::Techlog::GraphicsSceneInteractionArgs&)));
graphicsScene.connect(GraphicsScene::MouseMoving, this,
SLOT(onMouseMoving(const
Slb::Ocean::Techlog::GraphicsSceneInteractionArgs&)));
graphicsScene.connect(GraphicsScene::KeyPressed, this,
SLOT(onKeyPressed(const
Slb::Ocean::Techlog::GraphicsSceneInteractionArgs&)));
graphicsScene.connect(GraphicsScene::PolygonSelectionChanged,
this, SLOT(onPolygonSelectionChanged(const
Slb::Ocean::Techlog::PolygonSelectionChangedArgs&)));

lock.release();
```

Mouse pressed, double click, released, moving and key pressed signals include an argument that gives mouse coordinates and the key pressed (only fill when the **KeyPressed** event occurred). This argument is modeled in Ocean through the **GraphicsSceneInteractionArgs** class.

```
class GraphicsSceneInteractionArgs : public
SignalArgsT<GraphicsScene>
{
public:
  QPointF clickCoordinates() const;
  Qt::Key key() const;
  …
};
```

Some examples:

```
void activity::onMouseClickPressed(const
Slb::Ocean::Techlog::GraphicsSceneInteractionArgs &args)
{
  QPointF coordinates = args.clickCoordinates();
  qWarning() << "X = " << coordinates.x() << ", Y = " <<
  coordinates.y();
}
```

```
void activity::onKeyPressed(const
Slb::Ocean::Techlog::GraphicsSceneInteractionArgs &args)
{

  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

```

```
GraphicsScene scene = args.sender();
DomainObject parent = scene.parent();
if (!parent.isA<Track>())
{
  lock.release();
  return;
}

Track track = parent.cast<Track>();
Logview logview = track.logview();
int trackPosition = logview.trackPosition(track);
qWarning() << "trackPosition = " << trackPosition;

// duplicate the current track at the right position
if (args.key() == Qt::Key_Plus)
{
  QString trackName =
  QString("TrackRight_%1").arg(QString::number(rand()));

  NormalTrack newTrackRight = NormalTrack::create(trackName,
    logview, trackPosition + 1);

  foreach(TrackItem trackItem,
    track.cast<NormalTrack>().trackItems())
  {
    LineTrackItem lineTrackItem =
    LineTrackItem::create(newTrackRight,
    trackItem.findVariable());

    lineTrackItem.setHeaderName(trackName);

    lineTrackItem.setHeaderNameType(
    HeaderNameTypeVariableAndTitle);
  }
}

// duplicate the current track at the left position
if (args.key() == Qt::Key_Minus)
{
  QString trackName =
  QString("TrackLeft_%1").arg(QString::number(rand()));

  if ((trackPosition - 1) < 0)  trackPosition = 1;

  NormalTrack newTrackLeft = NormalTrack::create(trackName,
    logview, trackPosition - 1);
```

SLB-Private

```
    foreach(TrackItem trackItem,
      track.cast<NormalTrack>().trackItems())
    {
      LineTrackItem lineTrackItem =
      LineTrackItem::create(newTrackLeft,
      trackItem.findVariable());

      lineTrackItem.setHeaderName(trackName);
      lineTrackItem.setHeaderNameType(
      HeaderNameTypeVariableAndTitle);
    }
  }
  lock.release();
}
```

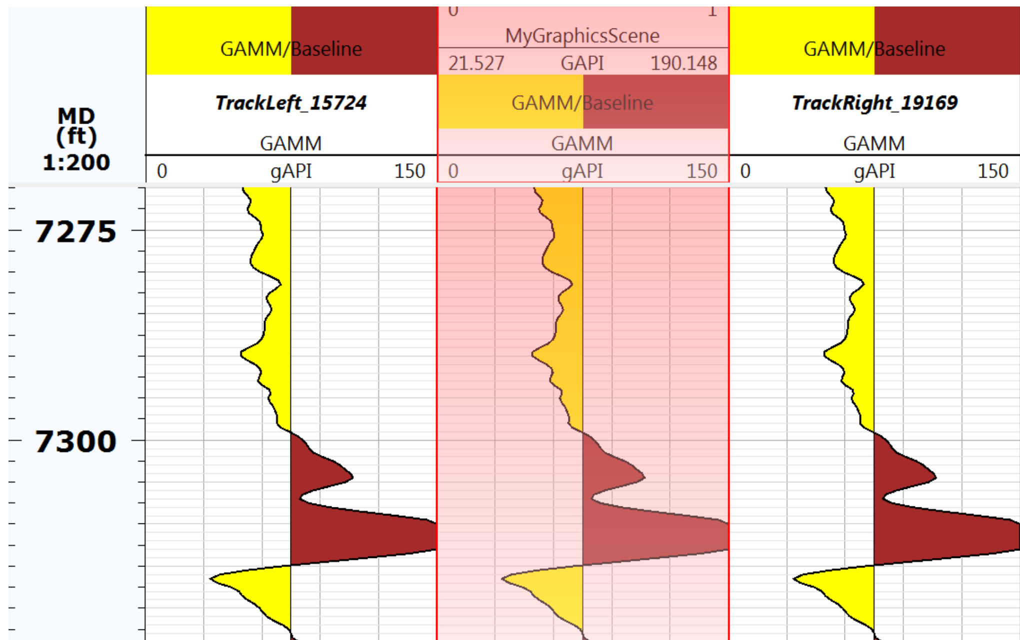


Figure 85 Track duplication on Key pressed event

The **`PolygonSelectionChanged`** signal includes an argument that gives the **`GraphicsItem`** domain objects that have been added or removed from **`GraphicsScene::items()`** domain object collection and **`GraphicsItem`** domain objects that were selected before the current selection. This argument is modeled in Ocean through the **`PolygonSelectionChangedArgs`** class.

```
class PolygonSelectionChangedArgs : public
SignalArgsT<GraphicsScene>

{

public:
  DomainObjectCollection<GraphicsItem>
  alreadySelectedGraphicsItems() const;
```

```
  DomainObjectCollection<GraphicsItem>
  newSelectedGraphicsItems() const;

  DomainObjectCollection<GraphicsItem>
  unselectedGraphicsItems() const;
  …
};
```

Example:

```
void activity::onPolygonSelectionChanged(const
Slb::Ocean::Techlog::PolygonSelectionChangedArgs &args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  // collection of new graphics items selected in the
GraphcicsScene
  DomainObjectCollection<GraphicsItem> newSelectedGraphicsItems
  = args.newSelectedGraphicsItems();

  qWarning() << "new selected graphics item: ";
  foreach(GraphicsItem graphicsItem, newSelectedGraphicsItems)
  {
    if (graphicsItem.isA<Ellipse2d>())
    {
      Ellipse2d ellipse2d = graphicsItem.cast<Ellipse2d>();
      qWarning() << "\tEllipse2d with coordinates = " <<
      ellipse2d.center().x() << ", " << ellipse2d.center().y();
    }
    if (graphicsItem.isA<Rectangle2d>())
    {
      Rectangle2d rectangle2d =
      graphicsItem.cast<Rectangle2d>();
      qWarning() << "\Rectangle2d with coordinates = " <<
      rectangle2d.center().x() << ", " <<
      rectangle2d.center().y();
    }
    if (graphicsItem.isA<Line2d>())
    {
      Line2d line2d = graphicsItem.cast<Line2d>();
      qWarning() << "\Line2d with coordinates = P1(" <<
       line2d.p1().x() << ", " << line2d.p1().y()
       << ") P2(" << line2d.p2().x() << ", " <<
       line2d.p2().y() << ")";
     }
  }

  lock.release();
```

```
}
```

The slot receiver implemented above is triggered when the Techlog end-user select the graphics items in the **GraphicsScene** using the "Polygon selection for shapes" tool as shown in the screenshot below and the polygon is closed with ALT key + mouse click button.

Note: The "Polygon selection for shapes" tool is showing up in the Techlog mousebar only if an Ocean **GraphicsScene** is created into the plot. Take care to not mistake the "Polygon selection for shapes" tool for the "Polygon selection" tool used to select native Techlog objects in the plot and displayed lower down the Techlog mousebar with the same icon.

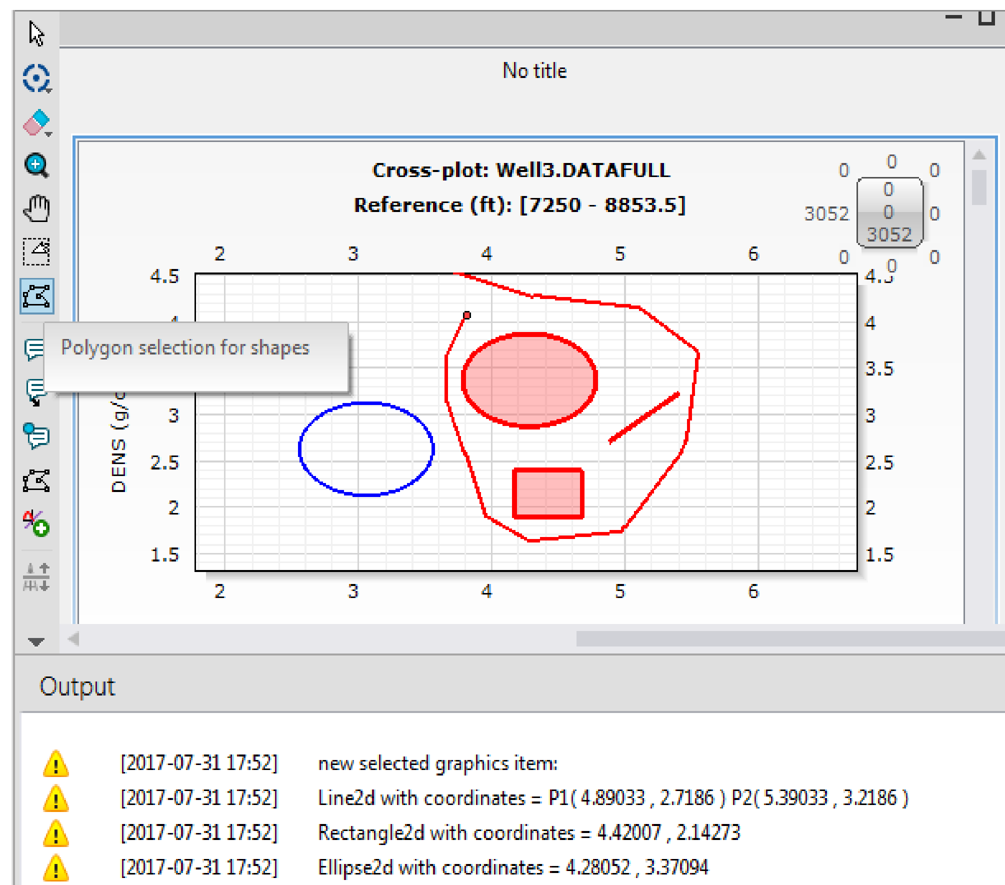


Figure 86 Polygon selection changed event

ChartParameter Domain Object

The **ChartParameter** class inherits from the **DomainObject** base class and does not support name. A **ChartParameter** object is instancied through **create** static method of the class with the parent **GraphicsScene** object as argument. The uniqueness of a **ChartParameter** object is handled by the system. The only way to retrieve a **ChartParameter** object from the **GraphicsScene** is by its droid.

```
class ChartParameter : public DomainObject
{
public:
```

```
  static ChartParameter create(const GraphicsScene
  &graphicsScene);
  …
  const double x() const;
  void setX(const double x);
  const double y() const;
  void setY(const double y);

  double xMin() const;
  double xMax() const;
  double yMin() const;
  double yMax() const;
  void setXRange(double xMin, double xMax);
  void setYRange(double yMin, double yMax);

  const Unit xUnit() const;
  void setXUnit(const Unit &xUnit);
  const Unit yUnit() const;
  void setYUnit(const Unit &yUnit);

  const QString markerLabel() const;
  void setMarkerLabel(const QString &markerLabel);
  const QColor markerColor() const;
  void setMarkerColor(const QColor &markerSize);
  const float markerSize() const;
  void setMarkerSize(const float markerSize);
  const QString description() const;
  void setDescription(const QString &description);
  const QString xyRelation() const;
  void setXyRelation(const QString &xyRelation);
};
```

You need to define X and Y position of the **`ChartParameter`** in the plot. Setting a min and max on X and Y axes implies that the user is only able to move the chart parameter on a limited range of values into the chart area.

If you want to see what is the current position of the chart parameter (X and Y coordinated in the chart area of the plot) in the properties pane of Techlog when the **`ChartParameter`** is selected then you need to set X and Y unit properties.

Some others optional properties allow to:

- get and set a text label for the **`ChartParameter`**, setting the **`markerLabel`** property makes that X and Y values of the **`ChartParameter`** position are automatically displayed in brackets close to the label (X and Y units are shown only if the **`xUnit`** and **`yUnit`** properties are set)
- get and set a color for the **`ChartParameter`**
- get and set a size for the **`ChartParameter`**
- get and set a description of what this **`ChartParameter`** is used for
- get and set the **`xyRelation`** property allows to define a parametric equation where y is a function of x
  - if this property is set the **`ChartParameter`** displacement is limited to the values defined by the parametric equation

Properties

Chart parameter

Property	Value
Name	{c2ad45ab-a2b9-41fc-99a...
Description	my chart parameter
Label	Move me again
Size	8
Color	
X position (%)	46.7697
Minimum value	0
Maximum value	60
Y position (mD)	189.338
Minimum value	0
Maximum value	10000
X and Y relationship	y = pow(10, log10(x)+(lo...

Figure 87 ChartParameter properties

A **ChartParameter** domain object can subscribe on some events.

```
class ChartParameter : public DomainObject
{
public:
  …
  enum EventType
  {
    ChartParameterChanged,
    ChartParameterValidated
  };
}
```

The signals are emitted whenever:

- **ChartParameterChanged** – chart parameter is moved in the chart area (mouse move event)
- **ChartParameterReleased** – chart parameter is released (mouse released event)

The following shows how to include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkchartparameterchangedargs.h"
#include "tsdkchartparametervalidatedargs.h"
```

```
private slots:
   void onChartParameterChanged(
   const Slb::Ocean::Techlog::ChartParameterChangedArgs &args);
   void onChartParameterValidated(
   const Slb::Ocean::Techlog::ChartParameterValidatedArgs
&args);
```

In the following example shows a **ChartParameter** domain object added to a **GraphicsScene** in a custom plot connects to the signals. **ChartParameter** is

SLB-Private

limited on X and Y axes by the boundaries of the plot and by a parametric equation defined through the **xyRelation** property.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

CustomPlot plot = CustomPlot::create(workspace);
plot.setHorizontalUserLimits(0, 60);
plot.setVerticalUserLimits(0, 10000);
plot.setYScaleType(PlotScaleTypeLogarithmic);
plot.setXUnit("%");
plot.setYUnit("mD");

GraphicsScene graphicsScene =
GraphicsScene::create("MyGraphicsScene", plot);

ChartParameter chartParameter =
ChartParameter::create(graphicsScene);

chartParameter.setX(1);
chartParameter.setXRange(plot.horizontalUserLowerLimit(),
plot.horizontalUserUpperLimit());
chartParameter.setXUnit("%");
chartParameter.setY(0.5);
chartParameter.setYRange(plot.verticalUserLowerLimit(),
plot.verticalUserUpperLimit());
chartParameter.setYUnit("mD");
chartParameter.setXyRelation("y = pow(10, log10(x)+(log10(5) -
log10(2.665))/0.45)");

chartParameter.setMarkerLabel("move me");
chartParameter.setMarkerColor(Qt::black);
chartParameter.setMarkerSize(8.0F);
chartParameter.setDescription("my chart parameter");

chartParameter.connect(ChartParameter::ChartParameterChanged,
this, SLOT(onChartParameterChanged(const
Slb::Ocean::Techlog::ChartParameterChangedArgs&)));
chartParameter.connect(ChartParameter::ChartParameterValidated
, this, SLOT(onChartParameterValidated(const
Slb::Ocean::Techlog::ChartParameterValidatedArgs&)));

lock.release();
```

The **ChartParameterChanged** and **ChartParameterValidated** signals include an argument that gives the **ChartParameter** instance the slots were connected to through the sender method inherited from the **SignalArgs** base class and the parent **Plot**. Those arguments are modeled respectively in Ocean through the **ChartParameterChangedArgs** and **ChartParameterValidatedArgs** classes.

```
class ChartParameterChangedArgs : public
SignalArgsT<ChartParameter>

{

public:

  Plot plot() const;

  …
};
```

```
class ChartParameterValidatedArgs : public
SignalArgsT<ChartParameter>

{

public:

  Plot plot() const;

  …
};
```

The following example shows a polyline drawn along the parametric equation defined through the **`xyRelation`** property of the **`ChartParameter`** domain object.

```
void activity::onChartParameterChanged(const
Slb::Ocean::Techlog::ChartParameterChangedArgs &args)
{
  ChartParameter chartParameter = args.sender();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  chartParameter);

  GraphicsScene graphicsScene = chartParameter.graphicsScene();

  lock.release();

  lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  graphicsScene);

  Polyline2d polyline;
  if (graphicsScene.polylines().count() != 0)
    polyline = graphicsScene.polylines().at(0);
  else
  {
    polyline = Polyline2d::create(graphicsScene);
    polyline.setPointsSize(0);
  }

  polyline.appendPoint(QPointF(chartParameter.x(),
  chartParameter.y()));
```

```
  lock.release();
}
```

In the slot receiver of **ChartParameterValidated** signal the **ChartParameter** label is changed.

```
void activity::onChartParameterValidated(const
Slb::Ocean::Techlog::ChartParameterValidatedArgs &args)
{
  ChartParameter chartParameter = args.sender();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  chartParameter);

  chartParameter.setMarkerLabel("Move me again");

  lock.release();
}
```

The following screenshot shows the display obtained when those events are triggered.

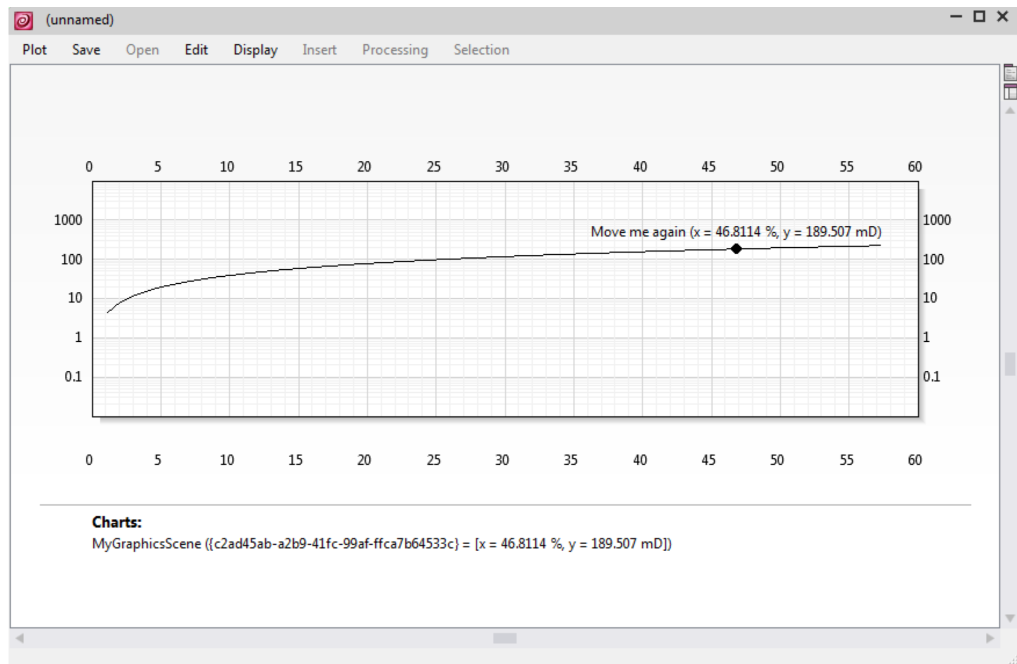


Figure 88 ChartParameter moved along a parametric equation

GraphicsItem Domain Objects

The **GraphicsItem** class is a base class that holds graphics-related fields common to all graphics item classes. Each **GraphicsItem** derived classes are instancied through **create** static method of the class passing the parent **GraphicsScene** object as argument.

GraphicsItem base class inherits from the **DomainObject** base class and all graphics classes derived from **GraphicsItem** does not support name. The uniqueness

of a **GraphicsItem** object is handled by the system. The only way to retrieve a **GraphicsItem** object from the **GraphicsScene** is by its droid.

```
class GraphicsItem : public DomainObject
{
public:
  const bool visible() const;
  void setVisible(const bool value);
  const bool selected() const;
  void setSelected(const bool value);
  const Unit xUnit() const;
  void setXUnit(const Unit &value);
  const Unit yUnit() const;
  void setYUnit(const Unit &value);
  const GraphicsScene graphicsScene() const;
  const bool selectable() const;
  void setSelectable(const bool selectable);
  const bool movable() const;
  void setMovable(const bool movable);
  …
};
```

Common properties holds by the class are:

- **visible** – hide / show the **GraphicsItem** in the **GraphicsScene**
- **selected** – unselect / select the **GraphicsItem** in the **GraphicsScene**
- **xUnit / yUnit** – **GraphicsItem** has its own units on X and Y axes
  - those properties can be set through the **setXUnit** and **setYunit** functions of parent **GraphicsScene**
- **selectable** – makes the **GraphicsItem** selectable in the **GraphicsScene**
  - this property has to be turned on in order to emit the **Selected** signal
- **movable** – makes the **GraphicsItem** movable in the **GraphicsScene**
  - this property has to be turned on in order to emit the **Moving** signal

Shape Domain Objects

The **Shape** class is derived from **GraphicsItem** base class and holds graphics-related fields common to shape classes.

```
class Shape : public GraphicsItem
{
public:
  const QColor backgroundColor() const;
  void setBackgroundColor(const QColor &value);
  const QColor borderColor() const;
  void setBorderColor(const QColor &value);
  const int strokeWidth() const;
  void setStrokeWidth(const int value);

  void setCursorShape(Qt::CursorShape cursorShape);

  Qt::CursorShape cursorShape();
  …
```

```
};
```

Common properties holds by the class are:

- **backgroundColor** – changes the background color of a shape
    - transparency can be applied through a value of QColor constructor QColor::QColor(int r, int g, int b, int a = 255)
- **borderColor** – changes the border color of a shape
- **strokeWidth** – changes the stroke width of a shape
- **cursorShape** – specify a different CursorShape when the mouse is over the shape

Ellipse2d Domain Object

Ellipse2d class inherits from **Shape** base class and allows you to draw ellipses into a **GraphicsScene**.

An **Ellipse2d** domain object is defined by a center point and radiuses on X and Y axes. The values are expressed in the unit of the variables plotted into the **Track** or **Plot**.

```
class Ellipse2d : public Shape
{
public:
  static Ellipse2d create(GraphicsScene graphicsScene);
  …
  void setCenter(const QPointF & value);
  void setHorizontalRadius(const float & value);
  void setVerticalRadius(const float & value);
  …
};
```

Below is an example:

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
graphicsScene);

Ellipse2d ellipse = Ellipse2d::create(graphicsScene);
ellipse.setCenter(QPointF(x, y));
ellipse.setHorizontalRadius(hRadius);
ellipse.setVerticalRadius(vRadius);

lock.release();
```

Line2d Domain Object

Line2d class inherits from **Shape** base class and allows you to draw to draw lines into a **GraphicsScene**.

A **Line2d** domain object is defined by two extremity points on X and Y axes. Those values are expressed in the unit of the variables plotted into the **Track** or **Plot**.

Others properties allow you to change the pen and extremity styles.

```
class Line2d : public Shape
{
public:
  static Line2d create(GraphicsScene graphicsScene);
  void setP1(const QPointF & value);
  void setP2(const QPointF & value)
  void setLineStyle(const Qt::PenStyle & value)
  void setP1ExtremityStyle(const LineExtremity & value);
  void setP2ExtremityStyle(const LineExtremity & value);
  …
};
```

Example:

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
graphicsScene);

Line2d line = Line2d::create(graphicsScene);
line.setP1(QPointF(x1, y1));
line.setP2(QPointF(x2, y2));
line.setLineStyle(Qt::SolidLine);

lock.release();
```

Rectangle2d Domain Object

`Rectangle2d` class inherits from **`Shape`** base class and allows you to draw rectangles into a **`GraphicsScene`**.

A **`Rectangle2d`** domain object is defined by a center point, the width and height of the rectangle on X and Y axes. The values are expressed in the unit of the variables plotted into the **`Track`** or **`Plot`**.

```
class Rectangle2d : public Shape
{
public:
  static Rectangle2d create(GraphicsScene graphicsScene);
  …
  void setCenter(const QPointF & centerPosition);
  void setSize(const QSizeF & value);
  …
};
```

Example:

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
graphicsScene);

Rectangle2d rectangle = Rectangle2d::create(graphicsScene);
rectangle.setCenter(QPointF(x, y));
rectangle.setSize(QSizeF(width, height));
```

```
lock.release();
```

Image2d Domain Object

`Image2d` class inherits from **`Rectangle2D`** class and allows you to display an image in a **`GraphicsScene`** with **`center`** and **`size`** set through properties of the parent class. The image must be stored in a file, and the format must be known by Qt standard loaders. The absolute path to this file is passed through **`setImageFile`** public method. An image has an opacity property which enables it to be transparent.

```
class Image2d : public Rectangle2d
{
public:
  static Image2d create(GraphicsScene graphicsScene);
  …
  const QString imageFile() const;
  void setImageFile(const QString &imageFile);

  const unsigned int imageOpacity() const;
  void setImageOpacity(const unsigned int imageOpacity);
  …
};
```

The following example shows an image file stored in the plug-in folder into a cross-plot.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Dataset dataset =
DomainObject::get(datasetDroid).cast<Dataset>();

Variable neutron = dataset.variables().get("NEUT");
Variable density = dataset.variables().get("DENS");
Variable gammaRay = dataset.variables().get("GAMM");

CrossPlot crossplot = CrossPlot::create(workspace);
crossplot.setWindowTitle("My Cross-Plot");
PlotScale scale = crossplot.defaultPlotScale();
scale.addXAxisVariable(neutron);
scale.addYAxisVariable(density);
scale.setColor(gammaRay);

GraphicsScene graphicsScene = GraphicsScene::create("MyScene",
crossplot);

Image2d image = Image2d::create(graphicsScene);
```

```
QPointF center = QPointF((scale.horizontalVariableUpperLimit() -
scale.horizontalVariableLowerLimit()) / 2 +
scale.horizontalVariableLowerLimit(),
(scale.verticalVariableUpperLimit() -
scale.verticalVariableLowerLimit()) / 2 +
scale.verticalVariableLowerLimit());
image.setCenter(center);

QSizeF size = QSizeF(scale.horizontalVariableUpperLimit() -
scale.horizontalVariableLowerLimit(),
scale.verticalVariableUpperLimit() -
scale.verticalVariableLowerLimit());
image.setSize(size);

image.setImageFile(Session::pluginDirectory() + "/" +
"ocean.png");image.setImageOpacity(5);

lock.release();
```

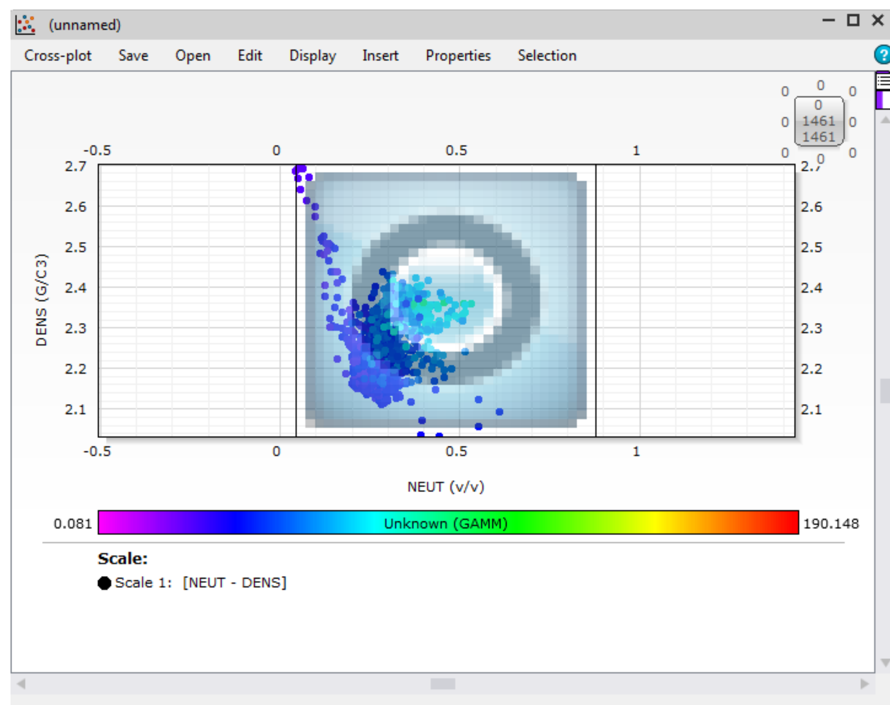


Figure 89 Display an image into a plot

Triangle2d Domain Object

Triangle2d class inherits from **Shape** base class and allows you to draw triangles into a **GraphicsScene**.

A **Triangle2d** domain object is defined by three points on X and Y axes. The values are expressed in the unit of the variables plotted into the **Track** or **Plot**.

```
class Triangle2d : public Shape
{
```

```
public:
  static Triangle2d create(GraphicsScene graphicsScene);
  …
  void setP1(const QPointF & value);
  void setP2(const QPointF & value);
  void setP3(const QPointF & value);
  …
};
```

Example:

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
graphicsScene);

Triangle2d triangle = Triangle2d::create(graphicsScene);
triangle.setP1(QPointF(x1, y1));
triangle.setP2(QPointF(x2, y2));
triangle.setP3(QPointF(x3, y3));

lock.release();
```

PlotAnnotation2d Domain Object

The **`PlotAnnotation2d`** domain object is used to add text notes in plots. A **`PlotAnnotation2d`** domain object belongs to a **`Plot`** domain object which is the argument of **`create`** static method.

```
class PlotAnnotation2d : public Annotation2d
{
public:
  static PlotAnnotation2d create(GraphicsScene graphicsScene);

  Plot plot() const;
  const float rotation() const;
  void setRotation(const float value);
  const QPointF position() const;
  void setPosition(const QPointF &value);
  const QPointF size() const;
  void setSize(const QPointF &value);
  const PlotAnnotation2dType type() const;
  void setType(const PlotAnnotation2dType value);
  …
};
```

The **`PlotAnnotation2d`** class inherits from the **`Annotation2d`** base class and does not support name.

```
class Annotation2d : public Shape
{
public:
  const QString htmlText() const;
  void setHtmlText(const QString &value);
  const QString plainText() const;
  void setPlainText(const QString &value);
```

```
  const Qt::Alignment horizontalAlignment() const;
  void setHorizontalAlignment(const Qt::Alignment value);
  const QFont textFont() const;
  void setTextFont(const QFont &value);
  const QColor textColor() const;
  void setTextColor(const QColor &value);
  const QColor textHighlightColor() const;
  void setTextHighlightColor(const QColor &value);
  const Annotation2dVerticalAlignment verticalAlignment()
  const;
  void setVerticalAlignment(const Annotation2VerticalAlignment
  value);
  const Annotation2dOrientation orientation() const;
  void setOrientation(const Annotation2Orientation value);
   …
};
```

The uniqueness of a **PlotAnnotation2d** object is handled by the system. The only way to retrieve a **PlotAnnotation2** object from the **Plot** is by its droid.

Plot class provides a method to retrieve the collection of **PlotAnnotation2d** domain objects.

```
class Plot : public DomainObject
{
public:

  DomainObjectCollection<PlotAnnotation2d> annotations() const;

  …
};
```

PointsList Domain Objects

The **PointsList** class is a base class that holds graphics-related fields and accessors common to list of point classes. Each **PointsList** derived classes are instancied through **create** static method of the derived class with the parent **GraphicsScene** object as argument.

The **PointsList** base class inherits from the **GraphicsItem** base class and does not support name. The uniqueness of a **PointsList** object is handled by the system. The only way to retrieve a **PointsList** object from the **GraphicsScene** is by its droid.

```
class PointsList : public GraphicsItem
{
public:
  void appendPoint(const QPointF & point);
  void insertPoint(const QPointF & point, const unsigned int
  index);
  void replacePoint(const QPointF & point, const unsigned int
  index);
  void appendPoints(const QList<QPointF> & points);
  …
  const QStringList labels() const;
  void setLabels(const QStringList & labels);
   const float pointsSize() const;
```

```
  void setPointsSize(const float & pointsSize);
  const QColor labelsColor() const;
  void setPointsColor(const QColor & pointsColor);
  …
};
```

PointsCloud Domain Object

`PointsCloud` class inherits from **`PointsList`** base class and allows you to draw points cloud into a **`GraphicsScene`**.

The class only holds **`create`** static method used to instanciate the domain object.

```
class PointsCloud : public PointsList
{
public:
  static PointsCloud create(GraphicsScene graphicsScene);
  …
};
```

Polyline2d Domain Object

`Polyline2d` class inherits from **`PointsList`** base class and allows you to draw polylines into a **`GraphicsScene`**. **`Poyline2d`** holds additional methods for line attributes.

```
class Polyline2d : public PointsList
{
public:
  static Polyline2d create(GraphicsScene graphicsScene);

  void setLineStyle(const Qt::PenStyle &value);
  const Qt::PenStyle lineStyle() const;
  void setLineThickness(const int &value);
  const int lineThickness() const;
  void setLineColor(const QColor &value);
  const QColor lineColor() const;
  void setClosed(const bool &value);
  const bool closed() const;
  …
};
```

Below is an example creating a **`Polyline2d`** into the **`GraphicsScene`** of a **`CrossPlot`** and populating the polyline with point values where y = x + 2.

```
Lock lock = LOCK_CREATE();
lock.add(crossplotDroid);
lock.add(dataset);
LOCK_ACQUIRE_OR_RETURN(lock);

CrossPlot crossPlot =
DomainObject::get(crossplotDroid).cast<CrossPlot>();

// Create a graphics scene
```

```
GraphicsScene graphicsScene = GraphicsScene::create("MyScene",
crossplot);

QList<QPointF> pointsPolyline;
// Create a polyline into the graphics scene with equation y = x
+ 2
quint64 rowCount = dataset.rowCount();
for (int i = 0; i < rowCount; i++)
{
  float x = neutron.getFloatValue(i);
  float y = density.getFloatValue(i);
  if (x == Absent::MissingValue || y == Absent::MissingValue)
continue;
  pointsPolyline.append((QPointF(x,x+2)));
}

Polyline2d polyline = Polyline2d::create(graphicsScene);
// add list of points to the polyline
polyline.appendPoints(pointsPolyline);
// set polyline points shape, size and color properties
polyline.setPointsType(PointTypeEmptyTriangle);
polyline.setPointsSize(8);
polyline.setPointsColor(Qt::darkGreen);

lock.release();
```

MarkerLabel Domain Object

The **MarkerLabel** class inherits from the **GraphicsItem** base class and does not support name. A **MarkerLabel** object is instancied through **create** static method of the class with the parent **GraphicsScene** object as argument. The uniqueness of a **MarkerLabel** object is handled by the system. The only way to retrieve a **MarkerLabel** object from the **GraphicsScene** is by its droid.

MarkerLabel allows you to show some plain text into the chart area of any plots.

A **MarkerLabel** domain object is defined by a center point on X and Y axes. Those values are expressed in the unit of the variables plotted into the **Track** or **Plot**.

The color, font and orientation of the text can be changed through corresponding properties of the class.

```
class MarkerLabel : public GraphicsItem
{
public:
  static MarkerLabel create(GraphicsScene graphicsScene);

  const QPointF center() const;
  void setCenter(const QPointF &value);
  const QString text() const;
  void setText(const QString &text);
```

```
  const QColor markerColor() const;
  void setMarkerColor(const QColor &markerColor);
  const QColor textColor() const;
  void setTextColor(const QColor &textColor);
  const QFont font() const;
  void setFont(const QFont &textColor);
  const float rotation() const;
  void setRotation(const float rotation);
  …
};
```

The following example shows a **`MarkerLabel`** created at the center point of a linear regression calculated from X and Y variables values plotted in a standard crossplot. The **`MarkerLabel`** is rotated following the linear regression slope.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

CrossPlot crossplot =
DomainObject::get(crossplotDroid).cast<CrossPlot>();

PlotScale scale = crossplot.defaultPlotScale();

Variable xVar = scale.xAxisVariables().get("NEUT");
Variable yVar = scale.yAxisVariables().get("DENS");
quint64 rowCount = xVar.dataset().rowCount();

LinearRegression linearRegression;
// Compute linear regression
for (int i = 0; i < rowCount; i++)
{
  if ((xVar.getFloatValue(i) != Absent::MissingValue) &&
  (yVar.getFloatValue(i) != Absent::MissingValue))
  {
    linearRegression.addXY((double)xVar.getFloatValue(i),
   (double)yVar.getFloatValue(i));
  }
}

qWarning() << "Linear regression equation is  y = " <<
linearRegression.getA() << "x + " << linearRegression.getB();

// Create a graphics scene
GraphicsScene graphicsScene = GraphicsScene::create("MyScene",
crossplot);
QList<QPointF> pointsPolyline;
// Show linear regression with equation y = a*x + 2
for (int i = 0; i < rowCount; i++)
{
  float x = xVar.getFloatValue(i);
  float y = yVar.getFloatValue(i);
```

```
  if (x == Absent::MissingValue || y == Absent::MissingValue)
    continue;
  pointsPolyline.append(
  (QPointF(x,linearRegression.estimateY(x))));
}

Polyline2d polyline = Polyline2d::create(graphicsScene);
// add list of points to the polyline
polyline.appendPoints(pointsPolyline);
polyline.setLineThickness(2);
polyline.setPointsSize(0);

QPointF center =
polyline.points().at((int)(polyline.points().count() / 2));

MarkerLabel markerLabel = MarkerLabel::create(graphicsScene);
markerLabel.setCenter(center);
// rotate MarkerLabel following linear regression slope
markerLabel.setRotation((float)linearRegression.getA());
QFont qf = QFont("Comic Sans MS", 12, QFont::Bold);
markerLabel.setText("Linear regression slope");
markerLabel.setFont(qf);
markerLabel.setTextColor(Qt::darkBlue);
markerLabel.setSize(5);

lock.release();
```

The screenshot below shows the **MarkerLabel** at the center point of the linear regression and following the slope.

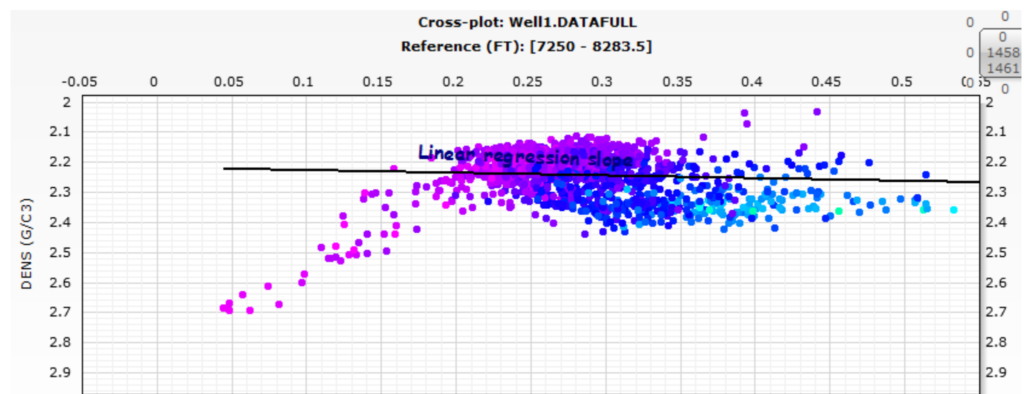


Figure 90 MarkerLabel color, font and rotation

GraphicsItem signals

All classes described previously in “Shape Domain Objects”, “PointsList Domain Objects” and “MarkerLabel Domain Object” chapters are derived classes from main **GraphicsItem** base class.

All domain objects derived from **GraphicsItem** base class can subscribe on some events.

```
class GraphicsItem : public DomainObject
{
public:
  …
  const bool selectable() const;
  void setSelectable(const bool selectable);
  const bool movable() const;
  void setMovable(const bool movable);

  enum EventType
  {
    Selected,
    Unselected,
    Moving,
    Released
  };
};
```

The signals are emitted whenever:

- **Selected** – the shape is selected (mouse click event)
- **Unselected** – the shape is unselected and has lost the focus (mouse click outside of the shape area)
- **Moving** – the shape is moved over the chart area
- **Released** – the shape is released (mouse release event)

Note: **Selected**, **released** and **Unselected** signals are emitted only when **selectable** property is turned on. **Moving** signal is emitted only when both **selectable** and **movable** properties are turned on.

The following shows how to include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkgraphicsiteminteractionargs.h"
```

```
private slots:
  void onGraphicsItemSelected(
  const Slb::Ocean::Techlog::GraphicsItemInteractionArgs
&args);
  void onGraphicsItemUnselected(
  const Slb::Ocean::Techlog::GraphicsItemInteractionArgs
&args);
  void onGraphicsItemMoving(
  const Slb::Ocean::Techlog::GraphicsItemInteractionArgs
&args);
  void onGraphicsItemReleased(
  const Slb::Ocean::Techlog::GraphicsItemInteractionArgs
&args);
```

The following example shows how to connect the **MarkerLabel** domain object to those signals previously defined in "MarkerLabel Domain Object" section.

```
lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, markerLabel);

markerLabel.setSelectable(true);
markerLabel.setMovable(true);

markerLabel.connect(GraphicsItem::Selected, this,
SLOT(onGraphicsItemSelected(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs&)));
markerLabel.connect(GraphicsItem::Unselected, this,
SLOT(onGraphicsItemUnselected(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs&)));
markerLabel.connect(GraphicsItem::Moving, this,
SLOT(onGraphicsItemMoving(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs&)));
markerLabel.connect(GraphicsItem::Released, this,
SLOT(onGraphicsItemReleased(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs&)));

lock.release();
```

`GraphicsItem` signals include an argument that gives the **`GraphicsItem`** instance the slots were connected to through the sender method inherited from the **`SignalArgs`** base class. The **`GraphicsItem`** instance returned by **`sender`** method can be cast to the derived object. This argument is modeled in Ocean through the **`GraphicsItemInteractionArgs`** class.

```
class GraphicsItemInteractionArgs : public
SignalArgsT<ShapeGraphicsItem>
{
};
```

When the **`Selected`** signal is emitted **`MarkerLabel`** color is changed to red.

```
void activity::onGraphicsItemSelected(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs &args)
{
  MarkerLabel markerLabel = args.sender().cast<MarkerLabel>();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  markerLabel);

  markerLabel.setTextColor(Qt::red);

  lock.release();
}
```

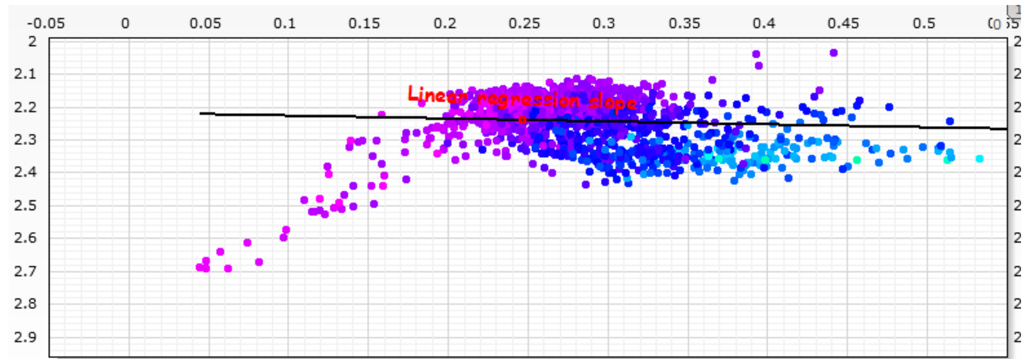


Figure 91 GraphicsItem Selected signal emitted

When the **`Moving`** signal is emitted **`MarkerLabel`** text property is changed to display X and Y position of the **`Shape`**.

```
void activity::onGraphicsItemMoving(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs &args)
{
  MarkerLabel markerLabel = args.sender().cast<MarkerLabel>();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  markerLabel);

  QString textLabel = QString("Moving  X = %1, Y = %2")
  .arg(markerLabel.center().x())
  .arg(markerLabel.center().y());

  markerLabel.setText(textLabel);

  lock.release();
}
```

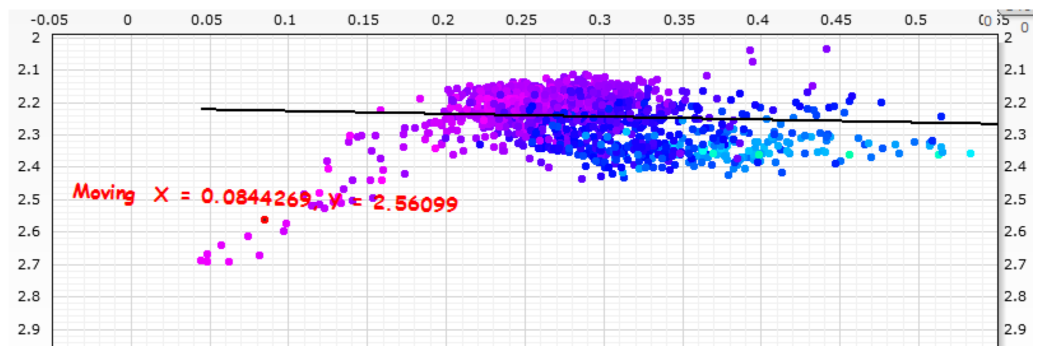


Figure 92 GraphicsItem Moving signal emitted

When the **`Unselected`** signal is emitted **`MarkerLabel`** color and text properties are changed to original values and **`MarkerLabel`** position is recalculated following the linear regression equation.

SLB-Private

```
void activity::onGraphicsItemUnselected(const
Slb::Ocean::Techlog::GraphicsItemInteractionArgs &args)
{
  MarkerLabel markerLabel = args.sender().cast<MarkerLabel>();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  markerLabel);

  markerLabel.setTextColor(Qt::darkBlue);

  QPointF pointOnLinearRegression =
  QPointF(markerLabel.center().x(),
  linearRegression.estimateY(markerLabel.center().x()));

  markerLabel.setCenter(pointOnLinearRegression);

  markerLabel.setText("Linear regression slope");

  lock.release();
}
```

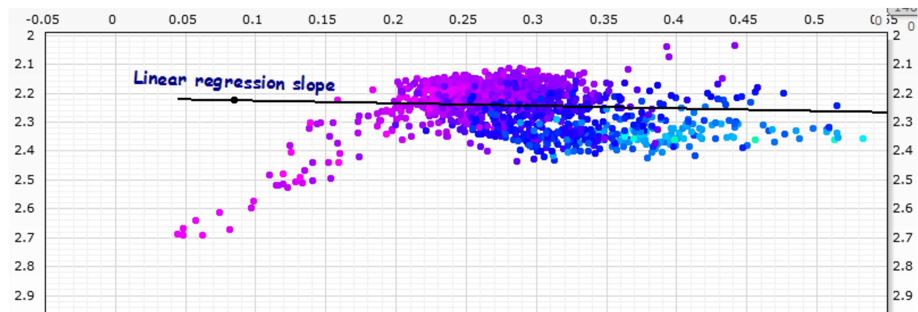


Figure 93 GraphicsItem Unselected signal emitted

Action Domain Object

Ability to extend mouse bar, tool bar and context menu of any plots derived from **`Plot`** base class is available in Ocean through **`Action`** class.

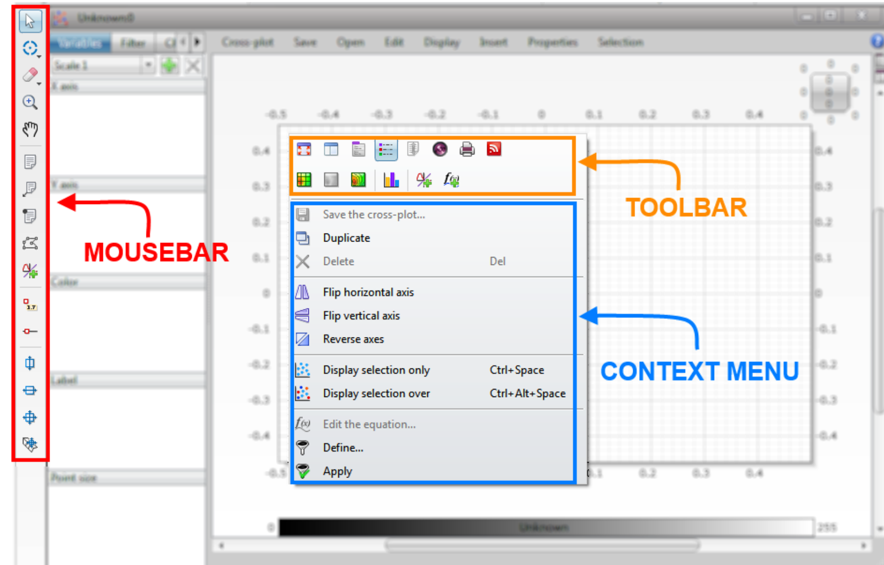


Figure 94 Actions in a Plot

Action domain object belongs to a **Plot**. An **Action** domain object can be enumerated from a **Plot** instance through **actions** domain object collection.

```
class Plot : public DomainObject
{
public:
  const DomainObjectCollection<Action> actions() const;
  …
};
```

The **Action** class inherits from the **DomainObject** base class and does not support name. The uniqueness of an **Action** object is handled by the system. The only way to retrieve an **Action** object from the parent **Plot** is by its droid.

An **Action** object is instancied through **create** static method of the class passing as arguments the text of the action (displayed as tooltip for mouse bar and tool bar actions and as text item for context menu action), location of the action (mouse bar, tool bar or context menu), the type of action (click or toggle button) and the parent **Plot** instance.

```
class Action : public DomainObject
{
public:
  static Action create(const QString &text,  ActionType
  actionType, ActionLocations locations, Plot plot);
  …
};
```

ActionType enum class has the following values:

```
enum ActionType
{
  ActionTypeButton,
  ActionTypeToggleButton,
  ActionTypeToggleGroup
```

```
};
```

An **Action** domain object created with an **ActionType** equals to **ActionTypeToggleGroup** is added to the group of native and custom toggle buttons existing in the mouse bar, tool bar or context menu (depending of **ActionLocation** enum value) of the **Plot**. This makes that only one toggle button is active at the time in the mouse bar, tool bar or context menu.

ActionLocation enum class has the following values:

```
enum ActionLocation
{
  ActionLocationToolBar,
  ActionLocationMouseBar,
  ActionLocationContextMenu
};
```

You can create the action in different locations combining **ActionLocation** enum values with the bitwise inclusive OR "|" as shown in the example below:

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
crossplotDroid);

CrossPlot crossplot =
DomainObject::get(crossplotDroid).cast<CrossPlot>();

Action action = Action::create("I am everywhere",
ActionTypeButton,
ActionLocationMouseBar|ActionLocationToolBar|ActionLocationCon
textMenu, crossplot);

lock.release();
```

In the following screenshot the **Action** is added to mouse bar, tool bar and context menu of the cross-plot.

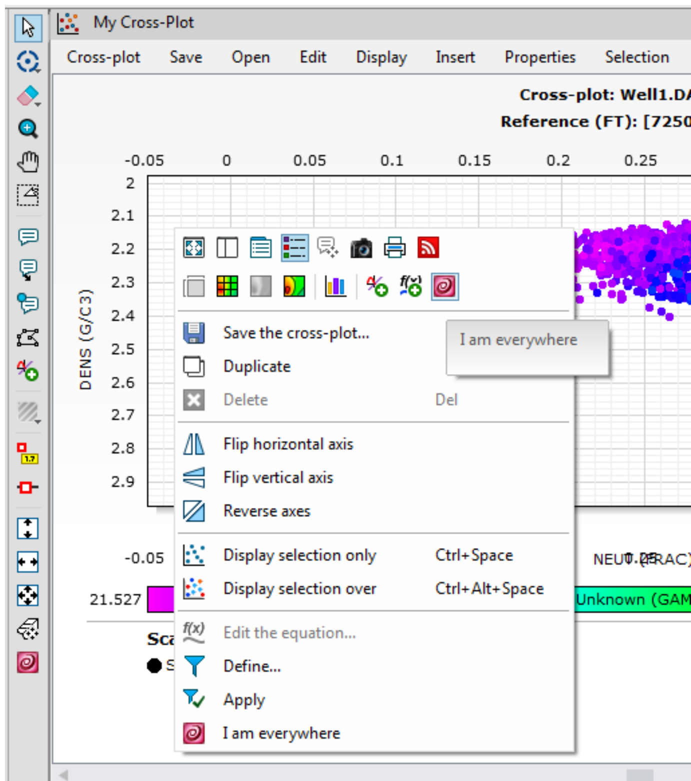


Figure 95 Action added to mouse bar, tool bar and context menu of the cross-plot

```
class Action : public DomainObject
{
public:
  …
  bool isCheckable() const;
  bool isChecked() const;
  void setChecked(bool checked);

  bool isEnabled() const
  void setEnabled(bool enabled);

  bool isVisible() const;
  void setVisible(bool visible);

  QString text() const;
  void setText(const QString &text);

  QString toolTip() const;
  void setToolTip(const QString &toolTip);

  QIcon icon() const;
  void setIcon(const QIcon &icon);
};
```

The properties of **`Action`** class allow you to:

- get if the **Action** is a toggle button and can be checked through **isCheckable** property
- check or uncheck a toggle button
  - **isChecked** property returns false for an **ActionTypeButton**
  - only **ActionTypeToggleButton** and **ActionTypeToggleGroup** can be checked through **setChecked** method
  - calling **setChecked** method on an **ActionTypeButton** throw an exception
- enable or disable the **Action** (button is greyed out)
- hide or show the **Action**
- get and set the **text** property value
  - if no **toolTip** is defined for the **Action** (**setToolTip**) then **text** property value is used by default to display tooltip over the **Action**
- get and set the **toolTip** property value overidding the tooltip defined through **text** property value
  - **Action** located in the context menu does not show any tooltip
- get and set an **icon** for the **Action** (default icon is Techlog icon)
  - icon files can be stored in the png folder of Techlog baseline or in the plug-in folder

An **Action** domain object can subscribe on some events.

```
class Action : public DomainObject
{
public:
 …
 enum EventType
 {
   ActionHovered,
   ActionToggled,
   ActionTriggered
 };
}
```

The signals are emitted whenever:

- **ActionHovered** – mouse hovers over the **Action** button (mouse over event)
- **ActionToggled** – **isChecked** property value of a checkable **Action** (**ActionTypeToggleButton** or **ActionTypeToggleGroup**) has changed
- **ActionTriggered** – **Action** is clicked by the user (mouse click event)

The following explains how to include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkactionhoveredargs.h"
#include "tsdkactiontoggledargs.h"
#include "tsdkactiontriggeredargs.h"
private slots:
```

```
void onActionHovered(
const Slb::Ocean::Techlog::ActionHoveredArgs &args);
void onActionToggled(
const Slb::Ocean::Techlog::ActionToggledArgs &args);
void onActionTriggered(
const Slb::Ocean::Techlog::ActionTriggeredArgs &args);
```

An **`Action`** domain object with **`ActionType`** set to button or toggle button can connect to all of these signals. But it makes sense to connect an **`ActionTypeButton`** to **`ActionTriggered`** signal and an **`ActionTypeToggleButton`** or **`ActionTypeToggleGroup`** to the **`ActionToggled`** signal.

Note: **`ActionToggled`** signal can be emitted programmatically if the **`isChecked`** property value of the toggle button has changed.

The following example explains where two actions are created one as simple click button and the other one as toggle button. Both actions connect to **`ActionHovered`** and **`ActionTriggered`** signals. Toggle **`Action`** connects to **`ActionToggled`** signal.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
crossplotDroid);

CrossPlot crossplot =
DomainObject::get(crossplotDroid).cast<CrossPlot>();

Action actionButton = Action::create("I am a push button",
ActionTypeButton, ActionLocationMouseBar, crossplot);

actionButton.connect(Action::ActionHovered, this,
SLOT(onActionHovered(const
Slb::Ocean::Techlog::ActionHoveredArgs&)));
actionButton.connect(Action::ActionTriggered, this,
SLOT(onActionTriggered(const
Slb::Ocean::Techlog::ActionTriggeredArgs&)));

Action actionToggle = Action::create("I am a toggle button",
ActionTypeToggleButton, ActionLocationMouseBar, crossplot);

actionToggle.connect(Action::ActionHovered, this,
SLOT(onActionHovered(const
Slb::Ocean::Techlog::ActionHoveredArgs&)));
actionToggle.connect(Action::ActionTriggered, this,
SLOT(onActionTriggered(const
Slb::Ocean::Techlog::ActionTriggeredArgs&)));
actionToggle.connect(Action::ActionToggled, this,
SLOT(onActionToggled(const
Slb::Ocean::Techlog::ActionToggledArgs&)));

lock.release();
```

`ActionHovered` and **`ActionTriggered`** signals include arguments that give the **`Action`** instance the slots were connected to through the **`sender`** method inherited

from the **SignalArgs** base class. The arguments are modeled respectively in Ocean through **ActionHoveredArgs** and **ActionTriggeredArgs** classes.

```
class ActionHoveredArgs : public SignalArgsT<Action>
{
};
```

```
class ActionTriggeredArgs : public SignalArgsT<Action>
{
};
```

In the following example when **ActionHovered** or **ActionTriggered** signals are emitted we can retrieve the sender **Action** domain object and know if signals have been emitted by the **ActionTypeButton** or by the **ActionTypeToggleButton** through the **isCheckable** property.

```
void activity::onActionHovered(const
Slb::Ocean::Techlog::ActionHoveredArgs &args)
{
  Action action = args.sender();
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, action);
  if (action.isCheckable())
    qWarning() << "Action is a toggle button";
  else
    qWarning() << "Action is a push button";
  lock.release();
}

void activity::onActionTriggered(const
Slb::Ocean::Techlog::ActionTriggeredArgs &args)
{
  Action action = args.sender();
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, action);
  if (action.isCheckable())
    qWarning() << "Action is a toggle button";
  else
    qWarning() << "Action is a push button";
  lock.release();
}
```

ActionToggled signal includes an argument that gives the **Action** instance that the slots were connected to through the **sender** method inherited from the **SignalArgs** base class and for a toggle button (**ActionTypeToggleButton** or **ActionTypeToggleGroup**) if its state is checked or unchecked.

This argument is modeled in Ocean through **ActionToggledArgs** class.

```
class ActionToggledArgs : public SignalArgsT<Action>
{
public:
```

```
  bool isChecked() const;
};
```

Below is an example.

```
void activity::onActionToggled(const
Slb::Ocean::Techlog::ActionToggledArgs &args)
{
  Action action = args.sender();
  qWarning() << "Action toggle button state is " <<
args.isChecked();
}
```

Custom plots

What is not possible to do with Techlog native plots available in Ocean can be achieved in a customplot. This is available in Ocean through the **`CustomPlot`** domain object.

In a custom-plot you are not able to take advantage of any Techlog native plots functionalities and you must handle the drawing in the chart area by yourself through custom graphics.

You have also the ability to add custom UI's in custom tabs located in the left pane of the custom-plot.

See the "Customize plots" section on page 140 for more information on how to create graphics objects in a plot.

The following screenshot shows "Cipher" plug-in developed with Ocean for Techlog by HFE Schlumberger group and using **`CustomPlot`** domain object.

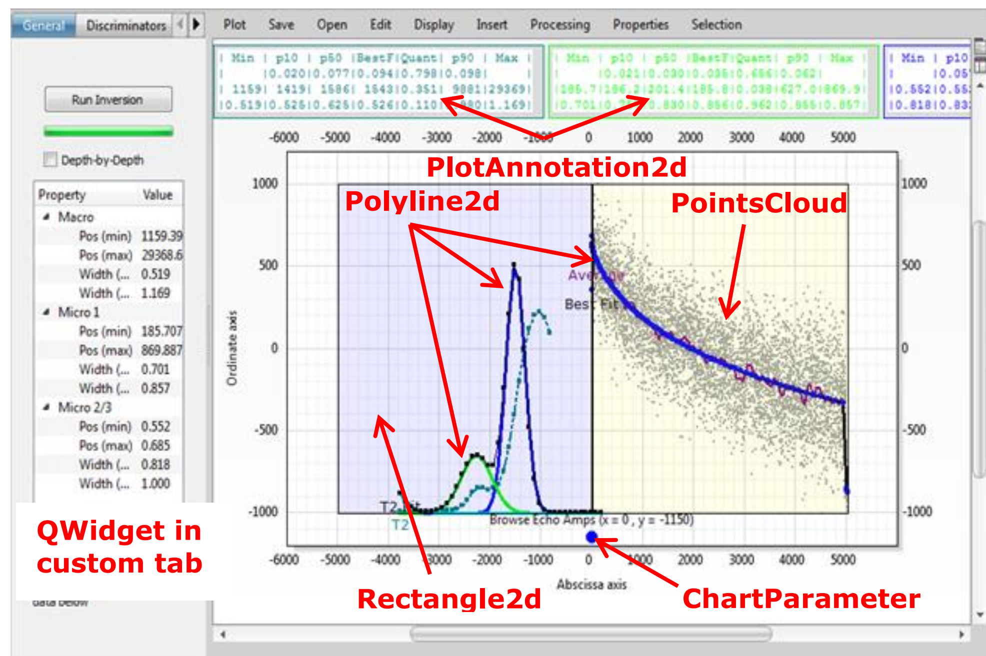


Figure 96 CustomPlot with custom graphics objects

The second type of plot fully customizable in Ocean is available through the **`ContainerPlot`** domain object. This plot allows you to group several plots in one **`Plot`**. You can group any Techlog native plots available in Ocean and **`CustomPlot`** as well. Like the **`CustomPlot`** you have the ability to add custom UI's in custom tabs located in the left pane of the **`ContainerPlot`**.

The followig screenshot shows an example of ContainerPlot containing native Techlog plots.

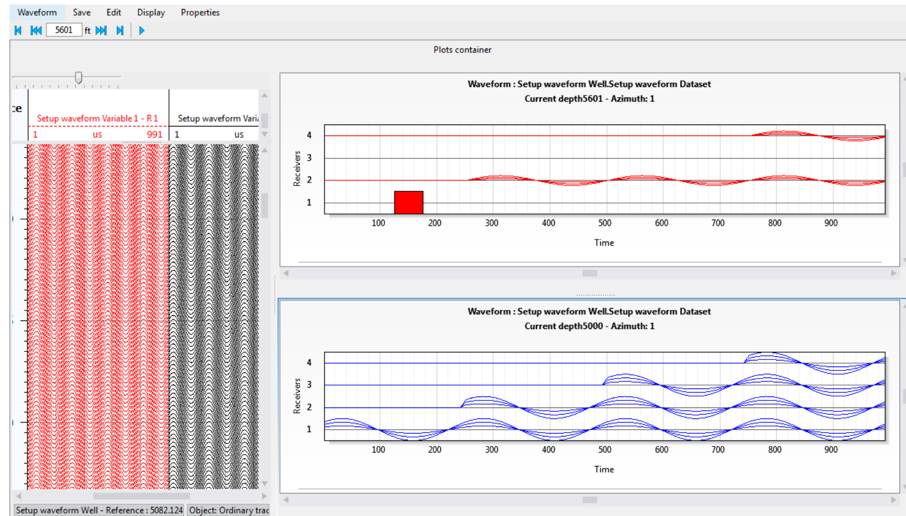


Figure 97 ContainerPlot grouping some native Techlog plots

CustomPlot Domain Object

The **`CustomPlot`** class inherits from the **`Plot`** class and does not support name. A **`CustomPlot`** object is instancied through **`create`** static method of the class with the parent **`Workspace`** object as argument. The uniqueness of a **`CustomPlot`** object is handled by the system. The only way to retrieve a **`CustomPlot`** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

A **`CustomPlot`** can also be added to a container plot (matrix-plot) using the dedicated **`create`** static methods.

See the "ContainerPlot Domain Object" section on page 189 for more information on how to create a custom-plot into a container plot.

```
class CustomPlot : public Plot
{
public:
  static CustomPlot create(Workspace workspace);

  static CustomPlot create(const ContainerPlotPosition
    &containerPlotPosition);

  void addCustomTab(QWidget *widget, const QString &tabName);
  …
};
```

You can add a custom UI in a custom tab of the **`CustomPlot`** passing a **`QWidget`** pointer to the **`addCustomTab`** function and the name of the new tab.

The following example shows how to build an UI with some Qt widgets and connect those widgets to signals.

Four QComboxes are declared in the header file in order to list:

- well names in the project
- dataset names for the well selected
- variable names with "Neutron Porosity" family for the dataset selected

SLB-Private

- variable names with ”Density” family for the dataset selected

Methods to populate the combo boxes and the slot receivers of combo boxes index changed signals are also declared in the header file.

```
#include "tsdkassignmenttypeenum.h"
#include "tsdktrackitemenums.h"
```

```
public:
  CustomPlot _customPlot;
  QPointer<QComboBox> _wellComboBox;
  QPointer<QComboBox> _datasetComboBox;
  QPointer<QComboBox> _porosityComboBox;
  QPointer<QComboBox> _densityComboBox;

  QStringList listDatasetsByWell(Well well);
  QStringList listVariablesByDataset(Dataset dataset, QString
familyName);
  void computePorosityVsDensity(Variable porosity, Variable
density);
```

```
private slots:
  void onWellChanged(const QString &arg);
  void onDatasetChanged(const QString &arg);
  void onVariableChanged(const QString &arg);
  void onCustomPlotClosed(
  const Slb::Ocean::Techlog::DomainObjectErasedArgs &args);
```

Source file contains the code below:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();
Project project = Session::current().mainProject();

_customPlot = CustomPlot::create(workspace);
_customPlot.setWindowTitle("My Custom Plot");

_customPlot.connect(CustomPlot::DomainObjectErased, this,
SLOT(onCustomPlotClosed(const
Slb::Ocean::Techlog::DomainObjectErasedArgs&)));



QLabel *wellLabel = new QLabel();
wellLabel->setText("Well");

_wellComboBox = new QComboBox();
foreach (const Well &well, project.wells())
  _wellComboBox->addItem(well.name());
```

```
QObject::connect(_wellComboBox,
SIGNAL(currentIndexChanged(const QString &)), this,
SLOT(onWellChanged(const QString &)));

QLabel *datasetLabel = new QLabel();
datasetLabel->setText("Dataset");
_datasetComboBox = new QComboBox();
_datasetComboBox->addItems(listDatasetsByWell(project.wells().
get(_wellComboBox->currentText())));

QObject::connect(_datasetComboBox,
SIGNAL(currentIndexChanged(const QString &)), this,
SLOT(onDatasetChanged(const QString &)));

QLabel *porosityLabel = new QLabel();
porosityLabel->setText("Porosity");

_porosityComboBox = new QComboBox();
_porosityComboBox->addItems(listVariablesByDataset(project.wel
ls().get(_wellComboBox->currentText()).datasets().get(_dataset
ComboBox->currentText()), "Neutron Porosity"));

QObject::connect(_porosityComboBox,
SIGNAL(currentIndexChanged(const QString &)), this,
SLOT(onVariableChanged(const QString &)));

QLabel *densityLabel = new QLabel();
densityLabel->setText("Density");

_densityComboBox = new QComboBox();
_densityComboBox->addItems(listVariablesByDataset(project.well
s().get(_wellComboBox->currentText()).datasets().get(_datasetC
omboBox->currentText()), "Bulk Density"));

QObject::connect(_densityComboBox,
SIGNAL(currentIndexChanged(const QString &)), this,
SLOT(onVariableChanged(const QString &)));

QHBoxLayout *wellLayout = new QHBoxLayout();
wellLayout->addWidget(wellLabel);
wellLayout->addWidget(_wellComboBox, 1);

QHBoxLayout *datasetLayout = new QHBoxLayout();
datasetLayout->addWidget(datasetLabel);
datasetLayout->addWidget(_datasetComboBox, 1);

QHBoxLayout *porosityLayout = new QHBoxLayout();
```

```
porosityLayout->addWidget(porosityLabel);
porosityLayout->addWidget(_porosityComboBox, 1);

QHBoxLayout *densityLayout = new QHBoxLayout();
densityLayout->addWidget(densityLabel);
densityLayout->addWidget(_densityComboBox, 1);

QVBoxLayout *layout = new QVBoxLayout();
layout->addLayout(wellLayout);
layout->addLayout(datasetLayout);
layout->addLayout(porosityLayout);
layout->addLayout(densityLayout);

QWidget* widget = new QWidget();
widget->setMaximumWidth(100);
widget->setLayout(layout);

_customPlot.addCustomTab(widget, "My Custom Tab");

if ((!_porosityComboBox->currentText().isEmpty()) &&
(!_densityComboBox->currentText().isEmpty()))
{

  Variable porosity =
project.wells().get(_wellComboBox->currentText()).datasets().g
et(_datasetComboBox->currentText()).variables().get(_porosityC
omboBox->currentText());

  Variable density =
project.wells().get(_wellComboBox->currentText()).datasets().g
et(_datasetComboBox->currentText()).variables().get(_densityCo
mboBox->currentText());

  computePorosityVsDensity(porosity, density);
}

lock.release();
```

Slots are implemented as follows:

- if the well name is changed the QComboBox containing the dataset name is recomputed
- if the dataset name is changed the QComboBoxes containing the porosity and density variables are recomputed
- if the variables selected in the porosity combo box and in the density combo box have changed a **`PointsCloud`** is calculated with X porosity values and Y density values to represent a cross-plot of Porosity against Density in the chart area of the **`CustomPlot`**

```
void activity::onCustomPlotClosed(const
Slb::Ocean::Techlog::DomainObjectErasedArgs &args)
{
  CustomPlot customPlot = args.sender().cast<CustomPlot>();
  qWarning() << "Customplot " << customPlot.windowTitle()
  << " closed";
  stop();
}

void activity::onWellChanged(const QString &arg)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  Project project = Session::current().mainProject();
  Well well = project.wells().get(arg);
  QStringList wells = listDatasetsByWell(well);
  lock.release();

  _datasetComboBox->clear();
  _datasetComboBox->addItems(wells);
}

void activity::onDatasetChanged(const QString &arg)
{
  if (arg.isEmpty()) return;

  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  Project project = Session::current().mainProject();
  Dataset dataset = project.wells()
  .get(_wellComboBox->currentText()).datasets().get(arg);

  QStringList porosities = listVariablesByDataset(dataset,
  "Neutron Porosity");
  QStringList densities = listVariablesByDataset(dataset,
  "Bulk Density");

  lock.release();

  _porosityComboBox->clear();
  _porosityComboBox->addItems(porosities);
  _densityComboBox->clear();
  _densityComboBox->addItems(densities);
}
void activity::onVariableChanged(const QString &arg)
{
  qWarning() << arg;
```

SLB-Private

```
  if (_porosityComboBox->currentText().isEmpty() ||
  _densityComboBox->currentText().isEmpty()) return;

  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  Project project = Session::current().mainProject();

  Variable porosity = project.wells()
  .get(_wellComboBox->currentText()).datasets()
  .get(_datasetComboBox->currentText()).variables()
  .get(_porosityComboBox->currentText());

  Variable density = project.wells()
  .get(_wellComboBox->currentText()).datasets()
  .get(_datasetComboBox->currentText()).variables()
  .get(_densityComboBox->currentText());

  computePorosityVsDensity(porosity, density);
  lock.release();
}

void activity::computePorosityVsDensity(Variable porosity,
Variable density)
{

  GraphicsScene graphicsScene =
  _customPlot.graphicsScenes().find("MyScene");

  if (graphicsScene.isNull())
    graphicsScene = GraphicsScene::create("MyScene",
   _customPlot);

  Dataset dataset = porosity.dataset();
  int rowCount = (int)dataset.rowCount();
  QList<QPointF> points;
  float xMin = 0, yMin = 0, xMax = 0, yMax = 0;
  for (int i = 0; i < rowCount; i++)
  {
    float x = porosity.getFloatValue(i);
    float y = density.getFloatValue(i);
    if (x == Absent::MissingValue || y == Absent::MissingValue)
      continue;

    if (x < xMin) xMin = x;
    else if (x > xMax) xMax = x;
```

```
    if (y < yMin) yMin = y;
    else if (y > yMax) yMax = y;

    points.append((QPointF(x,y)));
  }
  if (graphicsScene.pointsClouds().count() != 0)
    graphicsScene.pointsClouds().at(0).erase();

  PointsCloud pointsCloud = PointsCloud::create(graphicsScene);

  pointsCloud.appendPoints(points);
  pointsCloud.setPointsColor(Qt::blue);
  pointsCloud.setPointsSize(3);

  _customPlot.setHorizontalUserLimits(xMin, xMax);
  _customPlot.setXUnit(porosity.unit());
  _customPlot.setVerticalUserLimits(yMin, yMax);
  _customPlot.setYUnit(density.unit());
  _customPlot.setAxisXLegend(
  QString("% (%2)").arg(porosity.name()).arg(porosity.unit()));
  _customPlot.setAxisYLegend(
  QString("%1 (%2)").arg(density.name()).arg(density.unit()));
  _customPlot.setTitle(
  QString("Cross-plot: %1.%2").
  arg(dataset.name()).arg(dataset.well().name()));

  Variable ref = dataset.findReferenceVariable();
  if (!ref.isNull())
  {
     _customPlot.setSubtitle(QString("Reference (%1): [%2 -
%3]").arg(ref.unit()).arg(ref.getFloatValue(0)).arg(ref.getFlo
atValue(rowCount - 1)));
  }

  _customPlot.setDisplayChartLegend(false);
  _customPlot.setDisplayDefaultAction(false);
}

QStringList activity::listDatasetsByWell(Well well)
{
  QStringList result;
  foreach (const Dataset &dataset, well.datasets())
  {
    result.append(dataset.name());
  }
  return result;
```

SLB-Private

```
}

QStringList activity::listVariablesByDataset(Dataset dataset,
QString familyName)
{
  QStringList result;
  foreach (const Variable &variable, dataset.variables())
  {
    if (variable.family() == familyName)
      result.append(variable.name());
  }
  return result;
}
```

The following screenshot shows the **QWidget** added in the left pane to the **CustomPlot** and the porosity versus density crossplot created through **PointsCloud** domain object.

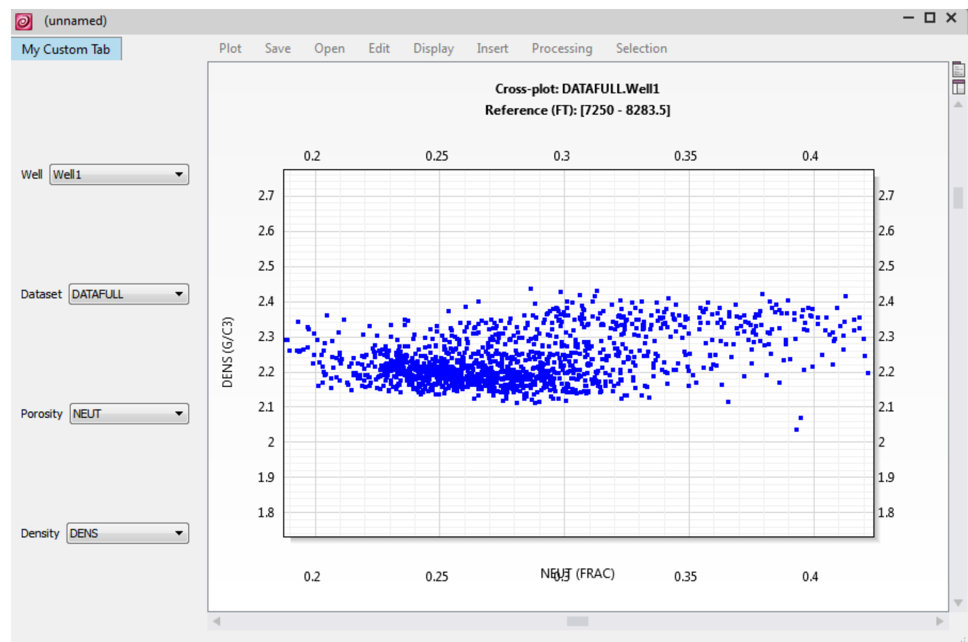


Figure 98 Example of CustomPlot

A depth player is added to the **CustomPlot** if a reference dataset is passed to the **create** static method.

```
class CustomPlot : public Plot
{
public:
  static CustomPlot create(Workspace workspace, const Dataset
  &depthPlayerReferenceDataset);
  …
  Unit referenceUnit() const;

  void setCurrentDepth(const double currentDepth,
```

```
  SendDepthInteractionEvent sendDepthInteractionEvent,
  const Unit &unit)
};
```

The **referenceUnit** property allows you to get the reference display unit which is deducted from the current project unit system and the dataset reference passed to the **create** static method to instantiate the plot (see “Display unit” in *Ocean for Techlog Developer Guide – Basics*).

The **currentDepth** function allows you to set the current depth value to the depth player. Through the function you can pass a depth value in a unit that will be converted to the display unit following the measurement of the reference dataset and the current project unit system.

Changing the **currentDepth** triggers the **Workspace::DepthInteractionChanged** signal.

See the “DepthInteractionChanged Signal” section in *Ocean for Techlog Developer Guide – Basics* for more information on **DepthInteractionChanged** signal.

```
class CustomPlot : public Plot
{
public:
  …
  double horizontalUserLowerLimit() const;
  double horizontalUserUpperLimit() const;
  double verticalUserLowerLimit() const;
  double verticalUserUpperLimit() const;
  double horizontalFamilyLowerLimit() const;
  double horizontalFamilyUpperLimit() const;
  double verticalFamilyLowerLimit() const;
  double verticalFamilyUpperLimit() const;
  double horizontalVariableLowerLimit() const;
  double horizontalVariableUpperLimit() const;
  double verticalVariableLowerLimit() const;
  double verticalVariableUpperLimit() const;
  void setHorizontalUserLimits(double
  horizontalUserLowerLimit, double horizontalUserUpperLimit);
  void setVerticalUserLimits(double verticalUserLowerLimit,
  double verticalUserUpperLimit);

  const Unit xUnit() const;
  void setXUnit(const Unit &xUnit);
  const Unit yUnit() const;
  void setYUnit(const Unit &yUnit);

  const PlotScaleType xScaleType() const;
  void setXScaleType(const PlotScaleType plotScaleType);
  const PlotScaleType yScaleType() const;
  void setYScaleType(const PlotScaleType plotScaleType);

  const QStringList legendItems() const;
  void setLegendItems(const QStringList &legendItems);
  void addLegendItem(const QString &htmlLegend);
  void removeLegendItems();
```

SLB-Private

```
const QString title() const;
void setTitle(const QString &title);
const QString subtitle() const;
void setSubtitle(const QString &subtitle);

const QString axisYLegend() const;
void setAxisYLegend(const QString &legend);
const QString axisXLegend() const;
void setAxisXLegend(const QString &legend);

const bool displayDefaultAction() const;
void setDisplayDefaultAction(const bool
displayDefaultAction);
const bool displayChartLegend() const;
void setDisplayChartLegend(const bool displayChartLegend);

const bool hideEmptyMenu() const;
void setHideEmptyMenu(const bool hideEmptyMenu);

bool isSideBoxVisible() const;
void setSideBoxVisible(bool visible);
quint64 sideBoxWidth() const;
void setSideBoxWidth(const quint64 sideBoxtWidth);

bool isMainGridVisible() const;
void setMainGridVisible(bool visible);
bool isHorizontalSecondaryGridVisible() const;
void setHorizontalSecondaryGridVisible(bool visible);
bool isVerticalSecondaryGridVisible() const;
void setVerticalSecondaryGridVisible(bool visible);
QColor secondaryGridColor() const;
void setSecondaryGridColor(const QColor &color);
int secondaryGridLineThickness() const;
void setSecondaryGridLineThickness(const int &thickness);
int secondaryGridHorizontalLineCount() const;
void setSecondaryGridHorizontalLineCount(const int
&lineCount);
int secondaryGridVerticalLineCount() const;
void setSecondaryGridVerticalLineCount(const int &lineCount);
QColor mainGridColor() const;
void setMainGridColor(const QColor &color);
int mainGridLineThickness() const;
void setMainGridLineThickness(int thickness);

void setKeepAspectRatio(PlotKeepAspectRatio keepAspectRatio);
PlotKeepAspectRatio keepAspectRatio() const;

bool isHorizontalAxisInverted() const;
void setHorizontalAxisInverted(bool inverted);
bool isVerticalAxisInverted() const;
void setVerticalAxisInverted(bool inverted);

bool isLegendBoxVisible() const;
void setLegendBoxVisible(bool visible);
```

SLB-Private

```
};
```

The properties of **CustomPlot** class allow you to:

- get variable and family upper and lower limits for X and Y axes
- get and set user upper and lower limits for X and Y axes
- get and set the unit X and Y axes
- get and set the scale type for X and Y axes through **PlotScaleType** enum class (can be linear or logarithmic)
- add and remove custom legend items, those items are added at the bottom of the plot and support HTML
- get and set a title and a subtitle for the plot
- get and set a text legend for X and Y axes
- show or hide mouse bar, tool bar and context menus of the plot plus enable or disable native menus though the **displayDefaultAction** property
- show / hide charts legend
- show / hide the side box that contains the custom tab Uis
- size the side box that contains the custom tab Uis
- change chart area grid properties as the visibility, the number of vertical and horizontal lines to display and the lines thickness
- enable / disable the "space proportion" option that locks / unlocks the aspect ratio of the chart area of the plot
- invert X and Y axes
- show / hide legend items in a **CustomPlot**

ContainerPlot Domain Object

The **ContainerPlot** class inherits from the **Plot** class and does not support name. A **ContainerPlot** object is instancied through **create** static method of the class with the parent **Workspace** object as argument. The uniqueness of a **ContainerPlot** object is handled by the system. The only way to retrieve a **ContainerPlot** object from the workspace is by its droid, stored for instance as a private member of the plug-in.

```
class ContainerPlot : public Plot
{
public:
  static ContainerPlot create(Workspace workspace);

  …
};
```

The **ContainerPlot** is used to group several plots in one. There is no limitation in the number of plots that you can group in a **ContainerPlot**.

The **ContainerPlot** domain object instance is passed as parent to the **create** static function of the plot that you want to add to the container plot.

For instance if you want to add a **CrossPlot** into a ContainerPlot you use one of the **create** static functions below:

```
class CrossPlot : public Plot
{
public:
  static CrossPlot create(const ContainerPlotPosition
  &containerPlotPosition);

  static CrossPlot create(const ContainerPlotPosition
  &containerPlotPosition, const CrossPlotTemplateDataBinding
  &crossPlotTemplateDataBinding);
};
```

The first **create** pattern allows to create a **Plot** into a **ContainerPlot** at a given position. The second create pattern allows to create a **Plot** by template into a **ContainerPlot** at a given position.

The position of the **Plot** in the **ContainerPlot** is handled by the **ContainerPlotPosition** class that holds:

- the **ContainerPlot** instance
- set the row and column indexes of the **Plot** in the **ContainerPlot**
- span on a number of rows and columns in the **ContainerPlot**

```
class ContainerPlotPosition : public Plot
{
public:
  ContainerPlotPosition(const ContainerPlot &containerPlot,
  int row, int column);

  ContainerPlotPosition(const ContainerPlot &containerPlot,
  int row, int column, int rowSpan, int columnSpan);
};
```

The plot template and data applied to the template is handled by **XXXDataBinding** classes added to the Ocean framework for each type of Plot as shown in the diagram below:

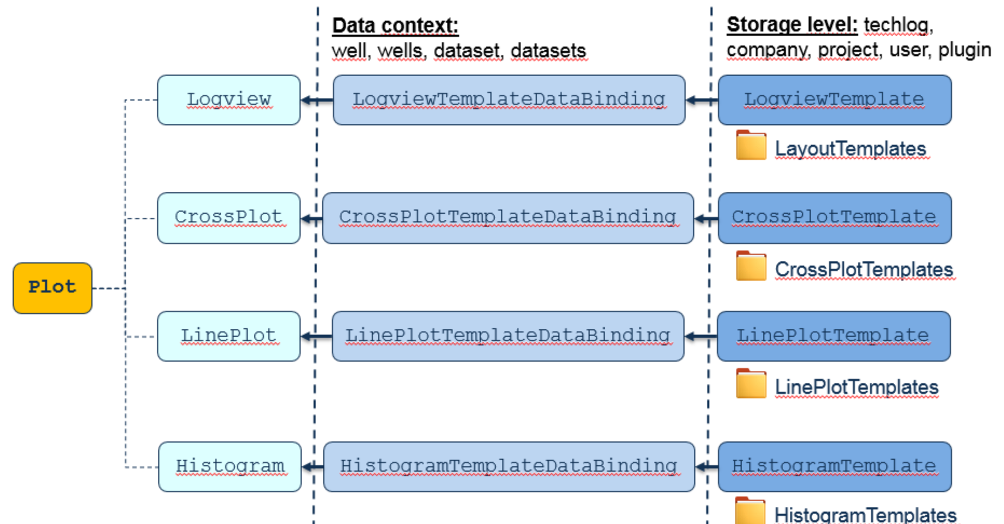


Figure 99 Class diagram for plot creation by template

See the plot sections that support to be created by template with Ocean for more information on how to create plots by template with Ocean.

The following is an example where a **Logview** is created into the **ContainerPlot** at first columm and row position spanning on two rows. Two **CrossPlot**’s are added to the second column of the **ContainerPlot**, one at first row and the other one at second row.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();

Project project = Session::current().mainProject();

Well well = project.wells().find("Well1");
if (well.isNull())
{
  lock.release();
  return;
}

Dataset dataset = well.datasets().get("DATAFULL");

ContainerPlot containerPlot = ContainerPlot::create(workspace);
containerPlot.setReferenceDataset(dataset);
containerPlot.setWindowTitle("My Group of plots");
containerPlot.setMaximized(true);

ContainerPlotPosition logviewPosition =
ContainerPlotPosition(containerPlot, 0, 0, 2, 1);
Logview logView = Logview::create(logviewPosition);

NormalTrack track1 = NormalTrack::create("Cali-Gamm-Soni",
logView);

Variable soni = dataset.variables().get("SONI");
LineTrackItem trackItemSoni = LineTrackItem::create(track1,
soni);
trackItemSoni.setColor(QColor(128, 0, 128));
Variable gamm = dataset.variables().get("GAMM");
LineTrackItem trackItemGamm = LineTrackItem::create(track1,
gamm);
QList<Baseline> baselines;
trackItemGamm.setBaselines(baselines);
Variable bs_dk = dataset.variables().get("BS_DK");
LineTrackItem::create(track1, bs_dk);
Variable cali = dataset.variables().get("CALI");
```

```
LineTrackItem::create(track1, cali);

NormalTrack track2 = NormalTrack::create("Density-Porosity",
logView);

Variable neut = dataset.variables().get("NEUT");
LineTrackItem::create(track2, neut);
Variable dens = dataset.variables().get("DENS");
LineTrackItem::create(track2, dens);
Variable c_phi = dataset.variables().get("C_PHI");
LineTrackItem trackItemCphi = LineTrackItem::create(track2,
c_phi);
trackItemCphi.setColor(Qt::blue);

ContainerPlotPosition crossPlot1Position =
ContainerPlotPosition(containerPlot, 0, 1);
CrossPlot crossPlot1 = CrossPlot::create(crossPlot1Position);
crossPlot1.setDepthListenerEnabled(true);
PlotScale defaultScale = crossPlot1.defaultPlotScale();
if (defaultScale.canAddXAxisVariable(soni))
  defaultScale.addXAxisVariable(soni);
if (defaultScale.canAddYAxisVariable(cali))
  defaultScale.addYAxisVariable(cali);
defaultScale.setColor(gamm);

ContainerPlotPosition crossPlot2Position =
ContainerPlotPosition(containerPlot, 1, 1);
CrossPlot crossPlot2 = CrossPlot::create(crossPlot2Position);
crossPlot2.setDepthListenerEnabled(true);
PlotScale defaultScale2 = crossPlot2.defaultPlotScale();
if (defaultScale2.canAddXAxisVariable(neut))
  defaultScale2.addXAxisVariable(neut);
if (defaultScale2.canAddYAxisVariable(dens))
  defaultScale2.addYAxisVariable(dens);
defaultScale2.setColor(c_phi);

lock.release();

lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

logView.setVerticalTopBottomPosition(7675, 7775,
logView.referenceUnit());

crossPlot1.setTitle(QString("%1.%2: %3 vs
%4").arg(well.name()).arg(dataset.name()).arg(soni.name()).arg
(cali.name()));
```

SLB-Private

```
crossPlot1.setSubtitle(QString("colored by
%1").arg(gamm.name()));

crossPlot2.setTitle(QString("%1.%2: %3 vs
%4").arg(well.name()).arg(dataset.name()).arg(neut.name()).arg
(dens.name()));
crossPlot2.setSubtitle(QString("colored by
%1").arg(c_phi.name()));

lock.release();
```

```
class ContainerPlot : public Plot
{
public:
  Unit referenceUnit() const;
  void setReferenceDataset(const Dataset &dataset);
  …
};
```

Setting the reference dataset of the **ContainerPlot** through **setReferenceDataset** function allows depth interaction between the plots within the **ContainerPlot** if depth listener is turned on (**CrossPlot::setDepthListenerEnabled**) and Logview "depth interaction" button with "Activate window mode" context menu button toggled on.

The **referenceUnit** property allows you to get the reference display unit which is deducted from the dataset reference and the current project unit system. See "Display unit" in *Ocean for Techlog Developer Guide – Basics*.

The following screenshot shows the rendering of the plots grouped into the **ContainerPlot**.

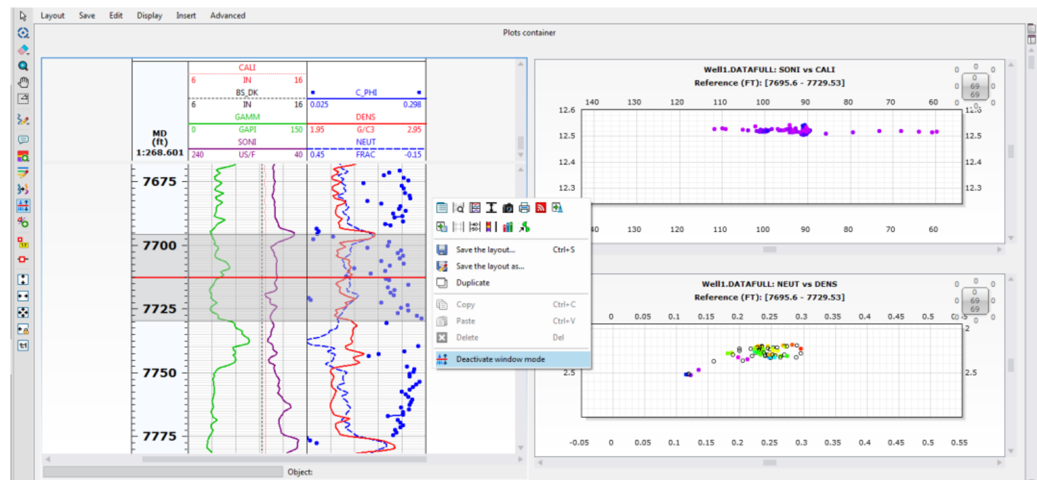


Figure 100 Plots grouped into a ContainerPlot

A depth player is added to the **CustomPlot** if all plots within the **ContainerPlot** are single-depth plots (they have a depth player).

Changing the currentDepth triggers the **Workspace::DepthInteractionChanged** signal.

See the "DepthInteractionChanged Signal" section in *Ocean for Techlog Developer Guide – Basics* for more information on **`DepthInteractionChanged`** signal.

SLB-Private