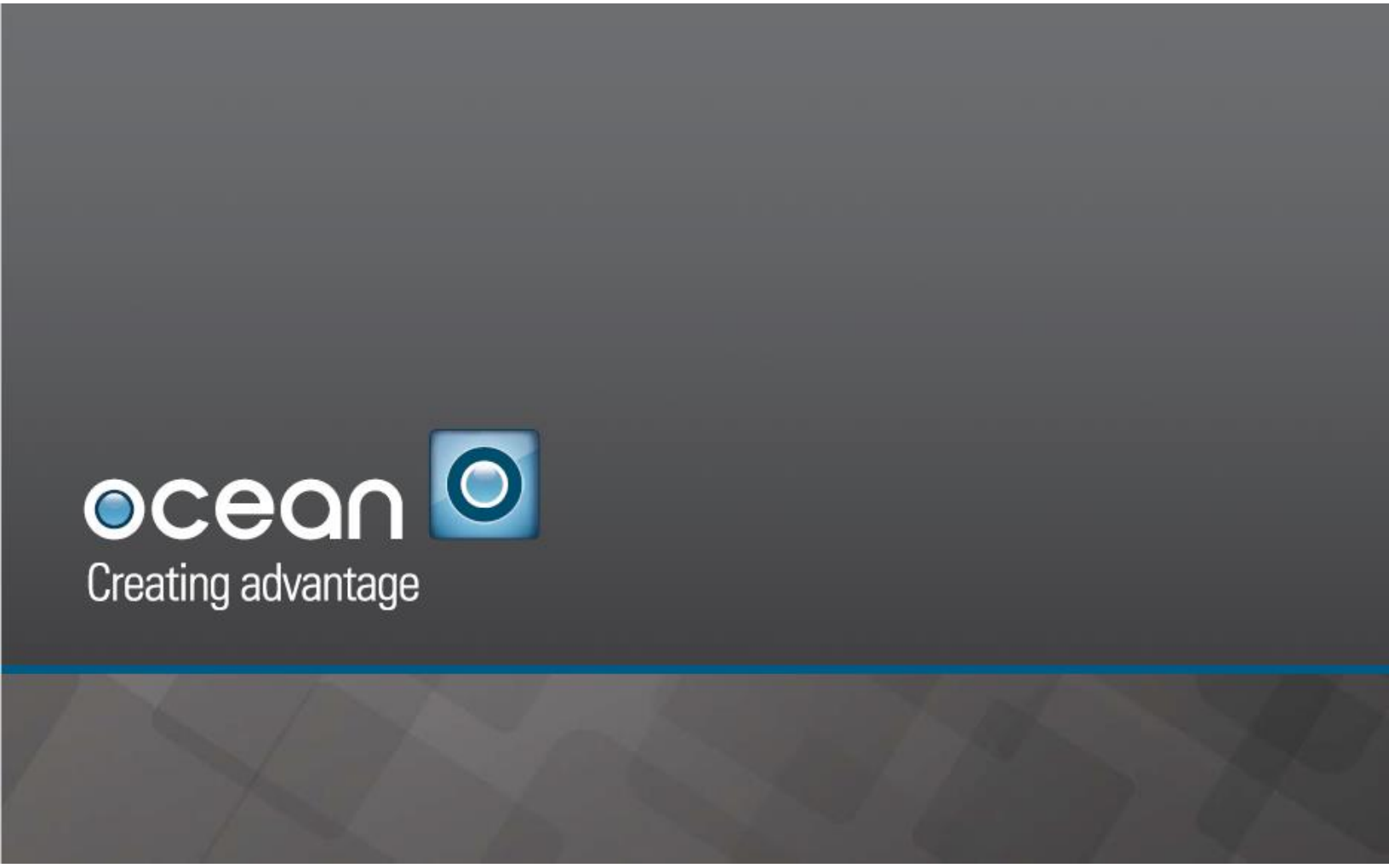
Data and Workflow

Volume 2



Ocean Software Development Framework for Techlog
Version 2023

Contents

1 Data Domain

In This Chapter

Project browser

The Techlog data domain is organized in one main hierarchy; access follows the Techlog **Project browser** data organization.

The data tree found in a Techlog project is organized this way:

- Well (or Wellbore) containing a dataset collection
- Dataset of variables with the same reference
- Variable with single or multi columns

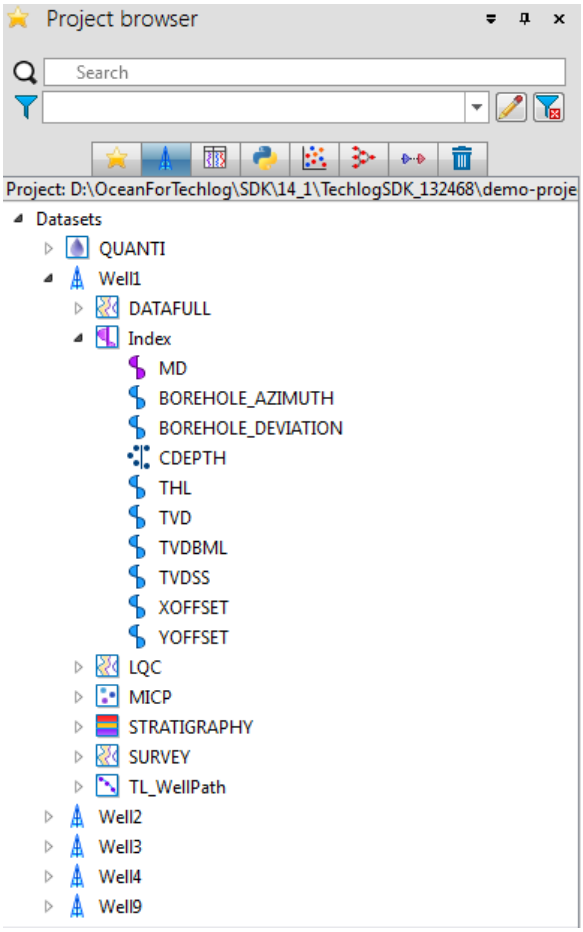


Figure 1-1 Techlog data tree

Dataset and variable

Dataset and variable are important concepts of the Techlog data model. One Techlog project contains wells. Each well contains one ore several datasets. And each dataset contains one reference and 0 or several variables.

A reference is the measurement, for any point in that well, of the distance between a reference point or elevation, and that point. It can be expressed in depth or time. Because wells are not always drilled vertically, there may be two "depths" for every given point in a wellbore: the measured depth (MD) measured along the path of the borehole, and the true vertical depth (TVD), the absolute vertical distance between the datum and the point in the wellbore. In perfectly vertical wells, the TVD equals the MD; otherwise, the TVD is less than the MD measured from the same datum. Common

datums used are ground level (GL), drilling rig floor (DF), rotary table (RT), kelly bushing (KB) and mean sea level (MSL).

In Techlog, the dataset reference is itself a variable. Any float 1-dimensional variable may be set as the reference of the dataset, but for Techlog to be able to use the variables in that dataset (to either process or plot them), the reference typically belongs to a "Reference" family and unit. References such as Measured Depth or Time enforce reference variable values to be monotically increasing.

A dataset has a count of rows. The rows represent the dataset natural index and all the variables in the dataset (including the reference) share the same index. The row count is stored as a property visible through the Techlog **Properties** editor when the dataset is selected in the project browser.

A variable has two storage formats: numerical or text. Each numerical variable can have multiple columns (it is an array variable). Capillary pressure and bore hole image variables are examples of array variables.

Data creation

Creating Techlog data domain objects is not done separately from placing them in the project data tree. This is intentional and the API does not provide constructors for domain object classes.

The placement of newly created Techlog data domain objects in the project follows the data tree organization that you have seen for data browsing.

You find methods to create specific domain object types in the class that represents the object itself. One argument of the create method is the domain object type that naturally contains that domain object (its parent).

These parent classes are entities that possess collection-type members and allow you to subscribe to children creation signals.

Navigation root

All data navigation starts from the `Project` object. Querying in a Techlog project is done by traversing or navigating the tree of domain object containers where the desired type of object is found.

The navigation starts at the root object which is be the main, import or export project. Those projects are retrieved from public methods by supplying the `Project` instance which is available from `Session` class.

From the `Project`, you access Techlog zonation and marker models or iterate over all the wells and plug-in domain objects belonging to the project. You can also retrieve a well or plug-in domain object by its name.

See "ProjectZonationModel object" on page 1-42 for more information on how to access zonations with Ocean from the project zonation model.

See "ProjectMarkerModel object" on page 1-50 for more information on how to access markers with Ocean from the project marker model.

See the "Plug-in domain object" section in *Ocean for Techlog Developer Guide - Plug-in Domain Object – Importer&Exporter* for more information on how to create and access `PluginDomainObject` with Ocean.

You can also get the list of wells filtered by the end-user in the Techlog project. If there is no filter applied in the project browser the **`filteredWells`** function returns the exhaustive list of **`wells`** in the project.

```
class Project : public DomainObject
{
public:
  ...
  ProjectZonationModel zonationModel() const;
  ProjectMarkerModel markerModel() const;

  DomainObjectCollection< Well> wells() const;
  Well findWell(const QString &wellName) const;
  Well getWell(const QString &wellName) const;

  DomainObjectCollection<Well> filteredWells() const;

  DomainObjectCollection<PluginDomainObject>
       pluginDomainObjects() const;
  PluginDomainObject findPluginDomainObject
      (const QString &pluginDomainObjectName) const;
  PluginDomainObject getPluginDomainObject
      (const QString &pluginDomainObjectName) const;
  bool canErase() const;
  static Project createTemporary();
  enum EventType
  {
    WellCreated
  };
};
```

Lock the Techlog session to get the project and navigate through its wells and zonations.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
// Get the current session
Session session = Session::current();
// Get the current workspace
Workspace workspace = session.currentWorkspace();
// Get the main project of the current session
Project mainProject = session.mainProject();
// Iterate on all the wells belonging to the main project
foreach (Well well, mainProject.wells())
{
  workspace.logEvent(LogLevelInformation, well.name());
}
// Iterate on all zonations belonging to the main project
foreach (GlobalZonation zonation,
mainProject.zonationModel().globalZonations())
```

```
{
  workspace.logEvent(LogLevelInformation, zonation.name());
}

// Iterate on all the markers belonging to the main project
foreach (GlobalMarkerSet globalMarkerSet,
mainProject.markerModel().globalMarkerSets())
{
  workspace.logEvent(LogLevelInformation,
  globalMarkerSet.name());
}

lock.release();
```

You cannot create a root **`Project`** object through Ocean. If there is no project open in Techlog, all plug-in activities are grayed out.

You can **`erase`** a **`Project`** with Ocean only if **`canErase`** function returns true. Only temporary projects can be erased. The **`createTemporary`** static function allows you to create a temporary project programmatically.

See "Temporary project" in *Ocean for Techlog Developer Guide - Basics* for more information on how temporay projects are handled.

The **`Project`** domain object implements **`WellCreated`** signal. A project instance can subscribe to this signal through the **`connect`** method inherited from the **`DomainObject`** base class. The **`WellCreated`** signal is emitted when a new well is added to the project.

Include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkwellcreatedargs.h"

private slots:
  void onWellCreated(
  const Slb::Ocean::Techlog::WellCreatedArgs &args);
```

Subscribe to the signals on a **`Project`** instance.

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

project.connect(Project::WellCreated, this,
SLOT(onWellCreated(const Slb::Ocean::Techlog::WellCreatedArgs
&)));

lock.release();
```

Unsubscribe from signals.

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

```
project.disconnect(Project::WellCreated, this,
SLOT(onWellCreated(const Slb::Ocean::Techlog::WellCreatedArgs
&)));

lock.release();
```

WellCreated signal

The **`WellCreated`** signal has a **`WellCreatedArgs`** argument that gives the new well added to the project.

```
class WellCreatedArgs : public SignalArgsT<Project>
{
public:
  ...
  Well newWell() const;
};
```

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

```
void activity::onWellCreated(const
Slb::Ocean::Techlog::WellCreatedArgs &args)
{
  Well well = args.newWell();
  // …
}
```

Well domain object

The well name is used as a unique identifier within a Techlog project and therefore you must always check for the existence of a well with a particular name in the project before creating a new one. This name may be changed after creation.

The well name is the first argument of the **`Well::create`** static method.

A well belongs to a project; see "Navigation root" section on page 1-4 for more details. A project parent instance must be passed as second argument of the **`Well::create`** static method.

```
class Well : public DataDomainObject
{
public:
  ...
  static Well create(const QString &name, Project project);
  void setName(const QString &);
};
```

This example retrieves a well by name in the root project before creating a new one.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
// Get the current session
```

```
Session session = Session::current();
// Get the main project of the current session
Project project = session.mainProject();
QString wellName = "MyWell";
// Check existence of the new well within main project
if (project.wells().contains(wellName))
{
  lock.release();
  return;
}
// or
Well well = project.wells().find(wellName);
if (!well.isNull())
{
  lock.release();
  return;
}
// or
well = project.findWell(wellName);
if (!well.isNull())
{
  lock.release();
  return;
}
// Create a new well within main project
well = Well::create(wellName, project);

lock.release();
```

Note: QUANTI and PQC are reserved names and you cannot create a well with these names.

A well in Techlog is a logical grouping of datasets which is normally used to represent data along a given borehole. Datasets belonging to the well are enumerated using the **`datasets`** function.

You can get the list of datasets filtered by the end-user in the Techlog project for the given **`Well`**. If there is no filter applied in the project browser the **`filteredDatasets`** function returns the exhaustive list of **`datasets`** in the well.

You can also iterate over all wellbore schematic datasets belonging to the project or retrieve a wellbore schematic dataset by its name.

A **`WellboreSchematic`** is a dataset that can be created programmatically with Ocean. A **`WellboreSchematic`** dataset contains data on well tools and drilling actions used along the wellbore.

See "WellboreSchematic domain object" on page 1-16 for more information on how to create a **`WellboreSchematic`** dataset with Ocean.

```
class Well : public DataDomainObject
{
public:
  ...
  const DomainObjectCollection<Dataset> datasets() const;
  Dataset findDataset(const QString &datasetName) const;
  Dataset getDataset(const QString &datasetName) const;

  DomainObjectCollection<Dataset> filteredDatasets() const;

  const DomainObjectCollection<WellboreSchematic>
       wellboreSchematics() const;
  WellboreSchematic findWellboreSchematic(const QString &name)
       const;

  enum EventType
  {
    DatasetCreated,
    PluginDomainObjectCreated,
    WellboreSchematicCreated
  };
};
```

The **`Well`** domain object implements **`DatasetCreated`**, **`PluginDomainObjectCreeated`** and **`WellboreSchematicCreated`** signal. A well object subscribes to those signals through the **`connect`** method inherited from **`DomainObject`** base class.

- The **`DatasetCreated`** signal is emitted when a new **`Dataset`** is added to the **`Well`**.
- The **`PluginDomainObjectCreated`** signal is emitted when a new **`PluginDomainObject`** is added to the **`Well`**.
- The **`WellboreSchematicCreated`** signal is emitted when a new **`WellboreSchematic`** is added to the **`Well`**.

Include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkdatasetcreatedargs.h"
#include "tsdkplugindomainobjectcreatedargs.h"
#include "tsdkwellboreschematiccreatedargs.h"
```

```
private slots:
  void onDatasetCreated(
  const Slb::Ocean::Techlog::DatasetCreatedArgs &args);
  void onPluginDomainObjectCreated(
  const Slb::Ocean::Techlog::PluginDomainObjectCreatedArgs
&args);
  void onWellboreSchematicCreated(
```

```
  const Slb::Ocean::Techlog::WellboreSchematicCreatedArgs
&args);
```

This example receives the new well object created in argument of the **`onWellCreated`** slot method; lock the well object before subscribing to the **`DatasetCreated`** signal.

```
void activity::onWellCreated(const
Slb::Ocean::Techlog::WellCreatedArgs &args)
{
  Well well = args.newWell();
  // …
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  well);

  well.connect(Well::DatasetCreated, this,
  SLOT(onDatasetCreated(const
  Slb::Ocean::Techlog::DatasetCreatedArgs&)));

  well.connect(Well::PluginDomainObjectCreated, this,
  SLOT(onPluginDomainObjectCreated(const
  Slb::Ocean::Techlog::PluginDomainObjectCreatedArgs&)));

  well.connect(Well::WellboreSchematicCreated, this,
  SLOT(onWellboreSchematicCreated(const
  Slb::Ocean::Techlog::WellboreSchematicCreatedArgs&)));

  lock.release();
}
```

DatasetCreated signal

The **`DatasetCreated`** signal includes a **`DatasetCreatedArgs`** argument that gives the new **`Dataset`** added to the well.

```
class DatasetCreatedArgs : public SignalArgsT<Well>
{
public:
  ...
  Dataset newDataset() const;
};
```

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

```
void activity::onDatasetCreated(const
Slb::Ocean::Techlog::DatasetCreatedArgs &args)
{
  Dataset dataset = args.newDataset();
  // …
```

```
}
```

PluginDomainObjectCreated signal

The **PluginDomainObjectCreated** signal includes a **PluginDomainObjectCreatedArgs** argument that gives the new **PluginDomainObject** added to the sender **DomainObject** that can be **Project**, **Well**, **Dataset**, **Variable** or **PluginDomainObject**.

```
class PluginDomainObjectCreatedArgs : public
SignalArgsT<DomainObject>
{
public:
  ...
  PluginDomainObject newPluginDomainObject() const;
};
```

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

```
void activity::onPluginDomainObjectCreated(const
Slb::Ocean::Techlog::PluginDomainObjectCreatedArgs &args)
{
  PluginDomainObject pdo = args.newPluginDomainObject();
  // …
}
```

WellboreSchematicCreated signal

The **WellboreSchematicCreated** signal includes a **WellboreSchematicCreatedArgs** argument that gives the new **WellboreSchematic** added to the well.

```
class WellboreSchematicCreatedArgs : public SignalArgsT<Well>
{
public:
  ...
  WellboreSchematic newWellboreSchematic() const;
};
```

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

```
void activity::onWellboreSchematicCreated(const
Slb::Ocean::Techlog::WellboreSchematicCreatedArgs &args)
{
  WellboreSchematic wellboreSchematic =
    args.newWellboreSchematic();
  // …
}
```

The dataset name is used as a unique identifier within a well. You must always check for the existence of a dataset with a particular name in the same well before creating a new one. This name may be changed after creation.

The dataset name is the first argument of the **`Dataset::create`** static method.

A dataset contains the well log data (called variables in Techlog). The list of variables contained in a dataset share the same index. The index is one of the dataset variables called the reference variable. The reference variable has a single column and has either a float or a double precision.

The reference variable name and format are the second and third arguments respectively of the **`Dataset::create`** static method. A reference variable with the name and format passed to the **`create`** method is automatically added to the new dataset.

The fourth argument of **`Dataset::create`** method is the size of the dataset. This can be changed after creation.

A dataset belongs to a well. A well parent instance must be passed as fifth argument of the **`Dataset::create`** static method.

Another **`create`** static method in the **`Dataset`** class allows you to create an empty dataset without reference variable.

```
class Dataset : public DataDomainObject
{
public:
  ...
  static Dataset create(const QString &name, const QString
      &referenceVariableName, VariableDataFormat
      referenceVariableFormat, quint64 rowCount, Well well);
  static Dataset create(const QString & name,
      quint64 rowCount, Well well);
 void setName(const QString &);
};
```

This example retrieves a dataset by its name before creating it:

```
Well well = createMyWell(true);

Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well);
QString datasetName = "MyDataset";
// Check existence of the new dataset within the well
if (well.datasets().contains(datasetName))
{
  lock.release();
  return;
}
// or
Dataset dataset = well.datasets().find(datasetName);
if (!dataset.isNull())
```

```
{
  lock.release();
  return;
}
// or
dataset = well.findDataset(datasetName);
if (!dataset.isNull())
{
  lock.release();
  return;
}
// Create a new dataset within the well
dataset = Dataset::create(datasetName, "MD",
                    VariableDataFormatFloat, 100, well);

lock.release();
```

At this stage, you have a dataset created with an empty reference variable named "MD". See "Variable domain object" section on page 1-19 for more information on how to populate and properly set a reference variable.

In addition, the **`Dataset`** class allows you to:

- find and set the reference variable
- unset the reference variable (empty dataset)
- get the parent well
- get and set the dataset type through **`DatasetType`** enum class
- get and set the row count of the dataset (if the dataset contains variables with a rowCount greater than the new rowCount property value entered for the dataset, variable values will be truncated)
- get the list of variables that belongs to the dataset
- retrieve a child variable by its name
- get the list of variables filtered by the end-user in the Techlog project for the given **`Dataset`**. If there is no filter applied in the project browser the **`filteredVariables`** function returns the exhaustive list of **`variables`**in the dataset.

```
class Dataset : public DataDomainObject
{
public:
  ...
  Variable findReferenceVariable() const;
  void setReferenceVariable(const Variable & variable);
  void unsetReferenceVariable();
  Well well() const;
  const DatasetType type() const;
  void setType(const DatasetType & value);
  quint64 rowCount() const;
```

```
  void setRowCount(const quint64 &value);
  const DomainObjectCollection<Variable> variables() const;
  const DomainObjectCollection<Variable> temporaryVariables()
        const;
  Variable findVariable(const QString &variableName) const;
  Variable getVariable(const QString &variableName) const;

  DomainObjectCollection<Variable> filteredVariables() const;

  enum EventType
  {
    VariableCreated,
    DatasetTypeChanged,
    DatasetDataSizeChanged,
    PluginDomainObjectCreated
  };
};
```

Enumerate the variables belonging to the dataset using the **`variables`** function.

Enumerate temporary variables belonging to the dataset using the **`temporaryVariables`** function.

The **`Dataset`** domain object implements **`VariableCreated`** signal. A dataset object can subscribe on this signal through the connect method inherited from **`DomainObject`** base class. **`VariableCreated`** signal is emitted when a new variable is added to the dataset.

The **`DatasetTypeChanged`** and **`DatasetDataSizeChanged`** signals are emitted if the **`type`** and **`rowCount`** properties are modified respectively.

See "PluginDomainObjectCreated signal" on page 1-11 for more information on how to connect this signal to a **`Dataset`**.

Include signal arguments and declare slot receivers in the activity header file.

```
include "tsdkvariablecreatedargs.h"
#include "tsdkdatasettypechangedargs.h"
#include "tsdkdatasetdatasizechangedargs.h"
```

```
private slots:
  void onVariableCreated(
  const Slb::Ocean::Techlog::VariableCreatedArgs &args);

  void onDatasetTypeChanged(
  const Slb::Ocean::Techlog::DatasetTypeChangedArgs &args);

  void onDatasetDataSizeChanged(
  const Slb::Ocean::Techlog::DatasetDataSizeChangedArgs &args);
```

This example receives the new dataset object created in the argument of the **`onDatasetCreated`** slot method; lock the dataset object before subscribing to the

VariableCreated, **DatasetSizeChanged** and **DatasetDataSizeChanged** signals.

```
void activity::onDatasetCreated(const
Slb::Ocean::Techlog::DatasetCreatedArgs &args)
{
  Dataset dataset = args.newDataset();
  // …
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  dataset);

  dataset.connect(Dataset::VariableCreated, this,
  SLOT(onVariableCreated(
  const Slb::Ocean::Techlog::VariableCreatedArgs&)));

  dataset.connect(Dataset::DatasetTypeChanged, this,
  SLOT(onDatasetTypeChanged(
  const Slb::Ocean::Techlog::DatasetTypeChangedArgs&)));

  dataset.connect(Dataset::DatasetDataSizeChanged, this,
  SLOT(onDatasetDataSizeChanged(
  const Slb::Ocean::Techlog::DatasetDataSizeChangedArgs&)));

  lock.release();
}
```

VariableCreated signal

The **VariableCreated** signal has a **VariableCreatedArgs** argument that gives the new variable added to the dataset.

```
class VariableCreatedArgs : public SignalArgsT<Dataset>
{
public:
  ...
  Variable newVariable() const;
};
```

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

```
void activity::onVariableCreated(const
Slb::Ocean::Techlog::VariableCreatedArgs &args)
{
  Variable variable = args.newVariable();
}
```

DatasetTypeChanged and DatasetDataSizeChanged signals

The **DatasetTypeChanged** and **DatasetDataSizeChanged** signals include the **DatasetTypeChangedArgs** and **DatasetDataSizeChangedArgs** arguments respectively; they return the **Dataset** object and the changed property value, respectively.

```
class DatasetTypeChangedArgs : public SignalArgsT<Dataset>
{
};
```

```
class DatasetDataSizeChangedArgs : public SignalArgsT<Dataset>
{
};
```

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

```
void activity::onDatasetTypeChanged(const
Slb::Ocean::Techlog::DatasetTypeChangedArgs &args)
{
  Dataset dataset = args.sender();
}
```

WellboreSchematic domain object

Ocean gives the ability to create a **WellboreSchematic** dataset programmatically that can be displayed in the **Logview** through the **WellSchematicTrackItem**.

See the "WellSchematicTrackItem domain object" section in *Ocean for Techlog Developer Guide - Plots* for more information on how to plot a **WellboreSchematic** into the **Logview** with Ocean.

Its is a domain object derived from the PluginDomainObject class and modeled in Ocean with the **WellboreSchematic** class.

This class contains a **create** static method that allows you to create the **WellboreSchematic** under a given Well.

The collection of **WellboreSchematic** can be listed from the **Well**.

See "Well domain object" on page number 1-7 for more information on how to access **WellboreSchematic**s from the Well.

```
class WellboreSchematic : public PluginDomainObject
{
public:
  ...
  static WellboreSchematic create(const QString &name, Well
    parent);

  QByteArray data() const;
  ReturnValue<bool> trySetData(const QByteArray &data);
```

```
  void setData(const QByteArray &data);

  enum EventType
  {
    WellboreSchematicDataChanged
  };
};
```

Once the **`WellboreSchematic`** is created you can populate it with some data using the **`setData`** function that expects a QByteArray formated from a WITSML file.

Shown below is an example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Project project = Session::current().mainProject();
Well well = project.findWell("WellSetupLogview");
if (well.isNull())
{
  lock.release();
  return;
}
QFile file("wellbore.witsml");
QByteArray data;
if (file.open(QIODevice::ReadOnly))
{
  data = file.readAll();
  file.close();
}
WellboreSchematic wellboreSchematic =
WellboreSchematic::create("WellSchematicPDO", well);
if (!wellboreSchematic.trySetData(data))
{
  qWarning() << "Unable to set wellboreSchematic data";
}
lock.release();
```

As you can see in the screenshot below the new **`WellboreSchematic`** created with the above code snippet is represented into the project browser without child variables.

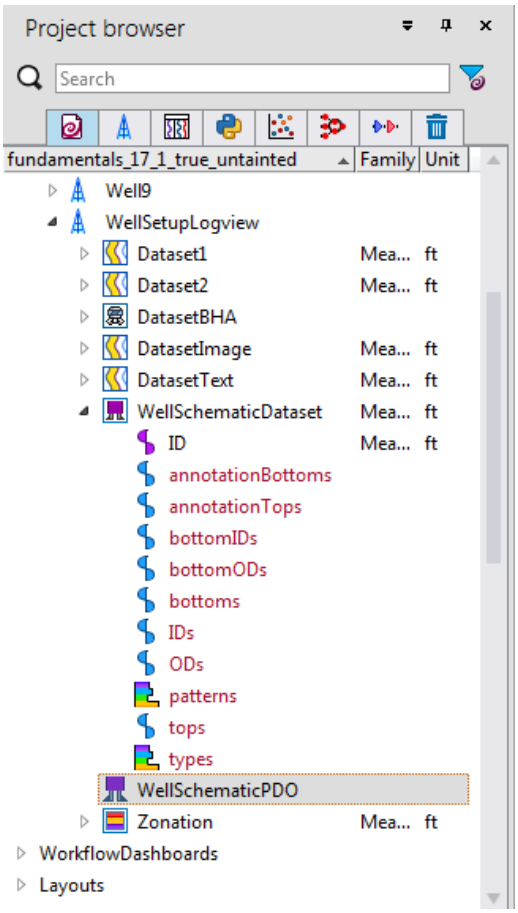


Figure 1-2 WellboreSchematic plug-in domain object

The **`WellboreSchematic`** plug-in domain object implements a **`WellboreSchematicDataChanged`** signal. A **`WellboreSchematic`** object can subscribe on this signal through the connect method inherited from **`DomainObject`** base class. The **`WellboreSchematicDataChanged`** signal is emitted when the **`WellboreSchematic`** bulk data is changed.

Include the signal argument and declare the slot receiver in the activity header file.

```
#include "tsdkplugindomainobjectdatachangedargs.h"
```

```
private slots:
  void onWellboreSchematicDataChanged(
  const Slb::Ocean::Techlog::PluginDomainObjectDataChangedArgs
&args);
```

As you can see in the example below the connection with the **`WellboreSchematicDataChanged`** signal is done with a slot receiver that is expecting as signature argument a **`PluginDomainObjectDataChangedArgs`** object. This is due to the fact that the **`WellboreSchematic`** object is exposed as a plug-in domain object.

```
void activity::onWellboreSchematicCreated(const
Slb::Ocean::Techlog::WellboreSchematicCreatedArgs &args)
{
```

```
  WellboreSchematic wellboreSchematic =
    args.newWellboreSchematic();
  // …

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  wellboreSchematic);

  wellboreSchematic.connect(
  WellboreSchematic::WellboreSchematicDataChanged, this,
  SLOT(onWellboreSchematicDataChanged(
  const
Slb::Ocean::Techlog::PluginDomainObjectDataChangedArgs&)));

  lock.release();
}
```

See the "PluginDomainObjectDataChanged signal" section in *Ocean for Techlog Developer Guide - Plug-in Domain Object – Importer&Exporter* for more information on how to implement the **`WellboreSchematicDataChanged`** slot receiver function.

Variable domain object

A **`Variable`** in Techlog is an array of values along an index.

The variable name is used as a unique identifier within a dataset. You must always check for the existence of a variable with a particular name in the same dataset before creating a new one.

The variable name is the first argument of the **`Variable::create`** static method.

A variable belongs to a dataset. A dataset parent instance must be passed as second argument of the **`Variable::create`** static method.

The format, type and number of columns of the variable must be passed respectively as third, fourth and fifth arguments of the **`create`** method.

The variable name and type can be changed after creation.

Note: The variable column count must be 1 for double and string formats because Techlog does not support arrays for these formats. The variable format, column count and parent dataset are not-mutable attributes.

```
class Variable : public DataDomainObject
{
public:
  ...
  static Variable create(const QString & name, Dataset dataset,
    VariableDataFormat format, VariableType type, quint64
    columnCount);
  void setName(const QString &name);
  const QString family() const;
  void setFamily(const QString & value);
```

```
  const QString unit() const;
  void setUnit(const QString & value);
  Unit displayUnit() const;
  VariableType type() const;
  void setType(const VariableType & value);
  VariableDataFormat format() const;
  quint64 columnCount() const;
  bool isReference() const;
};
```

After you create the variable, assign a family and unit to the variable.

The family is a tag applied to a group of variables which have equivalent characteristics, but they are different because they were not logged by the same tool. The family can set default values on some variable properties, like the variable type.

The **`unit`** property represents the unit with which the variable values are stored; this may be different than the **`displayUnit`**.

With the introduction of the project unit system, variables have now a display unit that may be different from the storage unit (the unit corresponding to the raw data). When the user changes the unit system, the display unit is updated depending on the current unit system.

The display unit is accessible directly from the **`Variable`** using the **`displayUnit`** property. See "Unit catalog" in *Ocean for Techlog Developer Guide - Basics* for details about how to retrieve the display unit from a **`Measurement`** using the **`UnitCatalog::getDisplayUnitFromMeasurement`** function.

This example creates a variable and sets those attributes:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
QString variableName = "GR";
// Check existence of the new variable within the dataset
if (dataset.variables().contains(variableName))
{
  lock.release();
  return;
}
// or
Variable variable = dataset.variables().find(variableName);
if (!variable.isNull())
{
  lock.release();
  return;
}
// or
variable = dataset.findVariable(variableName);
if (!variable.isNull())
{
  lock.release();
  return;
```

```
}
// Create a single curve variable within the dataset
variable = Variable::create(variableName, dataset,
VariableDataFormatFloat, VariableTypeContinuous, 1);
// set Gamma Ray family and unit
variable.setFamily("Gamma Ray");
UnitCatalog unitCatalog = Session::current().unitCatalog();
Unit unitGR = unitCatalog.getDisplayUnitFromMeasurement(
TechlogMeasurement::getGammaRay());
variable.setUnit(unitGR);

lock.release();
```

A variable can be a temporary variable. A temporary variable is not saved in the dataset and does not appear in the Techlog project browser. Get if a variable is temporary or not using the **`isTemporary`** property of the **`Variable`**. Temporary variables can be created programmatically through the **`createTemporary`** static function. Manipulate temporary variables the same way as normal variables during the Techlog session. They are removed at the end of the Techlog session.

Temporary variables are also generated by the Techlog Application Workflow Interface when it runs with apply mode set to "display only". See the "Workstep results display" section on page 2-54 for more information on Application Workflow Interface set to display only.

```
class Variable: public DataDomainObject
{
public:
  static Variable createTemporary(const QString &name,
      Dataset dataset, VariableDataFormat format,
      VariableType type, quint64 columnCount)
  bool isTemporary() const;
  ...
};
```

All variables in a dataset share the same index. The index is a variable itself, called the dataset reference variable. Any float 1-dimensional variable can be set as the reference of the dataset, but for Techlog to use (either process or plot them) the variables in that dataset, the reference typically belongs to a "Reference" family and unit. An exhaustive list of all reference families is available in Techlog **Families database** window.

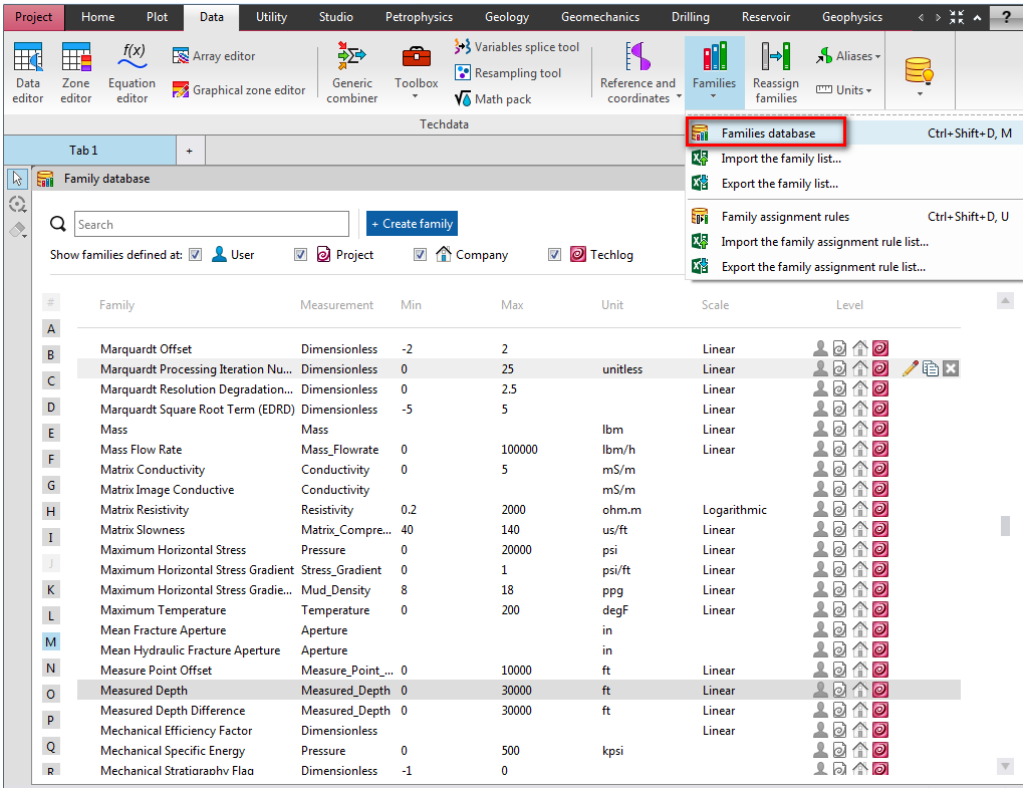


Figure 1-3 Families database

Get the reference variable using **`Dataset::findreferenceVariable`** function and set a "Reference" family and unit. If there is no reference variable set to the dataset then **`findReferenceVariable`** returns null.

Note: Some reference families like "Measured Depth" or "Time" imply that reference variable values should be monotically increasing. This is not mandatory in Techlog but this rule can be checked by some Techlog modules.

Populating the variable with values is the last step of the variable creation. **`Variable`** has read and write accessors for the bulk data.

```
class Variable : public DataDomainObject
{
public:
  ...
  float getFloatValue(quint64 row, quint64 column) const;
  float getFloatValue(quint64 row) const;
  double getDoubleValue(quint64 row, quint64 column) const;
  double getDoubleValue(quint64 row) const;
  QString getStringValue(quint64 row, quint64 column) const;
  QString getStringValue(quint64 row) const;
  QDateTime getDateTimeValue(quint64 row, quint64 column) const;
  QDateTime getDateTimeValue(quint64 row) const;
```

```
  QVector<float> getFloatValues() const;
  QVector<double> getDoubleValues() const;
  QVector<QString> getStringValues() const;
  QVector<QDateTime> getDateTimeValues() const;

  void setFloatValues(const QVector<float> &values);
  void setDoubleValues(const QVector<double> &values);
  void setStringValues(const QVector<QString> &values);
  void setDateTimeValues(const QVector<QDateTime> &values);

  QVector<float> getFloatValuesRange(quint64 row,
      quint64 count) const;
  QVector<double> getDoubleValuesRange(quint64 row,
      quint64 count) const;

  void setFloatValuesRange(quint64 row,
      const QVector<float> &values);
  void setDoubleValuesRange(quint64 row,
      const QVector<double> &values);
  void setStringValuesRange(quint64 row,
      const QVector<QString> &values);
  void setDateTimeValuesRange(quint64 row,
      const QVector<QDateTime> &values);

  void setFloatValue(quint64 row, quint64 column, float value);
  void setFloatValue(quint64 row, float value);
  void setDoubleValue(quint64 row, quint64 column, double value);
  void setDoubleValue(quint64 row, double value);
  void setStringValue(quint64 row, quint64 column,
       const QString &value);
  void setStringValue(quint64 row, const QString &value);
  void setDateTimeValue(quint64 row, quint64 column,
       const QDateTime &value);
  void setDateTimeValue(quint64 row, const QDateTime &value);

  enum EventType
  {
    VariableDataChanged,
    VariableDataSizeChanged,
    VariableDataFormatChanged,
    VariableTypeChanged,
    VariableUnitChanged,
    VariableFamilyChanged,
    PluginDomainObjectCreated
  };
};
```

Getters and setters are related to the variable format. You access a row value of a variable with float format using the **`getFloatValue`** function.

The different ways to access a variable are:

- get or set the variable value by row index and column index
- get or set a **`QVector<T>`** equal to the size of the variable (**`rowCount`***__`columnCount`__). For an array variable, **`QVector`** values must be set in row-major order.
- get or set a range of values in a **`QVector<T>`** starting at a row index (index + **`QVector`** size cannot be greater than the variable row count). Get a range of values can only be called on variables with double or float format.

The way variable data are manipulated can significantly affect performance. Best practices are to:

- Access the values sequentially to take advantage of CPU caches.
- Avoid implicit casts (use **`getFloatValue`** if the data type is float, etc.).
- Access the data by small chunks for large variables (less than 10MB) rather than all the values, to avoid the risk of running out of memory.
- Don't keep references on objects that you don't need any more so the system can remove them from the cache and recover some memory.
- Don't keep unsaved data modifications for too long so the system can recover the memory if needed.
- Only change the data that needs to change, and change the values sequentially in increasing index order.
- Avoid accessing variable attributes such as **`Variable::rowCount`** or **`Variable::columnCount`** in a loop:
  instead of "for (quint i = 0 ; i < rowCount() ; i++) {}",
  write "quint rowCount = rowCount(); for (quint i = 0 ; i < rowCount ; i++) {}"

This example populates variables with the different methods:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Dataset dataset = Dataset::create("MyDataset", "MD",
VariableDataFormatFloat, 10, well);
// get the reference variable and set a "Reference" family, unit
and depth values with 10 meters interval
Variable ref = dataset.findReferenceVariable();
ref.setFamily("Measured Depth");
ref.setUnit("ft");
quint64 rowCount = dataset.rowCount();
for (int i = 0; i < rowCount; i++)
{
  ref.setFloatValue(i, (float)i*10);
}

QString variableName = "GR";
// Check existence of the new variable within the dataset
Variable variable = dataset.variables().find(variableName);
```

```
if (!variable.isNull())
{
  lock.release();
  return;
}
// Create a single curve variable within the dataset
variable = Variable::create(variableName, dataset,
                    VariableDataFormatFloat,
VariableTypeContinuous, 1);
// set Gamma Ray family, unit
variable.setFamily("Gamma Ray");
UnitCatalog unitCatalog = Session::current().unitCatalog();
Unit unitGR = unitCatalog.getDisplayUnitFromMeasurement(
TechlogMeasurement::getGammaRay());
variable.setUnit(unitGR);

// Replace all the values of the variable
// by a vector with length = 10 and default value = 2.0
QVector<float> vector1((int)dataset.rowCount(), 2.0f);
variable.setFloatValues(vector1);

// Replace values of the variable starting from row index 3
// by a vector with length = 3 and default value = 5.0
QVector<float> vector2(3, 5.0f);
variable.setFloatValuesRange(3, vector2);

lock.release();
```

This screenshot displays the expected result:

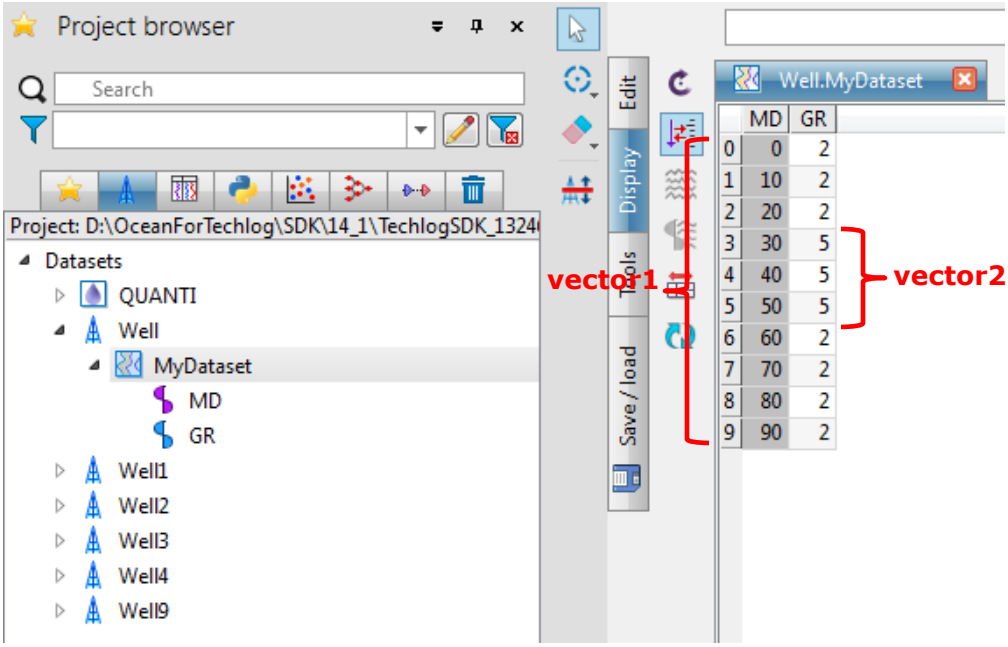


Figure 1-4 Variable data

In Techlog it is possible to set values for the secondary index of an array variable. At each column the index string, double or timestamp values can be set.

Ocean provides functions in the **`Variable`** class that allow you to set a vector of values with the corresponding type to variable columns. The vector size must be equal to the variable **`columnCount`** and the corresponding column data format must be set previously to avoid any plug-in exception.

The default **`columnDataFormat`** is inherited from the variable **`format`**. If the variable has been created with a float format, the column data format is set to double.

Note: DateTime values must be converted to UTC and **`VariableDataFormat`** set to double before setting the column timestamps vector.

```
class Variable: public DataDomainObject
{
public:
  VariableDataFormat columnDataFormat() const;
  void setColumnDataFormat(VariableDataFormat format);
  const QStringList columnNames() const;
  void setColumnNames(const QStringList &columnNames);
  void clearColumnNames();
  const QVector<double> columnReferences() const;
  void setColumnReferences(const QVector<double>
       &columnReferences);
  void clearColumnReferences();
  constQVector<QDateTime> columnTimeStamps() const;
  void setColumnTimeStamps(const QVector<QDateTime>
       &columnTimeStamps);
  void clearColumnTimeStamps();
  ...
};
```

This example sets the secondary index.

```
Lock lock = LOCK_CREATE();
lock.add(arrayVariable1);
lock.add(arrayVariable2);
lock.add(arrayVariable3);
LOCK_ACQUIRE_OR_RETURN(lock);

QStringList columnNames;
for (quint64 columnIndex = 0; columnIndex <
arrayVariable1.columnCount(); ++columnIndex)
  columnNames << QString::fromLatin1("Column
%1").arg(columnIndex);

arrayVariable1.setColumnDataFormat(VariableDataFormatString);
arrayVariable1.setColumnNames(columnNames);
```

```
QVector<double> columnReferences;
for (quint64 columnIndex = 0; columnIndex <
arrayVariable2.columnCount(); ++columnIndex)
  columnReferences << 25.0f*columnIndex;

arrayVariable2.setColumnDataFormat(VariableDataFormatDouble);
arrayVariable2.setColumnReferences(columnReferences);

QVector<QDateTime> columnTimeStamps;
QDateTime dateTime = QDateTime::currentDateTime();
for (quint64 columnIndex = 0; columnIndex <
arrayVariable3.columnCount(); ++columnIndex)
{
  columnTimeStamps << dateTime.toUTC();
  dateTime = dateTime.addDays(1);
}

arrayVariable3.setColumnDataFormat(VariableDataFormatDouble);
arrayVariable3.setColumnTimeStamps(columnTimeStamps);

lock.release();
```

Column values are shown in the **Columns** tab of the **Properties** editor for the variable.

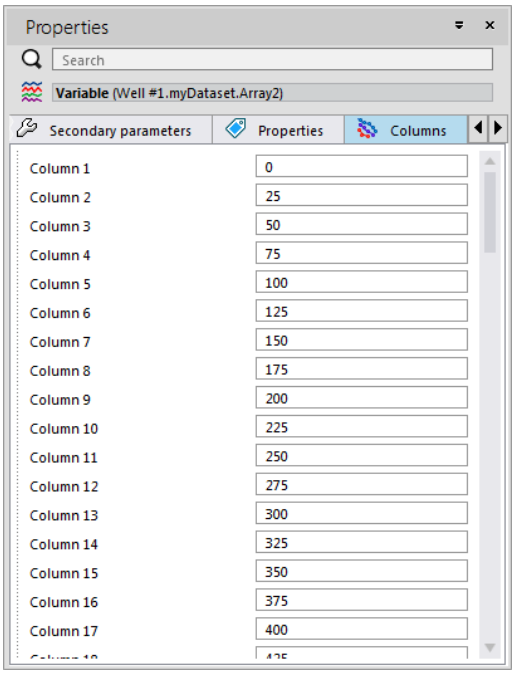


Figure 1-5 Column values in the properties editor

The **Variable** domain object implements signals. A variable object can subscribe to those signals through the **connect** method inherited from the **DomainObject** base class.

- **VariableDataChanged** signal is emitted when variable bulk data is changed.
- **VariableDataSizeChanged** signal is emitted when the variable data size is changed (related to properties **rowCount** and **columnCount**).
- **VariableDataFormatChanged** signal is emitted when the variable data format as **VariableDataFormat** is changed (related to property **format**).
- **VariableTypeChanged** signal is emitted when the variable type format as **VariableType** is changed (related to property **type**).
- **VariableUnitChanged** signal is emitted when the variable unit is changed (related to property **unit**).
- **VariableFamilyChanged** signal is emitted when the variable family is changed (related to property **family**).
- **PluginDomainObjectCreated** signal is emitted when a new **PluginDomainObject** is added to the **Variable**.

  See "PluginDomainObjectCreated signal" on page 1-11 for more information on how to connect this signal to a **Variable**.

Include signal arguments and declare slot receivers in the activity header file.

```
#include "tsdkvariabledatachangedargs.h"
#include "tsdkvariabledatasizechangedargs.h"
#include "tsdkvariabledataformatchangedargs.h"
#include "tsdkvariabletypechangedargs.h"
#include "tsdkvariableunitchangedargs.h"
#include "tsdkvariablefamilychangedargs.h"
```

```
private slots:
  void onVariableDataChanged(
  const Slb::Ocean::Techlog::VariableDataChangedArgs &args);

  void onVariableDataSizeChanged(
  const Slb::Ocean::Techlog::VariableDataSizeChangedArgs
&args);

  void onVariableDataFormatChanged(
  const Slb::Ocean::Techlog::VariableDataFormatChangedArgs
&args);

  void onVariableTypeChanged(
  const Slb::Ocean::Techlog::VariableTypeChangedArgs &args);

  void onVariableUnitChanged(
```

```
  const Slb::Ocean::Techlog::VariableUnitChangedArgs &args);

  void onVariableFamilyChanged(
  const Slb::Ocean::Techlog::VariableFamilyChangedArgs &args);
```

This example connects to those signals.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dataset);
QString variableName = "GR";
Variable variable = dataset.findVariable(variableName);
if (!variable.isNull())
{
  variable.connect(Variable::VariableDataChanged, this,
  SLOT(onVariableDataChanged(
  Slb::Ocean::Techlog::VariableDataChangedArgs&)));

  variable.connect(Variable::VariableDataSizeChanged, this,
  SLOT(onVariableDataSizeChanged(
  Slb::Ocean::Techlog::VariableDataSizeChangedArgs&)));

  variable.connect(Variable::VariableDataFormatChanged, this,
  SLOT(onVariableDataFormatChanged(
  Slb::Ocean::Techlog::VariableDataFormatChangedArgs&)));

  variable.connect(Variable::VariableTypeChanged, this,
  SLOT(onVariableTypeChanged(
  Slb::Ocean::Techlog::VariableTypeChangedArgs&)));

  variable.connect(Variable::VariableUnitChanged, this,
  SLOT(onVariableUnitChanged(
  Slb::Ocean::Techlog::VariableUnitChangedArgs&)));

  variable.connect(Variable::VariableFamilyChanged, this,
  SLOT(onVariableFamilyChanged(
  Slb::Ocean::Techlog::VariableFamilyChangedArgs&)));

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

Some statistical information on variable values are exposed in the **`Variable`** class through the functions listed below:

```
class Variable: public DataDomainObject
{
public:
```

```
  const QString sourceFileName();
  const double minimumValue();
  const double maximumValue();
  const double meanValue();
  const double standardDeviation();
  const uint numberOfMissingValues();
  const int topIndex();
  const int bottomIndex();

  void addHistory(const QString &message);
  QList<HistoryRecord> getHistoryRecords();
  ...
};
```

Those functions map some variable properties displayed below in the properties editor:

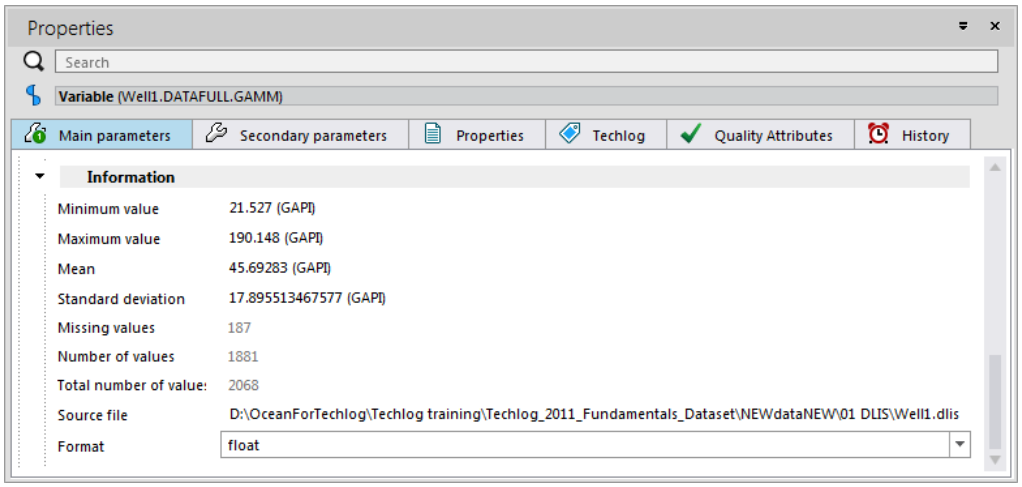


Figure 1-6 Variable information properties

The **`addHistory`** function allows to add a message to the history records of the variables that can be listed through the **`getHistoryRecords`** function and visualized in the History tab of the properties editor.

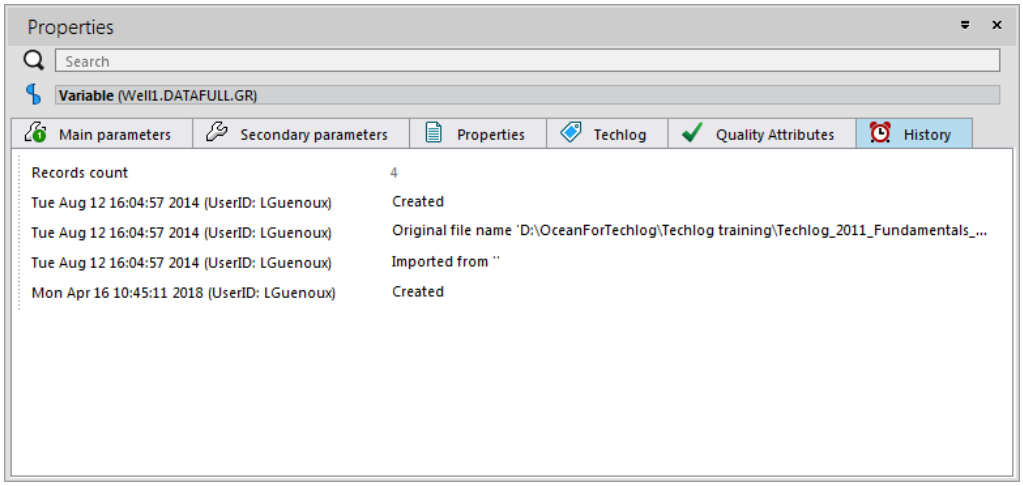


Figure 1-7 Variable history records

An entry in the history tab of the variable is modeled with Ocean through the **`HistoryRecord`** object from which you can read **`userID`**, **`dateTime`** and **`message`** values.

```
class HistoryRecord
{
public:
  QString userID() const;
  QDateTime dateTime() const;
  QString message() const;
};
```

VariableDataChanged signal

The **`VariableDataChanged`** signal includes a **`VariableDataChangedArgs`** argument that gives the affected object (variable where the data has been changed).

```
class VariableDataChangedArgs : public SignalArgsT<Variable>
{
};
```

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

```
void activity::onVariableDataChanged(const
Slb::Ocean::Techlog::VariableDataChangedArgs &args)
{
  Variable variable = args.sender();
}
```

VariableDataSizeChanged, VariableDataFormatChanged, VariableUnitChanged and VariableFamilyChanged signals

All the other signals include arguments that return the **`Variable`** object.

```
class VariableDataSizeChangedArgs : public SignalArgsT<Variable>
{
};
```

```
class VariableDataFormatChangedArgs :
      public SignalArgsT<Variable>
{
};
```

```
class VariableUnitChangedArgs : public SignalArgsT<Variable>
{
};
```

```
class VariableFamilyChangedArgs : public SignalArgsT<Variable>
{
};
```

The slot handler accesses the needed informationfrom the signal argument.

```
void activity::onVariableDataSizeChanged(const
Slb::Ocean::Techlog::VariableDataSizeChangedArgs &args)
{
  Variable variable = args.sender();
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  variable);
  quint64 rowCount = variable.rowCount();
  qWarning() << "new variable data size: " << rowCount;
  lock.release();
}
```

ImageVariable

Image variables are created with Ocean using the **`ImageVariable`** class. It derives from the **`Variable`** class. Pass the same arguments to the **`create`** static methods as you did for the **`Variable`** objects, except for the variable data format and column count. The variable data format is always **`VariableDataFormatString`** and the column count is always 1. A new argument for **`ImageVariable`** is the **`ImageVariableImageStorage`** image storage enum.

- **`ImageVariableImageStorageInternal`** – Images are saved in Techlog project.
- **`ImageVariableImageStorageExternal`** – Images are saved externally on the disk via an absolute path to the image.

`VariableType` is either a **`VariableTypeCoreImage`** or a **`VariableTypeBoreholeImage`**.

```
class ImageVariable: public Variable
{
public:
  static ImageVariable create(const QString &name,
      Dataset dataset, const VariableType type,
      const ImageVariableImageStorage storage);
  static ImageVariable createTemporary(const QString &name,
      Dataset dataset, const VariableType type,
      const ImageVariableImageStorage storage);
  ...
};
```

For an **`ImageVariable`** created with internal storage, set the image with a **`QImage`** or a **`QByteArray`** at a given row index of the dataset.

```
class ImageVariable: public Variable
```

```
{
public:
  void setImageForInternalStorage(const quint64 row,
      const QImage &image, const QString &name);
  void setImageDataForInternalStorage(const quint64 row,
      const QByteArray &data, const QString &name);
  ...
};
```

For an **`ImageVariable`** created with external storage, set the image with a **`QFileInfo`** at a given row index of the dataset.

```
class ImageVariable: public Variable
{
public:
  void setImageForExternalStorage(const quint64 row,
      const QFileInfo &fileInfo);
  ...
};
```

This scenario has 16 core images with a sampling rate of 3 feet which are stored in the plug-in folder.

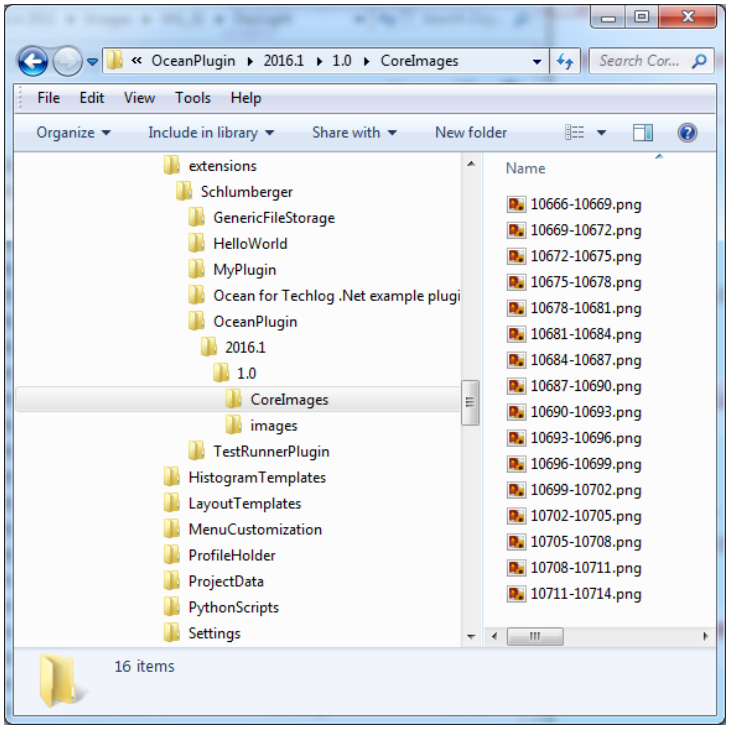


Figure 1-8 Core images

This example creates two internal image variables that store these core images respectively as **`QImage`** and **`QByteArray`** objects. It also creates an external image variable that points to these core image files using **`QFileInfo`** objects.

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

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

QString wellName = project.getUniqueWellName("Well");

Well well = Well::create(wellName, project);

int nbCoreImages = 16;

Dataset dataset = Dataset::create("myDataset", "MD",
VariableDataFormatFloat, nbCoreImages, well);

Variable reference = dataset.findReferenceVariable();
reference.setFamily("Measured Depth");
reference.setUnit("ft");

ImageVariable imgVarQByteArrayInternal =
ImageVariable::create("QByteArrayInternal", dataset,
VariableTypeCoreImage,
ImageVariableImageStorageInternal);ImageVariable
imgVarQImageInternal = ImageVariable::create("QImageInternal",
dataset, VariableTypeCoreImage,
ImageVariableImageStorageInternal);

ImageVariable imgVarExternal = ImageVariable::create("External",
dataset, VariableTypeCoreImage,
ImageVariableImageStorageExternal);

int samplingRate = 3;
float topDepth = 10666;
QString pathToImages = Session::current().pluginDirectory() +
"\\CoreImages\\";

for (int i = 0; i < nbCoreImages; i++)
{
  reference.setFloatValue(i, 0, topDepth);
  float bottomDepth = topDepth + samplingRate;
  QString imageName = QString::number(topDepth) + "-" +
  QString::number(bottomDepth);
  QString imageFullName = pathToImages + imageName + ".png";

  // set a QImage to internal storage image variable
  QImage qImage(imageFullName);
  qWarning() << imageName;
  imgVarQImageInternal.setImageForInternalStorage(i, qImage,
imageName);

  // set a QByteArray to internal storage image variable
  QFile file(imageFullName);
```

```
  file.open(QIODevice::ReadOnly);
  QByteArray qByteArray = file.readAll();
  imgVarQByteArrayInternal.setImageDataForInternalStorage(i,
qByteArray,
  imageName);

  // set a QFileInfo to external storage image variable
  QFileInfo fileInfo(pathToImages, imageName + ".png");
  imgVarExternal.setImageForExternalStorage(i, fileInfo);

  topDepth = bottomDepth;
}

lock.release();
```

This screenshot shows the result and how variables are stored following the storage type.

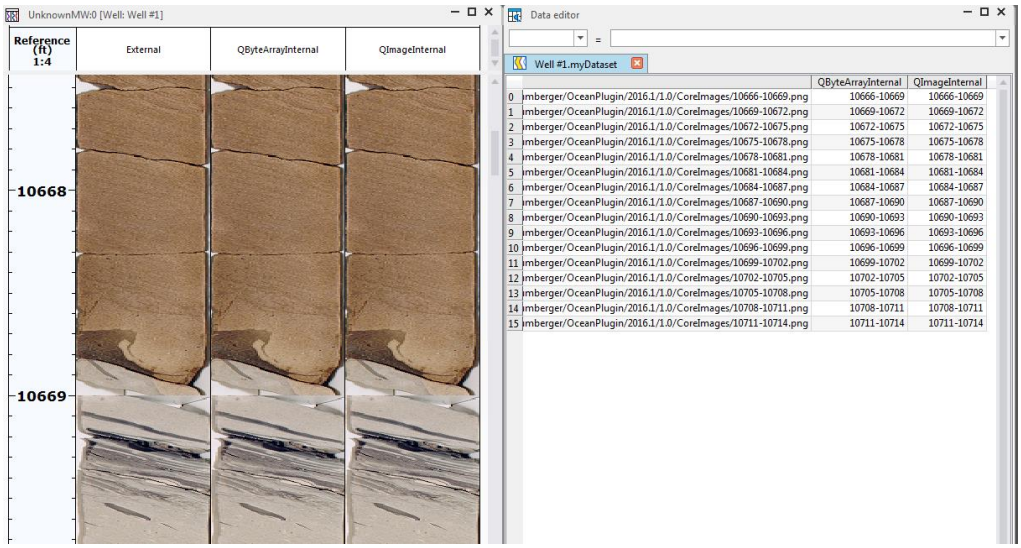


Figure 1-9 Image variables with different storage type

The **`ImageVariable`** can be displayed in a track of a **Logview** through the **`ImageTrackItem`** domain object.

See "Track and TrackItem Domain Objects" in *Ocean for Techlog Developer Guide - Plots* for more information on how to create a track in a **Logview** with an **`ImageTrackItem`**.

Data properties

Data like wells, datasets, variables and plug-in domain objects can have data properties.

See the "Plug-in domain object" section in *Ocean for Techlog Developer Guide - Plug-in Domain Object – Importer&Exporter* for more information on how to create and access **`PluginDomainObject`**.

A data property is the result of a measurement, a computation or an interpretation and brings some information about its related data entity. When a data domain object is selected in the Techlog project browser, its data properties are shown in the **Properties** tab of the Techlog **Properties** editor.

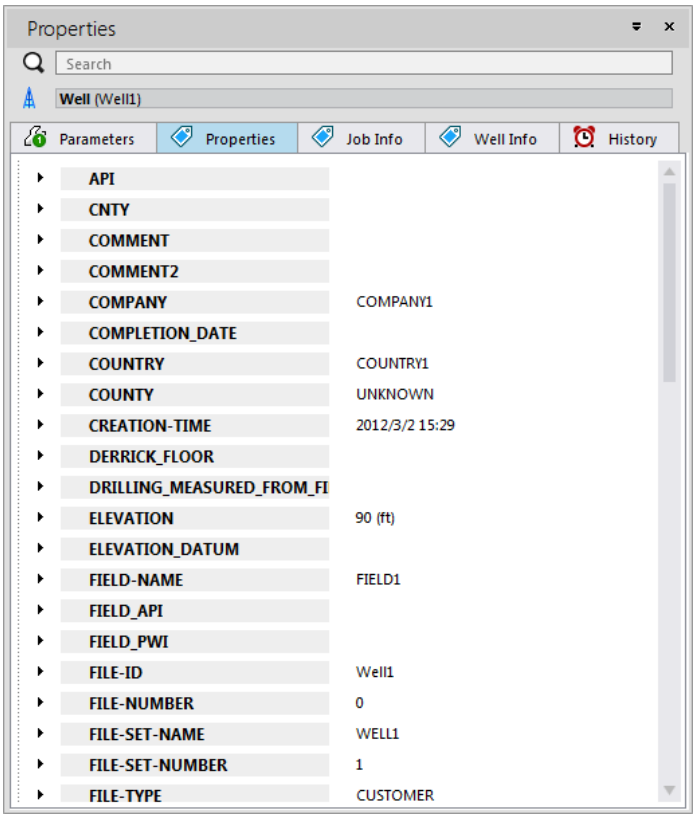


Figure 1-10 Data properties displayed in the Techlog properties editor

Use the **`DataProperty`** class to access a data property; it is not a domain object. A **`DataProperty`** object is instantiated using the class constructor.

```
class DataProperty : public IPrintable
{
public:
  DataProperty(const QString &name);
  const QString name() const;
  const QString value() const;
  void setValue(const QString & value);
  const QString unit() const;
  void setUnit(const QString & unit);
  const QString description() const;
  void setDescription(const QString & description);
  bool isEmpty() const;
};
```

`Well`, **`Dataset`**, **`Variable`** and **`PluginDomainObject`** classes inherit from the **`DataDomainObject`** class which contains a list of **`DataProperty`** objects.

```
class DataDomainObject : public DomainObject
{
public:
  QSet<DataProperty> dataProperties() const;

  DataProperty findDataProperty(const QString &dataPropertyName)
      const;
  DataProperty getDataProperty(const QString &dataPropertyName)
      const;
  bool containsDataProperty(const QString &dataPropertyName)
     const;

  void addDataProperty(const DataProperty &dataProperty);
  void updateDataProperty(const DataProperty &dataProperty);
  void addOrUpdateDataProperty(const DataProperty
       &dataProperty);
  void removeDataProperty(const QString &dataPropertyName);

};
```

A data property is composed of four strings: the name, value, unit and description.

The data property name is used as a unique identifier within a data entity. You must always check for the existence of a data property with a particular name in the same data entity before creating a new one.

Note: The name of the property cannot be empty and name uniqueness is validated by the parent **`DataDomainObject`** when adding or updating it.

A **`DataProperty`** is a value-semantic object, which means it is created in the stack. It has no knowledge of its parent. To modify (add, update, or remove) it in its parent **`DataDomainObject`**, you must modify it locally, and then pass it to its container, for it to be changed in the parent container.

Note: If **`DataProperty`** is not found in the **`DataDomainObject`**, the **`isEmpty`** method returns true.

Read and write accessors are available for the unit, value and description attributes. The name cannot be changed after creation. You must remove the **`DataProperty`** from its container and recreate it with the new name.

This example adds an elevation property to a well and copies this property to another one.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, well1);
if (well1.containsDataProperty("ELEVATION"))
{
  lock.release();
  return;
}
DataProperty dataProperty ("ELEVATION");
dataProperty.setValue("90");
dataProperty.setUnit("ft");
```

```
dataProperty.setDescription("Elevation reference is the RT");
well1.addDataProperty(dataProperty);
lock.release();

// ability to retrieve the "ELEVATION" data property from well1
lock = LOCK_CREATE();
lock.add(well1);
lock.add(well2);
LOCK_ACQUIRE_OR_RETURN(lock);

DataProperty elevation = well1.findDataProperty("ELEVATION");
if (elevation.isEmpty())
{
  lock.release();
  return;
}

// and add this property to well2 without any checks
well2.addOrUpdateDataProperty(elevation);

lock.release();
```

These are some data property rules:

- All data is stored as a string, even numerical values.
- The data property value can contain array values stored as a semi-colon separated string.
- There are restrictions for some reserved property names (See **`setValue`** description in **OceanForTechlog.chm** file)
- There is no restriction on data property value length or content (support HTML).
- The data property name cannot be empty and must be unique.
- A data property instance cannot be shared between data entities. If you want to have 2 data entities with the same property, you must create duplicate data properties on each data entity.

Zonation creation

Zonations are created at the well level and stored in datasets with "interval" type. This kind of dataset has two variables:

- A reference variable with family and unit like all other datasets. This reference variable contains the depth intervals of the zones.
- A zone name variable which contains the name of the zones.

The zonation dataset is modeled in Ocean for Techlog through the **`Dataset`** domain object.

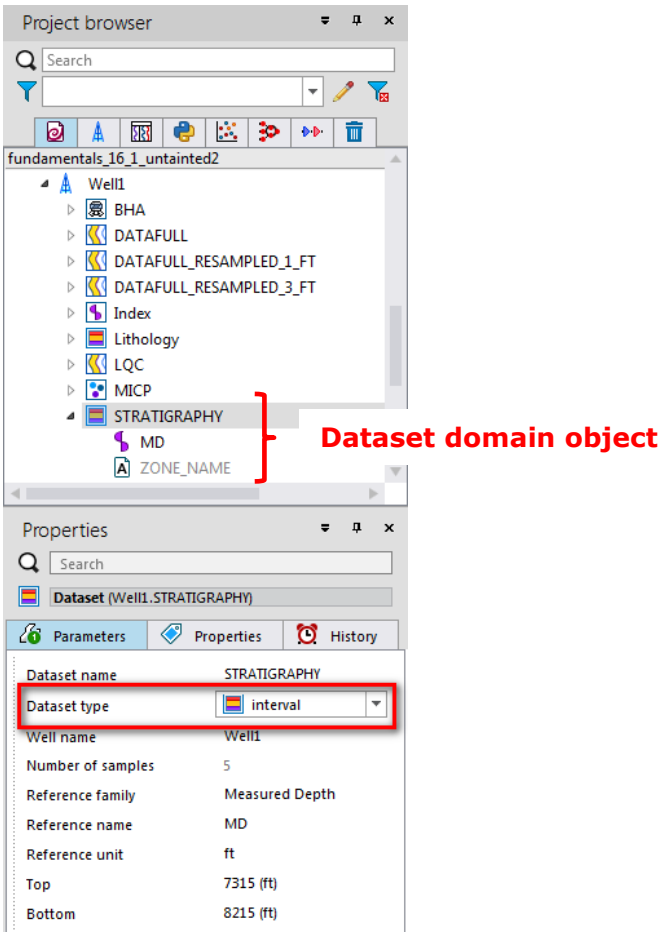


Figure 1-11 Interval Dataset

Interval datasets must have unique names within the same well but can share the same name if it is in a different well. This is the same rule for any dataset.

The Techlog **Zonation** window stores all the interval sets in the project. It is composed of two panes:

- the drop-down list with that merges the names of the zonation datasets
- the zone pane where the zones belonging to the selected dataset are displayed

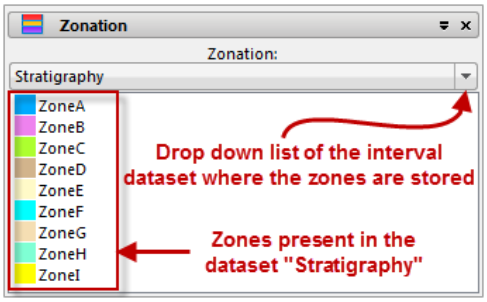


Figure 1-12 Techlog Zonation Window

In Ocean for Techlog, each item in this drop-down list is a `GlobalZonation` object.

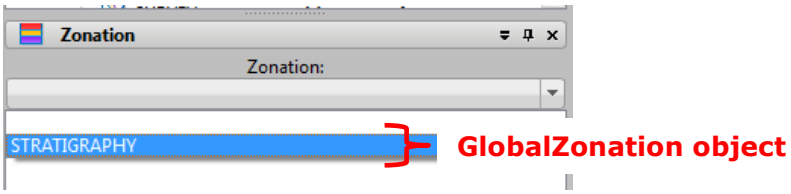


Figure 1-13 Zonation

Zonation dataset

Create the Zonation dataset using the **`Dataset`** domain object.

The zonation name is used as a unique identifier within a well, like is done for any other dataset, so you must check for the existence of a zonation dataset with a particular name in the same well before creating a new one.

The zonation name is the first argument of the **`Dataset::create`** static method. The reference family name, unit, and format are respectively the second, third and fourth arguments. A zonation belongs to a well. A well parent instance must be passed as fifth argument.

```
class Dataset : public DomainObject
{
public:
  ...
  static Dataset create( const QString& name, const QString&
    family, const QString& unit, VariableDataFormat format,
    Well well);

 Well well() const;
 QList<Zone> zones(const ZoneFamilyType &family) const;
 void setZones(const QList<Zone>& zones,
   const ZoneFamilyType &family);
 QString zonationFamily() const;
 QString zonationUnit() const;
 bool isZonationDataset() const;
 bool isZonationAll() const;
};
```

Once a zonation **`Dataset`** object is created there is:

- a dataset of type "interval" (**`DatasetTypeInterval`** enum value)
- a reference variable with float or double format
- a string variable with family: Zone Name, Hydraulic or FaultBlock

The reference variable values represent the intervals (top and bottom depth) of each zone. The other variable stores zone names regarding their family.

`ZoneFamilyType` defines the type of zones.

```
class ZoneFamilyType
{
public:
```

```
  QString name() const;
  bool isEmpty() const;

  static ZoneFamilyType zoneFamilyName();
  static ZoneFamilyType zoneFamilyHydraulic();
  static ZoneFamilyType zoneFamilyFaultBlock();
};
```

Instantiate it via the following static functions:

- **`zoneFamilyName`** – a table associating zone name with the depth index
- **`zoneFamilyHydraulic`** – a hydraulic zone is a structure or fault block that is hydraulically connected and that has common contacts at initial reservoir conditions
- **`zoneFamilyFaultBlock`** – a table associating depth index and fault block number

The variables are populated through a collection of zones set on the **`Dataset`** object.

Create a **`Zone`** object with a public constructor that takes the name, top depth and bottom depth of the zone as arguments.

```
class Zone
{
public:
   Zone( const Zone& rhs );
   Zone( const QString& name, double top, double bottom );
   ~Zone();

   QString name() const;
   void setName( const QString& value );
  double top() const;
   void setTop( double value );
   double bottom() const;
   void setBottom( double value );
};
```

This example creates a zonation data set.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
if (well.datasets().contains("Lithology"))
{
  lock.release();
  return;
}
// Create zonation "Lithology" in Well1
Dataset lithology = Dataset::create("Lithology", "MD", "m",
VariableDataFormatFloat, well);

// Add zones to the well zonation
// Zone name, top and bottom values (MD in meters)
```

```
QList<Zone> zones;
zones << Zone("Shale", 0.f, 50.f);
zones << Zone("Sandstone", 50.f, 150.f);
zones << Zone("Shale", 150.f, 230.f);
zones << Zone("Limestone", 230.f, 380.f);

// Set zones to the WellZonation collection
lithology.setZones( zones, ZoneFamilyType::zoneFamilyName() );

lock.release();
```

The family, unit and format of the reference variable containing the top and bottom depths of the zones are immutable properties. You may read those properties but not change them programmatically.

The Ocean for Techlog API provides read and write accessors for the name, top depth and bottom depth values of a **`Zone`** object.

Note: If after the creation of your zonation **`Dataset`**, you do not add any **`Zones`** before releasing the locked objects, an empty zonation is displayed in the Techlog zonation window without the corresponding dataset and variable instances in the well. Creating a zonation **`Dataset`** requires you to lock all.

ProjectZonationModel object

`ProjectZonationModel` object is only instantiated through the **`Project::zonationModel`** function. The **`ProjectZonationModel`** object contains all the functions to retrieve a **`GlobalZonation`** object in the Techlog project or to modify **`GlobalZonation`** object properties.

See the "GlobalZonation object" section on page 1-42 for more information on **`GlobalZonation`**.

```
class ProjectZonationModel
{
public:
  QList<GlobalZonation> globalZonations() const;
  GlobalZonation findGlobalZonation(const QString
    &zonationName) const;
  void  setGlobalZonePattern (const QString &zonationName,
    const QString &zoneName, const QString &patternName,
    const StorageLevel patternlevel);
  void setGlobalZoneColor(const QString &zonationName,
    const QString &zoneName, const QColor &color)
};
```

GlobalZonation object

You cannot create a **`GlobalZonation`** object; you access the **`GlobalZonation`** list with the **`ProjectZonationModel::globalZonations`** method.

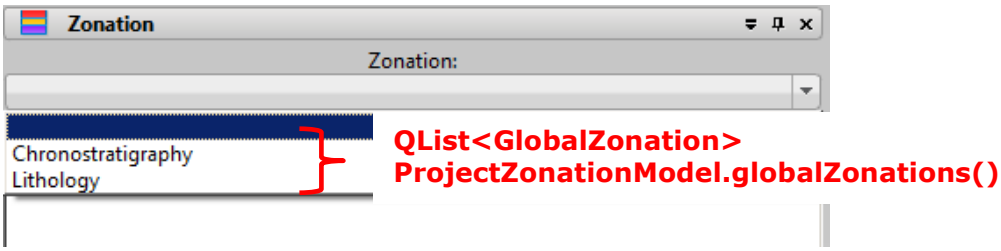


Figure 1-14 List of GlobalZonation from the Project

A **`GlobalZonation`** is the merge of all the zonation **`Datasets`** with the same name in different wells. If we have a "Chronostratigraphy" **`Dataset`** in Well1 and Well2 then "Chronostratigraphy" will be available as a **`GlobalZonation`** object in the zonation collection of the project. This "Chronostratigraphy" zonation object contains all the zones of Well1 and Well2.

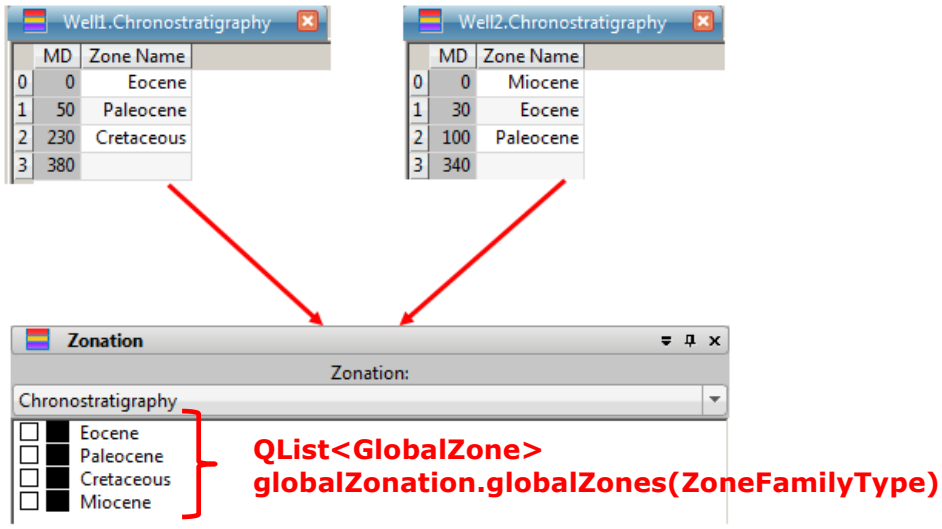


Figure 1-15 Zonation datasets with same name merge in one Zonation

Each zone in the **Zonation** window is a **`GlobalZone`** object. Retrieve the **`GlobalZones`** with a given family type that belong to a **`GlobalZonation`** object using the **`globalZones`** function.

```
class GlobalZonation
{
public:
  bool isEmpty() const;
  QString name() const;
  QList<GlobalZone> globalZones(ZoneFamilyType &zoneFamilyType)
    const;
  GlobalZone findGlobalZone(const QString &zoneName,
    const ZoneFamily zoneFamily) const;
};
```

A **`GlobalZonation`** may also contain a list of zones from different wells (same interval dataset name in different wells) that have been created with different zone families.

See the "Zonation dataset" section on page 1-40 for more information on how to create an interval dataset with different zone families.

In this example, the "MyZonation" interval dataset is created in two differents wells. "Well1" contains zones with **`zoneFamilyName`** type and "Well2" contains zones with **`zoneFamilyHydraulic`** type.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

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

Well well1 = Well::create("Well1", project);

Dataset wellZonation1 = Dataset::create("MyZonation", "MD", "m",
VariableDataFormatFloat, well1);

QList<Zone> zones1;
zones1 << Zone("ZoneA", 0.f, 50.f);
zones1 << Zone("ZoneB", 50.f, 150.f);
zones1 << Zone("ZoneC", 150.f, 230.f);

wellZonation1.setZones(zones1,
ZoneFamilyType::zoneFamilyName());

Well well2 = Well::create("Well2", project);

Dataset wellZonation2 = Dataset::create("MyZonation", "MD", "m",
VariableDataFormatFloat, well2);

QList<Zone> zones2;
zones2 << Zone("ZoneD", 0.f, 50.f);
zones2 << Zone("ZoneE", 50.f, 150.f);
zones2 << Zone("ZoneF", 150.f, 230.f);
wellZonation2.setZones(zones2,
ZoneFamilyType::zoneFamilyHydraulic());
lock.release();
```

This screenshot shows that zones with different families are not listed together in the zone pane of the **Zonation** dialog window.

If the **`GlobalZonation`** selected in the drop-down contains zones with **`zoneFamilyName`** type, they are displayed by default.

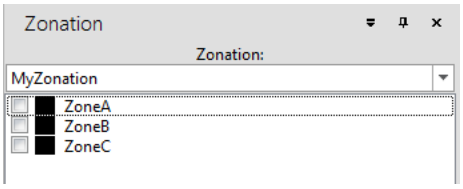


Figure 1-16 Zones with family "Zone name"

But this display can be changed after the fact in the **Zone family selection** dialog accessible through the Techlog zonation dialog **Zone family...** context menu item.

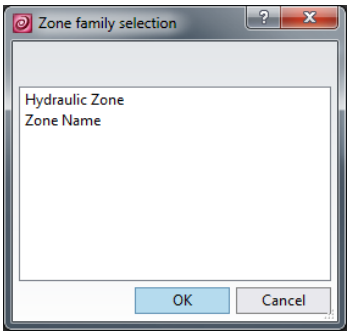


Figure 1-17 Zone family dialog window

`GlobalZonation::globalZones` function returns the list of zones by zone family type. In this example, in the GlobalZonation "MyZonation" only the zones with zone family type set to hydraulic are colored with a blue gradient.

```
lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, project);

ProjectZonationModel zonation = project.zonationModel();
QString globalZonationName = "MyZonation";
GlobalZonation globalZonation =
zonation.findGlobalZonation(globalZonationName);

int cpt;
cpt = 0;
foreach (GlobalZone globalZone,
globalZonation.globalZones(ZoneFamilyType::zoneFamilyHydraulic
()))
{
  if (cpt == 0)
    zonation.setGlobalZoneColor(globalZonationName,
globalZone.name(),
    QColor(0, 255, 255));
  else if (cpt == 1)
    zonation.setGlobalZoneColor(globalZonationName,
globalZone.name(),
    QColor(0, 128, 255));
  else if (cpt == 2)
    zonation.setGlobalZoneColor(globalZonationName,
globalZone.name(),
    QColor(0, 0, 255));

  cpt++;
}

lock.release();
```

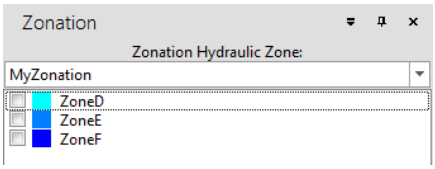


Figure 1-18 Hydraulic zones

GlobalZone object

Retrieve a **GlobalZone** object from the parent **GlobalZonation** and change it from the **ProjectZonationModel**.

A **GlobalZone** object represents properties of an interval. A **GlobalZone** object provides read accessors to name, color, pattern and zone family of the zone.

```
class GlobalZone
{
public:
  bool isEmpty() const;
  QString name() const;
  ZoneFamily zoneFamily() const;
  QString patternName() const;
  StorageLevel patternStorageLevel() const;
  QColor color() const;
};
```

Color and pattern properties are only set through the **ProjectZonationModel** object returned by the project (see the "ProjectZonationModel object" section on page 1-42).

This example shows how to set colors and patterns on **GlobalZone** objects; the "Zone name" family listed from two differents **GlobalZonation** objects "Chronostratigraphy" and "Lithology".

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
// Get the main project of the current session
Project project = Session::current().mainProject();

QString zonationName = "Chronostratigraphy";
ProjectZonationModel zonation = project.zonationModel();
GlobalZonation chronostratigraphy =
zonation.findGlobalZonation(zonationName);

// for each zone in Chronostratigraphy zonation set a color
foreach (GlobalZone zone,
chronostratigraphy.globalZones(ZoneFamilyType::zoneFamilyName(
)))
{
  if (zone.name() == "Cretaceous")
```

```
    zonation.setGlobalZoneColor(zonationName, zone.name(),
Qt::green);
  else if (zone.name() == "Paleocene")
    zonation.setGlobalZoneColor(zonationName, zone.name(),
Qt::red);
  else if (zone.name() == "Eocene")
    zonation.setGlobalZoneColor(zonationName, zone.name(),
    Qt::darkYellow);
  else if (zone.name() == "Miocene")
    zonation.setGlobalZoneColor(zonationName, zone.name(),
Qt::yellow);
}

GlobalZonation lithology =
zonation.findGlobalZonation("Lithology");
// for each zone in Lithology zonation set a pattern
// here the pattern match with the zone description name
foreach (GlobalZone zone,
lithology.globalZones(ZoneFamilyType::zoneFamilyName()))
{
  QString patternName = zone.name();
  zonation.setGlobalZonePattern("Lithology", zone.name(),
patternName,
  StorageLevelTechlog);
}

lock.release();
```

See the result expected in Techlog for the two zonations "Chronostratigraphy" and "Lithology":

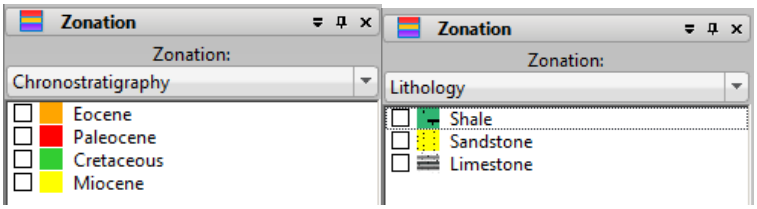


Figure 1-19 GlobalZone color and pattern properties

Note: All the pattern names are available in Techlog from the **Patterns** manager.

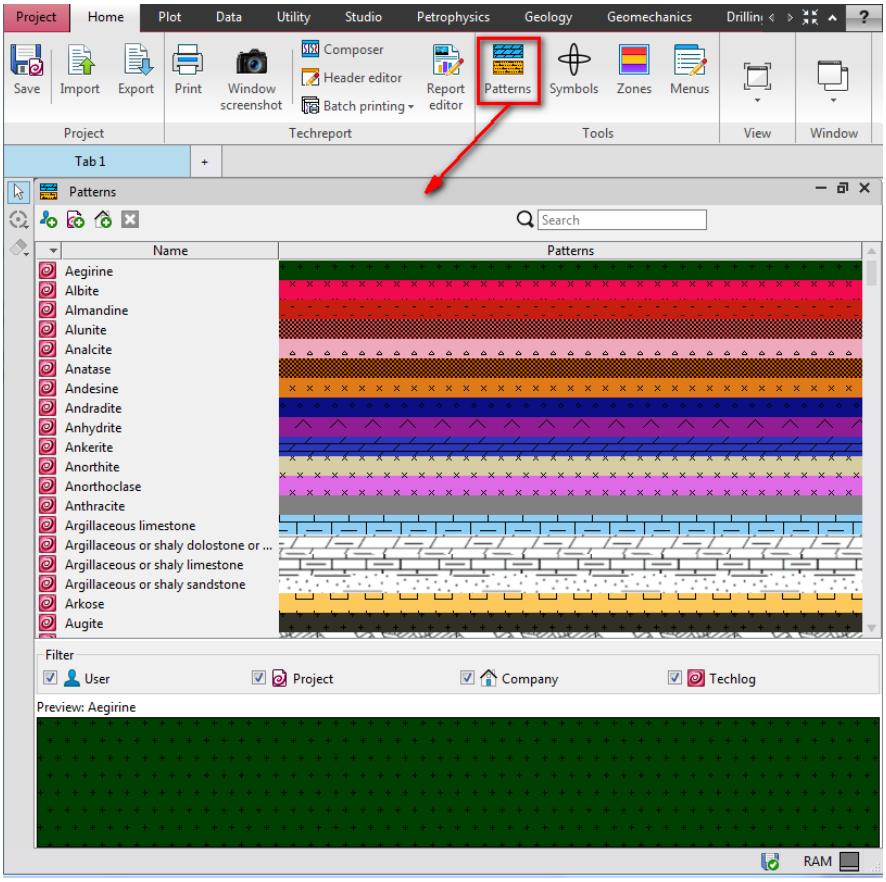


Figure 1-20 Techlog Patterns manager

Marker creation

Markers are handled in Techlog very similarly to the way zonations are handled. You create them in an equivalent way to zonations as well.

Marker dataset

Create the Marker dataset using the **`Dataset`** domain object.

See the "Data creation" section on page 1-4 for more information on dataset creation.

However there is no dedicated **`Dataset::create`** function that allows you to create the marker dataset variables structure as there is for zonation dataset. This means that you must create a dataset yourself with marker variables as follows:

- a dataset of type "markers" (**`DatasetTypeMarker`** enum value)
- a reference variable with float or double format
- a string variable with family "Marker Name"

As like all other datasets, the marker name is used as a unique identifier within a well, so you must always check for the existence of a marker dataset with a particular name in the same well before creating a new one.

This example shows how to create a valid marker dataset:

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

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

if (!well.findDataset("Depositional Events").isNull())
{
  lock.release();
  return;
}

Dataset dataset = Dataset::create("Depositional Events", "MD",
VariableDataFormatFloat, 3, well);
dataset.setType(DatasetTypeMarker);

Variable reference = dataset.findReferenceVariable();
reference.setFamily("Measured Depth");
reference.setUnit("ft");
reference.setFloatValue(0, 7315);
reference.setFloatValue(1, 7432);
reference.setFloatValue(2, 7907);

Variable markerName = Variable::create("MARKER_NAME", dataset,
VariableDataFormatString, VariableTypeText, 1);
markerName.setFamily("Marker Name");
markerName.setStringValue(0, "Turbidites");
markerName.setStringValue(1, "Sea-level change");
markerName.setStringValue(2, "Continental deposits");
lock.release();
```

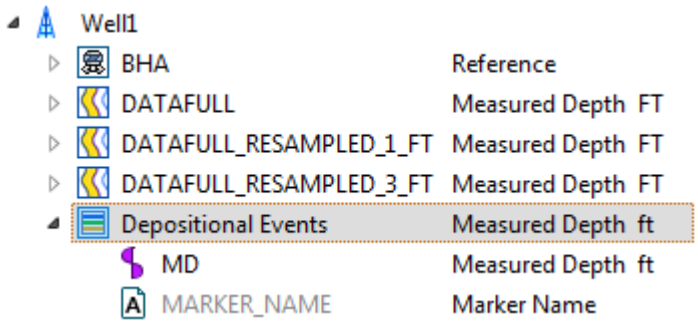


Figure 1-21 Marker dataset

ProjectMarkerModel object

Instantiate the **`ProjectMarkerModel`** object using the **`Project::markerModel`** function. The **`ProjectMarkerModel`** object has functionality to retrieve a **`GlobalMarkerSet`** object in the Techlog project or to modify **`GlobalMarkerSet`** color property.

See the "Project Domain Object" section in the *Ocean for Techlog Developer Guide - Basics* for more information on accessing Techlog project with Ocean.

See the "GlobalMarkerSet object" section on page 1-50 for more information on **`GlobaMarkerSet`**.

```
class ProjectMarkerModel
{
public:
  QList<GlobalMarkerSet> globalMarkerSets() const;
  GlobalMarkerSet findGlobalMarkerSet(const QString
    &markerSetName) const;
  void setGlobalMarkerColor(const QString &markerSetName,
    const QString &markerName, const QColor &color);
};
```

GlobalMarkerSet object

A **`GlobalMarkerSet`** can't be created programmatically. As with zonation, a **`GlobalMarkerSet`** is the merge of all the marker **`Datasets`** with the same name in different wells.

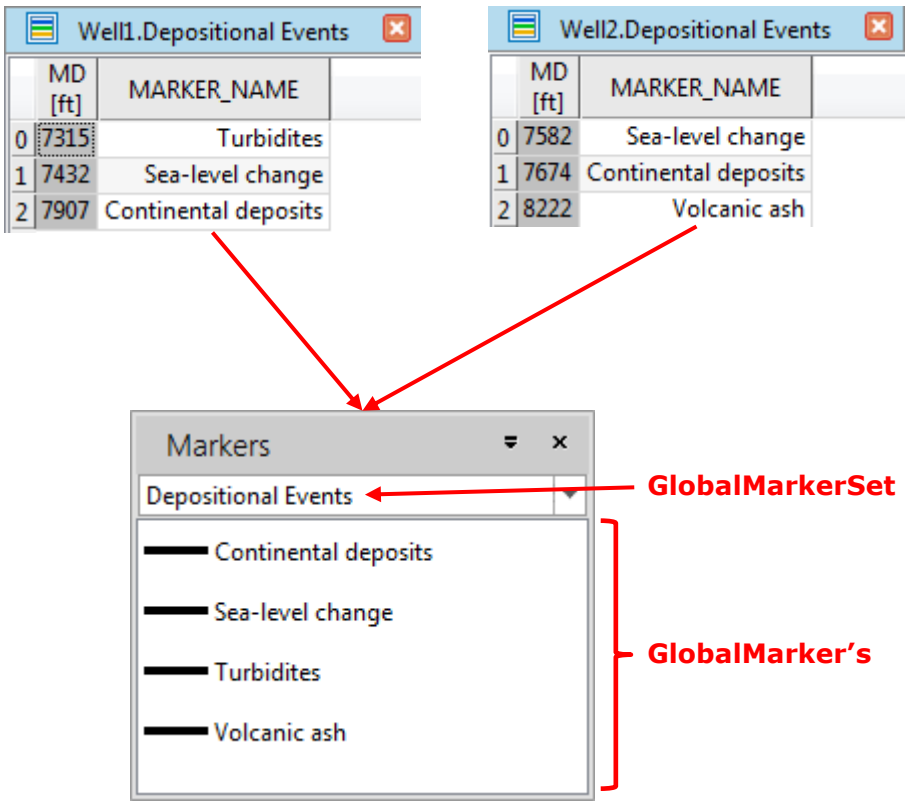


Figure 1-22 Marker datasets with same name merge in one set of markers

Each marker in the set of markers is modeled as a **`GlobalMarker`** object. Find **`GlobalMarker`** belonging to a **`GlobalMarkerSet`** object using the **`globalMarkerSets`** function.

```
class GlobalMarkerSet
{
public:
  bool isEmpty() const;
  QString name() const;
  Qlist<GlobalMarker> globalMarkers() const;
  GlobalMarker findGlobalMarker(const QString &markerName)
   const;
};
```

GlobalMarker object

Retrieve a **`GlobalMarker`** object from the parent **`GlobalMarkerSet`** and modify it from the **`ProjectMarkerModel`**.

The **`GlobalMarker`** object provides read accessors to name and color properties of the marker.

```
class GlobalMarker
```

```
{
public:

  bool isEmpty() const;
  QString name() const;
  QColor color() const;
};
```

The color property is set through the **`ProjectMarkerModel`** object returned by the project (see the "ProjectMarkerModel object" section on page 1-50).

This example shows how to set colors on **`GlobalMarker`** objects for the **`GlobalMarkerSet`** "Depositional Events".

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

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

ProjectMarkerModel  projectMarkerModel = project.markerModel();

GlobalMarkerSet globalMarkerSet =
projectMarkerModel.findGlobalMarkerSet("Depositional Events");
if (globalMarkerSet.isEmpty())
{
  lock.release();
  return;
}

if (!globalMarkerSet.findGlobalMarker("Continental deposits")
.isEmpty())
  projectMarkerModel.setGlobalMarkerColor("Depositional Events",
  "Continental deposits", Qt::yellow);

if (!globalMarkerSet.findGlobalMarker("Sea-level change")
.isEmpty())
  projectMarkerModel.setGlobalMarkerColor("Depositional Events",
  "Sea-level change", Qt::blue);

if (!globalMarkerSet.findGlobalMarker("Turbidites")
.isEmpty())
  projectMarkerModel.setGlobalMarkerColor("Depositional Events",
  "Turbidites", Qt::red);

lock.release();
```

See how the color property was modified for each marker in the Techlog **Markers** dock window for the "Depositional Events" set of markers.

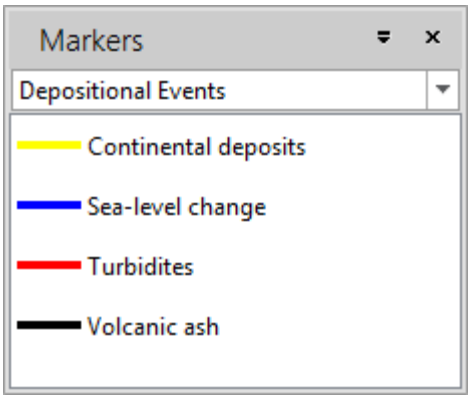


Figure 1-23 GlobalMarker color property

Data deletion

Delete an object from the object itself. All domain objects provide an **`erase`** method inherited from the **`DomainObject`** base class. The action is equivalent to the **Delete definitely...** menu entry in the object's context menu in the Techlog data tree.

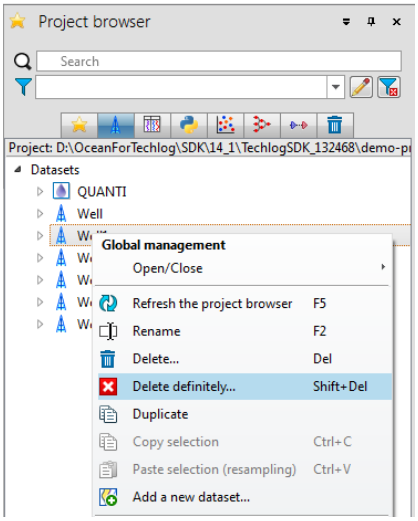


Figure 1-24 Popup menu showing Delete definitely item

The method signature is the same across domains. All the data domain objects (**`DataDomainObject`** derived classes) support being erased. Verify it through **`supportsErase`** function.

```
class DomainObject : public IPrintable
{
public:
    ...
    const bool isErased() const;
    void erase();
    bool supportsErase() const;
};
```

Deleting a parent deletes all its children. The locking parent rule applies also to deleting a domain object.

Deleting a variable which is the reference of a dataset throws an exception. The **`isReference`** method allows you to get this information from the **`Variable`** object.

```
class Variable : public DataDomainObject
{
public:
  ...
  bool isReference() const;
  bool isTemporary() const;
};
```

This example shows checking if a variable is a reference.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dataset);
if (!variable.isReference())
  variable.erase();
lock.release();
```

Deleting a temporary variable does not throw an exception but it is useless. The **`isTemporary`** method allows you to get this information from the **`Variable`** object.

Variable resampling

Resampling modifies the sampling rate of a given curve and defines the interval in which resampling is used.

In Techlog, if you drag a variable from one dataset (with a given sampling rate) to another dataset (with a different sampling rate), resampling is needed. Resampling with the drag operation only works if there is a common reference between the two datasets.

Resampling has several methods to adjust an existing variable data to a new dataset, with a different reference. Those methods compute a corresponding value at a given index that doesn't exist in the source dataset.

Ocean provides an easy to use resampling API under **`Slb::Ocean::Techlog::VariableResampler`** namespace.

The first function uses the default resampling based on the source variable; you only need to pass the source variable and the destination dataset to the function.

```
namespace Slb {
  namespace Ocean {
    namespace Techlog {
      namespace VariableResampler {
        Variable tryResampleVariable(const Variable
          &sourceVariable, const Dataset &destinationDataset);
        ...
} } } }
```

A temporary **`Variable`**, which is the result of interpolating the source variable is created in the destination **`Dataset`**; it has the same name as the source **`Variable`**. A null **`Variable`** is returned if **`tryResampleVariable`** fails. Use this function to perform resampling in a single line of code:

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

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

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

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

Dataset srcDataset = well.getDataset("DATAFULL");
Variable srcVariable = srcDataset.getVariable("GR");
Dataset destDataset = well.getDataset("LQC");

QList<ResamplingMethod> compatibleMethods =
VariableResampler::getCompatibleResamplingMethods(srcVariable,
destDataset);

foreach(const ResamplingMethod method, compatibleMethods)
{
  ResamplingOptions options;
  options.setResamplingMethod(method);
  options.setPersistenceMode(PersistenceModePersisted);
  options.setMaximumGapSize(Value(5.0, "ft"));
  options.setWindowSize(Value(5.0, "ft"));
  options.setVariableCopyOption(VariableCopyOptionFull);

  Variable varResampledWithOptions =
  VariableResampler::tryResampleVariable(srcVariable,
destDataset,
  QLatin1String("GR RESAMPLED WITH ") +
ArgChecker::toString(method),
  options);
}

lock.release();
```

The resampling method used is determined by the source variable as follows:

- **`ResamplingMethodLinear`** - Standard method for a continuous log. Applies a linear interpolation between two points.
- **`ResamplingMethodPointDataShift`** - Standard method for text, flags, most arrays, point data, etc. This is the only valid method for Images. Moves data to the nearest depth. The shift (distance between source and destination reference) is limited to the window size.
- **`ResamplingMethodInterval`** - Standard method for interval dataset variables.

- **`ResamplingMethodLinearForAngle`** - Standard method for a continuous log with angle units. Applies a linear interpolation with the range of 0-360.

The helper functions **`getDefaultResamplingMethod`** and **`getCompatibleResamplingMethods`** provide the default **`ResamplingMethod`** and the list of compatible **`ResamplingMethod`** for a source **`Variable`** and the destination **`Dataset`**.

```
namespace Slb {
  namespace Ocean {
    namespace Techlog {
      namespace VariableResampler {

       ResamplingMethod getDefaultResamplingMethod(const
         Variable &sourceVariable, const Dataset
         &destinationDataset);

       QList<ResamplingMethod> getCompatibleResamplingMethods
         (const Variable &sourceVariable, const Dataset
         &destinationDataset);
       ...
} } } }
```

The second function allows you to set some resampling options including the resamping method to use.

```
namespace Slb {
  namespace Ocean {
    namespace Techlog {
      namespace VariableResampler {
        Variable tryResampleVariable(const Variable
          &sourceVariable, const Dataset &destinationDataset,
           const QString &destinationVariableName, const
           ResamplingOptions &options);
       ...
} } } }
```

A **`Variable`**, which is the result of interpolating the source variable is created in the destination **`Dataset`**; it has the destination variable name passed to the function. A null **`Variable`** is returned if **`tryResampleVariable`** fails.

`ResamplingOptions` class represents the resampling options.

```
class Variable : public DataDomainObject
{
public:
  Value maximumGapSize() const;
  void setMaximumGapSize(const Value &maximumGapSize);

  Value windowSize() const;
  void setWindowSize(const Value &windowSize);
  ResamplingMethod resamplingMethod() const;
```

```
  void setResamplingMethod(const ResamplingMethod
    resamplingMethod);
  PersistenceMode persistenceMode() const;
  void setPersistenceMode(const PersistenceMode
    persistenceMode);
  VariableCopyOption variableCopyOption() const;
  void setVariableCopyOption(const VariableCopyOption
    variableCopyOption);
};
```

The **ResamplingOptions** class allows you to:

- get and set the maximum gap threshold (only used for **ResamplingMethodLinear**)
- get and set the window size; used for the methods other than **ResamplingMethodLinear** and **ResamplingMethodLinearForAngle**
- get and set the resampling method to use through **ResamplingMethod** enum values. The default **ResamplingMethod** is **ResamplingMethodDefault**. If the **ResamplingMethod** is set to **ResamplingMethodDefault** when the **tryResampleVariable** function is called, the system tries to determine from the source Variable and the destination Dataset the most appropriate resampling method to use. If the **ResamplingMethod** is changed, the **tryResampleVariable** function may fail if it cannot resample the source **Variable** into the destination **Dataset** with this resampling method.
- get and set the **PersistenceMode** (Persisted, Temporary) to define the computed **Variable** persistence mode
- get and set **VariableCopyOption** (simple, full) to define what the destination variable inherits from source variable
  - **VariableCopyOptionSimple** - copies only data, and ignores data properties, group information and history
  - **VariableCopyOptionFull** - copies data, data properties, group information and history

This example shows setting some resampling options.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

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

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

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

Dataset srcDataset = well.getDataset("DATAFULL");
Variable srcVariable = srcDataset.getVariable("GR");
```

```
Dataset destDataset = well.getDataset("LQC");

QList<ResamplingMethod> compatibleMethods =
VariableResampler::getCompatibleResamplingMethods(srcVariable,
destDataset);

foreach(const ResamplingMethod method, compatibleMethods)
{
  ResamplingOptions options;
  options.setResamplingMethod(method);
  options.setPersistenceMode(PersistenceModePersisted);
  options.setMaximumGapSize(Value(5.0, "ft"));
  options.setWindowSize(Value(5.0, "ft"));
  options.setVariableCopyOption(VariableCopyOptionFull);

  Variable varResampledWithOptions =
  VariableResampler::tryResampleVariable(srcVariable,
destDataset,
  QLatin1String("GR RESAMPLED WITH ") +
ArgChecker::toString(method),
  options);
}

lock.release();
```

Variable background data access

It is possible to avoid blocking the Techlog GUI while reading/writing **`Variable`** data. This is done automatically by Ocean when variables are locked in a separate **`Lock`**. Then variable data are accessed on a worker thread handled by Ocean.

```
Dataset dataset = …
Variable ref = dataset.findReferenceVariable();
Variable gr = dataset.getVariable("GR");
Lock lock = LOCK_CREATE();
lock.add(dataset);
LOCK_ACQUIRE_OR_RETURN(lock);
for (int j = 0 ; j < 1000; j++)
{
  ref.setFloatValue(j, 0, j);
  gr.setFloatValue(j, 0, j+1);
}
lock.release();
```

```
…
Lock lock = LOCK_CREATE();
lock.add(ref);
lock.add(gr);
LOCK_ACQUIRE_OR_RETURN(lock);
for (int j = 0 ; j < 1000; j++)
{
  ref.setFloatValue(j, 0, j);
  gr.setFloatValue(j, 0, j+1);
}
lock.release();
```

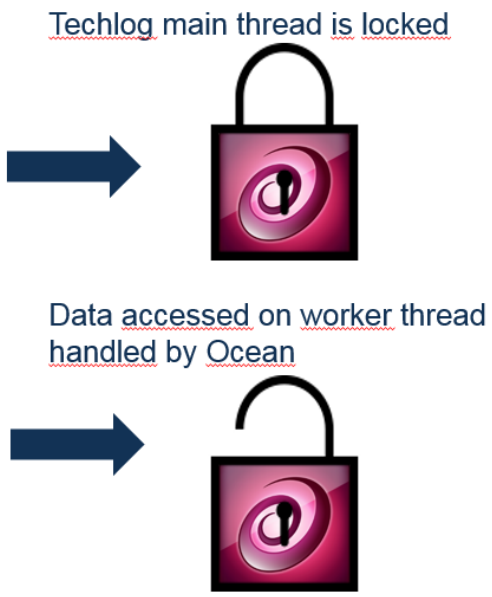


Figure 1-25 Variable background data access

2 Workflow and worksteps

In This Chapter

Overview

The Techlog workflow refers to the general steps and tasks to complete work within Techlog. A workflow in Techlog is a processing chain in which each processing element is called a workstep. Workflows and worksteps are accessible in Techlog through the Application Workflow Interface (AWI).

The Ocean for Techlog workflow API allows you to create a workflow with worksteps implementing processing logic.

This chapter shows how to create a workflow and a workstep using the Ocean for Techlog API. This guide does not cover user usage of the AWI. See the Techlog online help for detailed end-user instructions.

A workstep in Techlog is one algorithmic data processing step in a sequence of steps. A workflow is the sequence of steps.

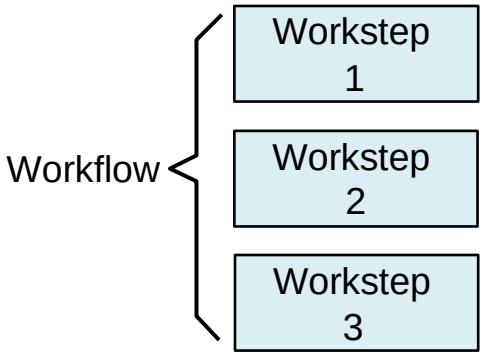


Figure 2-1 Workflow and Workstep Concepts

Each workstep takes input data, transforms or manipulates it and creates output data. The data may be chained through the workflow so that the output of one step is the input to a following step. You must run the workstep computations one by one for the next workstep to bind the output variable generated by the previous workstep as an input variable.

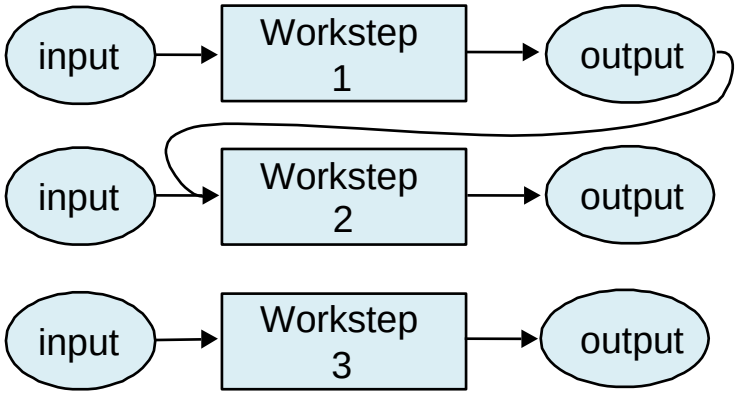


Figure 2-2 Chained Workstep Data in Workflow

Workstep

A workstep can be any data processing operation. A workstep tends to be an operation that can participate in different workflows.

Techlog is made of modules containing worksteps for common wellbore centric tasks like **Borehole computation** or **Shale volume** in the **Petrophysics > Quanti** module.

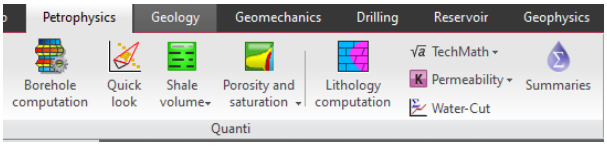


Figure 2-3 Native Techlog Worksteps

Worksteps are part of the Techlog application and are dependent on the data available in a project.

Workflow

A workflow is a processing step or a sequence of processing steps that perform a task within Techlog. Workflows help the end-users to perform their jobs correctly and efficiently. They automate time-consuming or specialized processing; the user can easily repeat standard processing on multiple domain objects or projects.

A workflow is built in Techlog using worksteps in the Application Workflow Interface (AWI). The native Techlog worksteps are available through the different domain tab menus accessible in the Techlog ribbon (for example: Petrophysics, Geology, or Drilling).

Combining the Techlog worksteps into a workflow makes it easy to apply them and guarantees that they run in the right order with the correct data every time. For instance, a workstep generates the output data that is used as an input by the next one.

Since worksteps are the building blocks of workflows, extending Techlog with your custom worksteps significantly enhances the end users' processing capabilities.

Workflows contain any combination of native Techlog and custom worksteps. The user manipulates a custom workstep just like a native Techlog workstep.

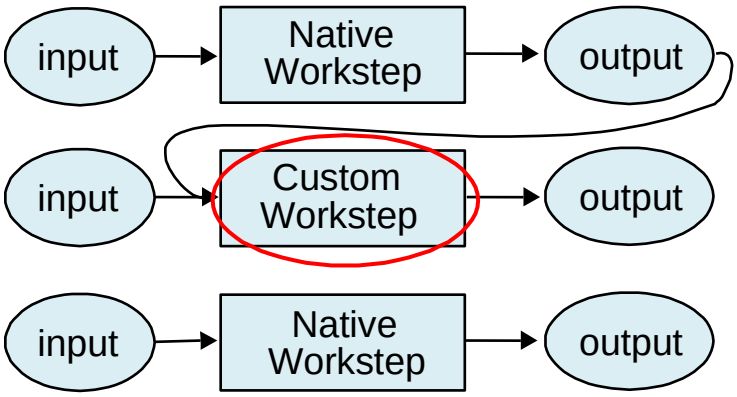


Figure 2-4 Chained Workstep Data in Workflow

Techlog does not supply any pre-defined workflows. Workflows are based on the available worksteps (Techlog modules) and data and the end users' business needs; they are inherently custom operations.

Workflows are specific to a Techlog project. All workflows are built within the context of a project and its data. Workflows in the project are available in the **Workflows** tree of the project browser.

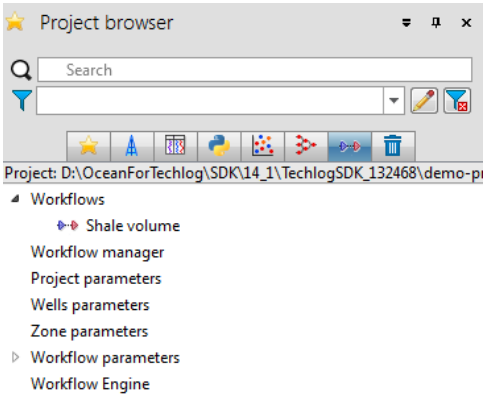


Figure 2-5 Workflows in the Project

Starting the Workflow manager

The **Workflow manager** is a user interface in Techlog that allows you to create, edit, run, and save workflows.

1. Select the method to open the corresponding **Application Workflow Interface** (for example: **Petrophysics > Quanti** > **Shale Volume** calculation).
2. Select the type of data (families, aliases or variables) to use.
   - The left pane lists all the wells in the project browser.
   - The right pane lists all the families, aliases and variables.

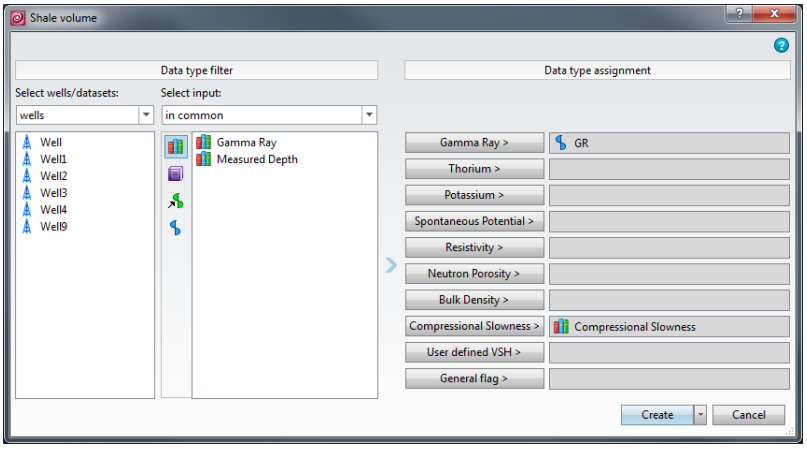


Figure 2-6 Input selection window

3. Click **Create** to launch the **Workflow manager**.
4. Drag the datasets in the **Workflow manager** right pane. In the **Inputs** tab, each line corresponds to a dataset. The group name, well name and dataset name are automatically filled in.

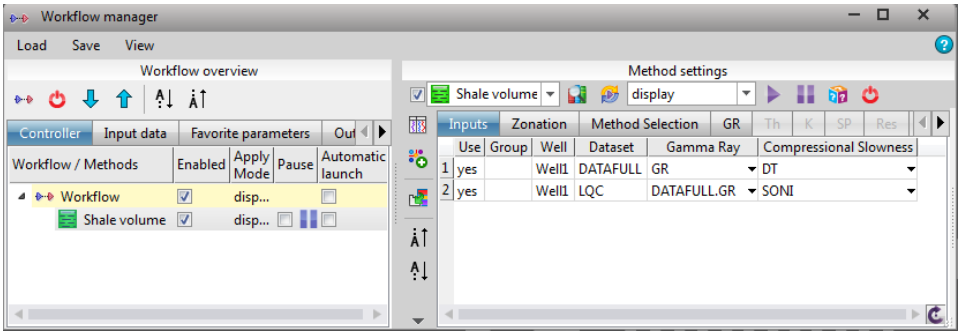


Figure 2-7 Workflow manager

Note: You can only launch one workflow in a given working space and you can only use one set of zones for a workflow. This means that you cannot launch different intervals for different methods in the same workflow. When a zone is disabled for a method, it is disabled for all the methods within the workflow.

Workflow manager description

The **Controller** tab in the left pane of the **Workflow manager** shows all workflows and methods (worksteps) in a tree view. You create new workflows in this window and a new group is displayed as a new branch of the tree. Each workflow is independent and has its own dataset, zonation and parameters. When you open a method, it is automatically added to the selected workflow. If there is no workflow, a new workflow is automatically created and the method is added to this new workflow which is now the selected workflow.

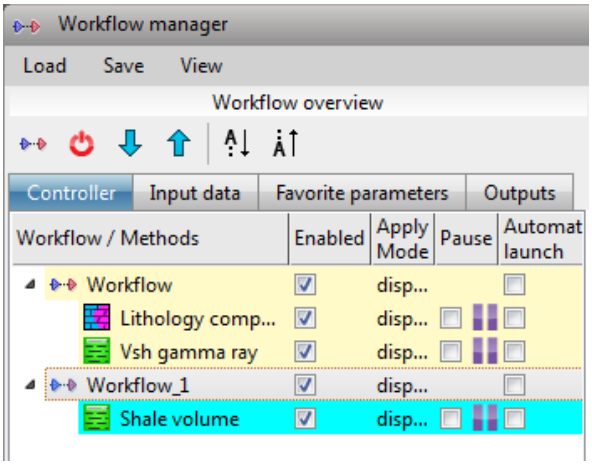


Figure 2-8 Controller tab

The **Input data** tab is a summary of all inputs for displayed by workflow.

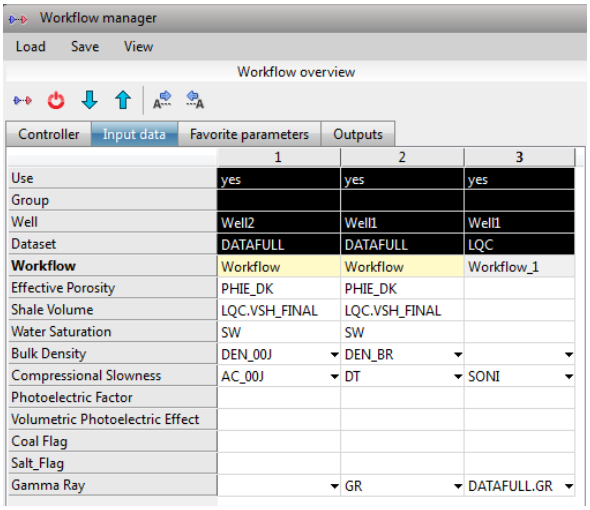


Figure 2-9 Input data tab

The **Outputs** tab shows a summary table of all outputs for the selected workflow. You can enable and disable outputs rename, family changes and edit the output name.

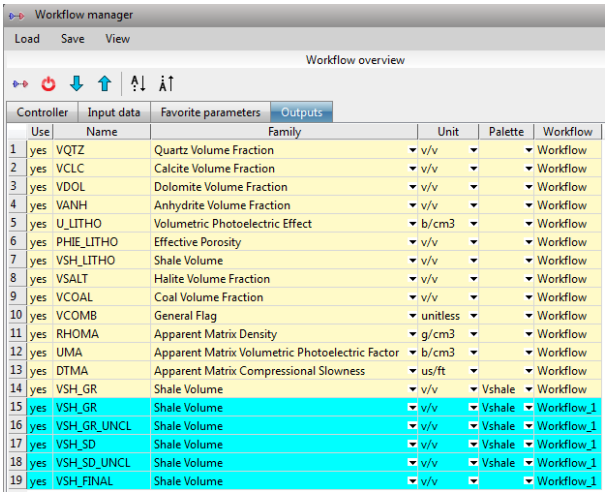


Figure 2-10 Outputs tab

The right pane of the **Workflow manager** contains the data context on which a workstep is run. It has three tabs:

- **Inputs**: Wells, datasets and curves
- **Zonation**: Defines the zones for each dataset used for the calculation
- Parameters: Additional equation parameters

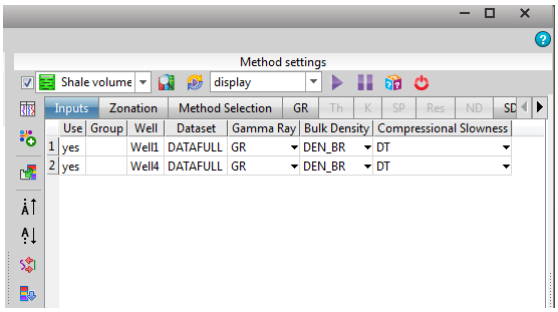


Figure 2-11 Data tabs

Plugin worksteps

The Ocean API enables you to create and add custom plug-in worksteps into Techlog's **Workflow manager**. A workstep is an operation which accepts user input, executes algorithms and produces results. Plug-in worksteps created with the Ocean API are displayed alongside native Techlog worksteps in the **Workflow manager**.

Steps to write a custom workstep

In the `AbstractActivity::run` method of the plug-in activity:

1. Find the selected `Workflow` from the current `Workspace`.
2. If the selected `Workflow` is null then create a new `Workflow`.
3. Check for the existence of the `Workstep` name within the selected `Workflow`. A workstep name must be unique in the workflow.
4. Create the `Workstep` using the static `Workstep::create` method, passing the workstep name and the parent workflow as parameters.
5. Add the workstep arguments (input, parameter, output).
6. The new `Workstep` must subscribe to the signals that the workstep activity needs to be notified about, like when:
    - new dataset is dropped in the AWI
    - workstep arguments are changed
    - workstep computation is run
    - workstep computation is done
7. Implement the corresponding slots of the signals
8. Implement the `onCompute` slot method which runs the workstep processing.
9. Override the `saveWorkstep` and `restoreWorkstep` virtual methods of the `AbstractActivity` class; this allows a workflow which contains a custom workstep to be saved and restored in the Techlog project.

The following sections show an example of a custom workstep that computes a gamma ray average value per zone from gamma ray log values. From the average value, the workstep determines the lithology of the zone. This is a simple computation but it clearly demonstrates that Ocean gives you the ability to have a custom workstep with an algorithm currently unavailable within Techlog.

Workstep creation

Worksteps are created at the workflow level. A workflow is modeled in Ocean for Techlog with the `Workflow` class and a workstep with the `Workstep` class. You access the `Workstep` collection from a `Workflow` object using the `pluginWorksteps` method. This method returns only plug-in worksteps created through Ocean for Techlog.

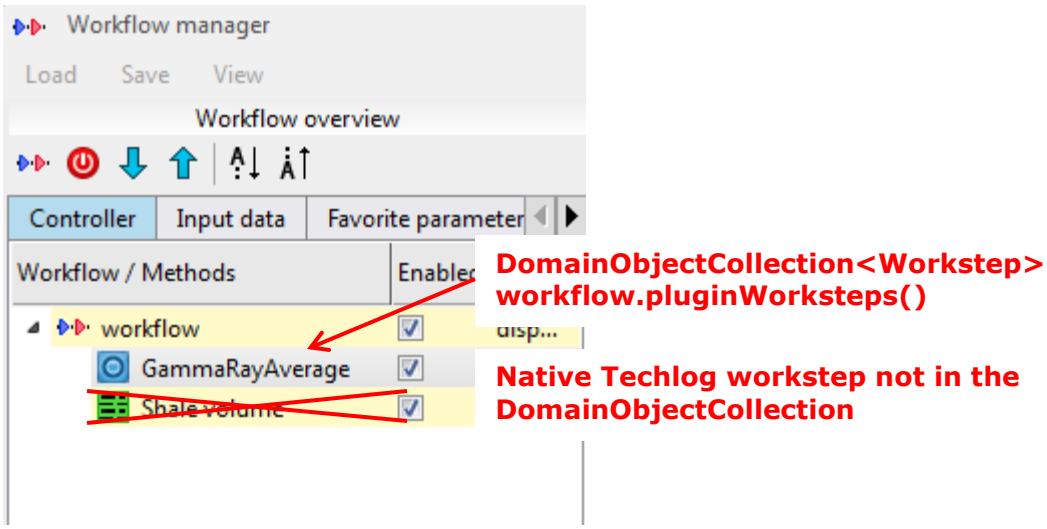


Figure 2-12 List of plug-in worksteps

Workflow domain object

The first step in the **`run`** method of your activity is to find the selected workflow in the Techlog **Workflow manager** and create your custom workstep in this workflow.

A workflow belongs to the workspace; you find the selected workflow by calling the **`findSelectedWorkflow`** method from the current workspace instance. If there is no selected workflow the method returns null. This means that there is no workflow available in the **Workflow manager**. In this case, you must create the workflow using the **`Workflow:create`** pattern. Once created, the new workflow is the selected workflow.

```
class Workflow : public DomainObject
{
public:
  static Workflow create(const QString & name, Workspace
     workspace);
  const DomainObjectCollection<Workstep>  pluginWorksteps()
const;
  void doAddDataset(const Dataset &dataset);
  void removeDatasets(const QList<Dataset> &datasets);
  Workstep findWorkstep(const QString &workstepName) const;
  Workstep getWorkstep(const QString &workstepName) const;
  const Workspace workspace() const;
  ...
};
```

The workflow name and workspace parent instance are the first and second arguments of the **`Workflow::create`** static method, respectively.

This example creates the workflow.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
Workspace workspace = Session::current().currentWorkspace();
// find the selected workflow from the current workspace activity
```

```
Workflow workflow = workspace.findSelectedWorkflow();
if (workflow.isNull())
  workflow = Workflow::create("workflow", workspace);
lock.release();
```

Find all plug-in **Worksteps** belonging to a **Workflow** object using the **pluginWorksteps** function. Retrieve a **Workstep** by its name using the **findWorkstep** or **getWorkstep** functions. See the "Find and get patterns" section for details on the differences in *Ocean for Techlog Developer Guide – Basics.*

The datasets are added to the AWI at the workflow level; this means that all the worksteps of the same workflow share the same list of datasets. Add and remove datasets using the **doAddDataset** and **removeDatasets** functions.

Workstep domain object

Once you have a valid workflow instance, add a new workstep to it using the **Workstep::create** pattern.

```
class Workstep : public DomainObject
{
public:
  static Workstep create(const QString &name,
    WorkstepProcessingScope processingScope, Workflow workflow);
  static Workstep create(const QString &name, const
    WorkstepProcessingScope processingScope, const QImage
    &image, Workflow workflow);
  WorkstepProcessingScope processingScope() const;
  Workflow workflow() const;

  void setApplyMode(const WorkstepApplyMode applyMode);
  WorksepApplyMode applyMode()const;
  bool isApplyModeDisplay() const;
  bool isApplyModeSave() const;

  Logview findDefaultLogview() const;
  QList<LogviewTemplate> availableLogviewTemplates() const;
  LogviewTemplate logviewTemplate() const;
  LogviewTemplateState logviewTemplateState() const;
  void setLogviewTemplate(const LogviewTemplate
       &logviewTemplate);
  void setLogviewTemplateState(const LogviewTemplateState
       &logviewTemplateState);

  void setAWI4SelectionWindowDisplayed(const bool show);
  bool isAWI4SelectionWindowDisplayed() const;
  void addCustomWidgetInTab(QWidget *widget, const QString
       &tabName);
  void setHelpId(const QString &guid);
```

```
  QString helpId() const;
  ...
};
```

The workstep name is used as a unique identifier within a workflow; you must check for the existence of a workstep with a given name in the same workflow before creating a new one.

The wokstep name is the first argument of the **`Workstep::create`** static method.

The processing scope is passed as second argument of the **`create`** method. The processing scope property is immutable; read it from the **`Workstep`** object using the **`processingScope`** public method. **`WorkstepProcessingScope`** enum has the following values:

```
enum WorkstepProcessingScope
{
  WorkstepProcessingScopeDataset,
  WorkstepProcessingScopeZone
};
```

A workstep belongs to a workflow. A workflow parent instance must be passed to the **`Workstep::create`** static method.

In the overloaded **`create`** static function you may pass a **`QImage`** to display an image next to the **`Workstep`** in the **`Workflow`**; the default image is the Ocean icon.

`create` returns a **`Workstep`** object with a scope of a zone or dataset.

The **`WorkstepApplyMode`** enum controls the apply mode of the workstep.

```
enum WorkstepApplyMode
{
  WorkstepApplyModeDisplayOnly,
  WorkstepApplyModeSaveOnly,
  WorkstepApplyModeSaveAndDisplay
};
```

The following public API allows you to change the workstep display options:

- **`setApplyMode`** – toggles the apply mode between "Display", "Save" or "Save and Display" through the **`WorkstepApplyMode`** enum
- **`isApplyModeDisplay`** and **`isApplyModeSave`** **-** convenience methods to get the **`applyMode`** state
- **`findDefaultLogview`** - retrieves the default AWI **`Logview`** instance (null if **`isApplyModeDisplay`** is false or if the logview has been closed)
- **`setAWI4SelectionWindowDisplayed`** – shows the family selection window when the workstep is created. The default value is false and the selection window is not displayed.
- **`setLogviewTemplate`** –applies a user template to the AWI Logview that opens at the end of the computation with **`isApplyModeDisplay`** equals to true. This function throws a plug-in exception if the layout template is stored at the plug-in level. The **`logviewTemplate`** can be retrieved only if **`logviewTemplateState`** is equal to

LogviewTemplateStateUserDefined and the layout template is contained into **availableLogviewTemplates** list.

- **setHelpId** - links the **Workstep** with Techlog help content (HTML page) deployed at the plug-in storage level (plug-in folder)

See "How to add plug-in documentation to Techlog Help Center" tutorial in the **OceanForTechlog.chm** file for more information on help.

This example shows the Gamma Ray Average computation workstep creation and sets some display options:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();
// find the selected workflow from the current workspace activity
Workflow workflow = workspace.findSelectedWorkflow();
if (workflow.isNull())
  workflow = Workflow::create("workflow", workspace);

// Check for existence of the workstep name within the workflow
if (workflow.pluginWorksteps().contains("GammaRayAverage"))
{
  qWarning() << "The workflow already contains the gamma ray
average computation workstep";
  lock.release();
  stop();
  return;
}

// create a workstep with a processing scope per zone
Workstep workstep = Workstep::create("GammaRayAverage",
WorkstepProcessingScopeZone, workflow);
// save and display results in the default Logview
workstep.setApplyMode(WorkstepApplyModeSaveAndDisplay);
// not display selection window for the workstep
workstep.setAWI4SelectionWindowDisplayed(false);

// Layout template stored at user level set as output Logview
template
if (LogviewTemplate::exists(StorageLevelUser, "Well9_short"))

workstep.setLogviewTemplate(LogviewTemplate::get(StorageLevelU
ser, "Well9_short"));

// ...

lock.release();
```

If you need to show additional parameters in a non-standard user interface, then you may design your own GUI and add this custom widget to the **`Workstep`** in a new tab using **`addCustomWidgetInTab`** by passing a pointer to the custom widget (**`QWidget*`**) and the name of the tab. A **`Workstep`** may have several custom widgets displayed as tabs to the right of the parameter tab.

See the "Save and restore an Ocean workstep" section on page 2-57 for more information on how to save and restore custom data in a workstep.

Workstep capabilities

You may enable or disable some AWI method capabilities of your **`Workstep`**.

```
class Workstep : public DomainObject
{
public:
  void refuseInterpolatedInputs();
  void acceptInterpolatedInputs();
  ...
}
```

An input argument added to a **`Workstep`** filters on the dataset that is dropped in the AWI workflow by family, variable name or alias name to show the end user possible input variables that may be used as inputs for the AWI method processing.

See "Workstep arguments" section on page 2-16 for more information on how to create workstep input arguments.

By default, the **`acceptInterpolatedInputs`** property is set to true and the input argument considers the parent well of the current dataset to show to the user variables from reference-compatible datasets with the same family, same name or under the same alias name. Those variables are interpolated to the current dataset. Disable this AWI capability after **`Workstep`** creation by calling the **`refuseInterpolatedInputs`** function.

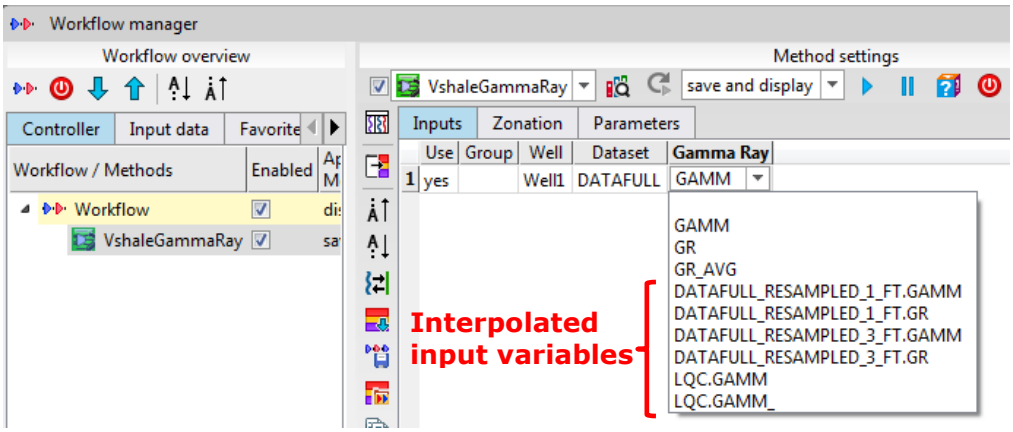


Figure 2-13 Interpolated input variables

```
class Workstep : public DomainObject
{
```

```
public:
  void enableMonteCarlo();
  void setMonteCarloActivated(bool activate);
  bool isMonteCarloActivated() const;
  ...
}
```

The Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results.

Use Monte Carlo to model probabilistic (or stochastic) systems and set up the odds for a variety of outcomes. The Monte Carlo simulation methods enable you to analyze systems with many coupled degrees of freedom: Fluids, Disordered materials, and Strongly coupled solids. The Monte Carlo methods also enable you to model phenomena with significant uncertainty in inputs.

You can run a Monte Carlo analysis to take into account the uncertainty on input measurements and the parameter values of your AWI method, by enabling the Monte Carlo option in the right pane of the AWI.

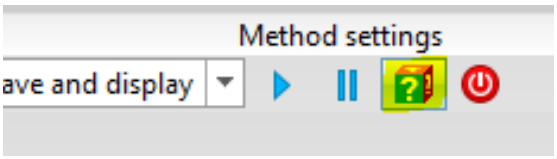


Figure 2-14 Monte-Carlo option

You may enable this option programmatically at the **`Workstep`** creation time and run a Monte Carlo statistical analysis during a plug-in **`Workstep`** (AWI method) computation.

`enableMonteCarlo` adds the Monte Carlo capability to the **`Workstep`**. By default, this capability is disabled and the Monte Carlo option button doesn't appear in the right pane of the AWI.

Set the **`isMonteCarloActivated`** property to true using the **`setMonteCarloActivated`** function to activate the Monte Carlo computation (toggle on the Monte Carlo option button).

Monte Carlo statistical analysis properties can't be changed programmatically with Ocean. Default Monte Carlo property values are applied to the **`Workstep`** as Monte Carlo variables that are created in the output of the Monte Carlo computation.

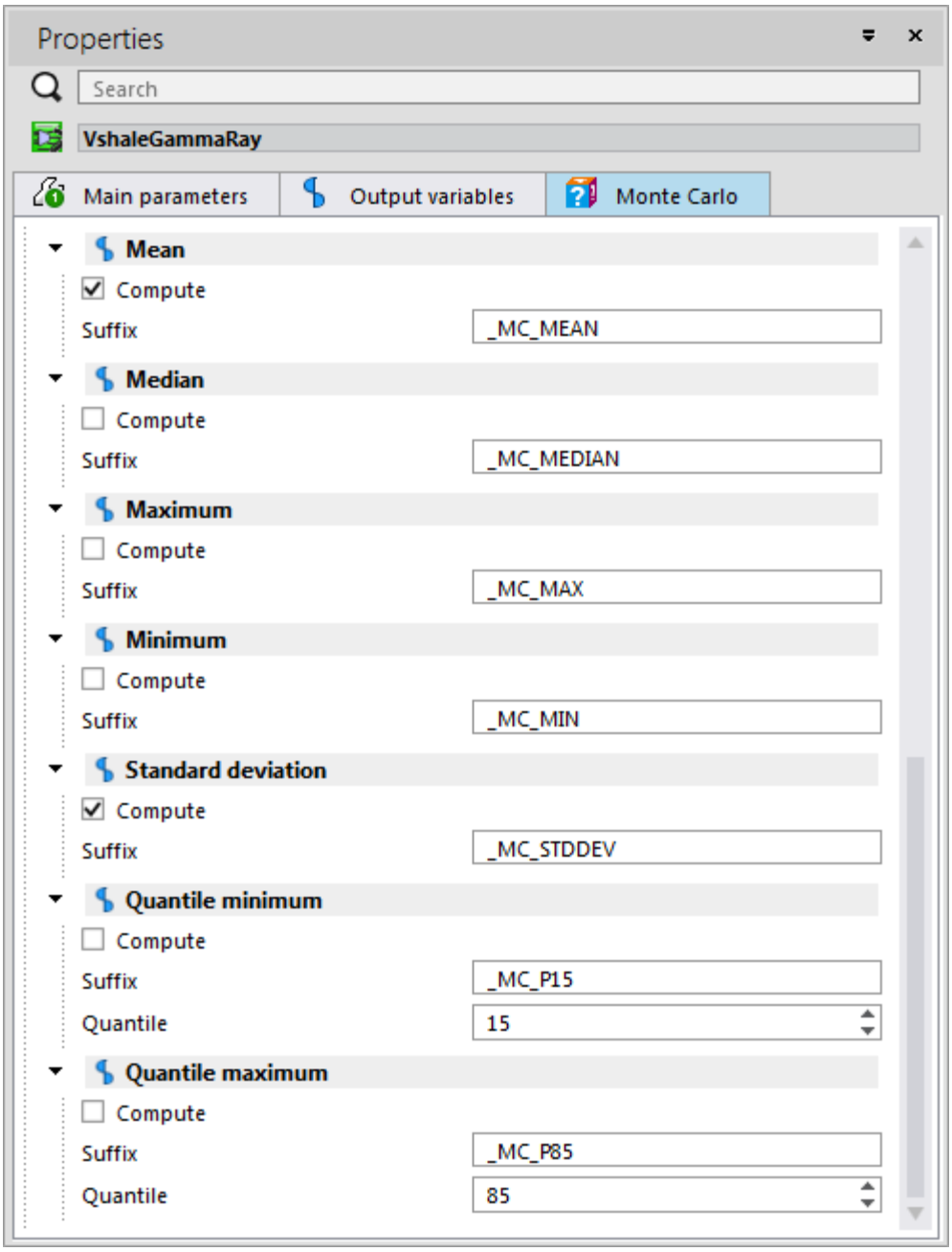


Figure 2-15 Monte-Carlo properties

Workstep arguments

Once a **`Workstep`** object is created, the next step in the run method of the activity is to add inputs, parameters and outputs which are known collectively as the workstep arguments.

Add input arguments to the **`Workstep`** using the static create methods of the **`InputWorkstepArgument`** domain object class.

```
class InputWorkstepArgument : public WorkstepArgument
{
public:
```

```
  static InputWorkstepArgument createInputVariableByFamilyName
    (const QString &argumentName, const QString &argumentUnit,
    const QString &familyName, Workstep workstep);
  static InputWorkstepArgument
    createInputVariableByVariableName(const QString
    &argumentName, const QString &argumentUnit, const QString
    &variableName, Workstep workstep);
  static InputWorkstepArgument createInputVariableByAlias(
    const QString &argumentName, const QString &argumentUnit,
    const QString &aliasName, Workstep workstep);
  static InputWorkstepArgument createInputVariableList(const
    QString &argumentName, const QStringList &variablesNames,
    Workstep workstep);
  Family familyName() const;
  void setFamilyName(const Family &familyName);

  QString  variableName() const;
  void setVariableName(const QString &variableName);

  QString aliasName() const;
  void setAliasName(const QString &aliasName);

  QList<QString> variableListNames() const;
  void setVariableListNames(const QStringList &varNameList);
  bool isModeList() const;
  ...
}
```

- **`createInputVariableByFamilyName`** – Creates an input variable argument that binds all the dataset variables with the family name automatically.
- **`createInputVariableByVariableName`** – Creates an input variable argument that binds all the dataset variables with the variable name automatically.
- **`createInputVariableByAlias`** – Creates an input variable argument that binds all the dataset variables with the alias name automatically. All the alias names are stored into the **AliasesTechlogList.xml** file at the root folder of your Techlog installation.
- **`createInputVariableList`** – Creates an input variable argument that binds all the dataset variables with variable names in **`QStringList`** automatically. The **`isModeList`** function returns true.

The binding types are not static; change them after creation using the **`setFamilyName`**, **`setVariableName`**, **`setAliasName`** and **`setVariableListNames`** functions.

Add additional equation parameters to the **`Workstep`** as float, double, string or Boolean using the create static methods of the **`ParameterWorkstepArgument`** domain object class.

```
class ParameterWorkstepArgument : public WorkstepArgument
{
public:
```

```
  static ParameterWorkstepArgument createBoolParameter(const
    QString &argumentName, bool defaultValue, const QString
    &tabName, Workstep workstep);
  static ParameterWorkstepArgument createFloatParameter(const
    QString &argumentName, const Unit &argumentUnit, const
    Measurement &measurement, float defaultValue, float
    minValue, float maxValue, const QString &tabName,
    Workstep workstep);
  static ParameterWorkstepArgument createStringParameter(const
    QString &argumentName, const QString &defaultValue, const
    QStringList &possibleValues, const QString &tabName,
    Workstep workstep);
  static ParameterWorkstepArgument createStringListParameter(
    const QString &argumentName, const QStringList
    &defaultValues, const QStringList &possibleValues, const
    QString &tabName, Workstep workstep);
  ...
}
```

A string parameter in the AWI is displayed as a drop-down list of possible values for the parameter by using the **`createStringParameter`** static method, but if the list of possible values is empty, then the parameter is displayed as an edit control where the end user may set any values for the parameter. The **`createStringListParameter`** allows you a multi selection of possible values (displayed as a drop-down list that contains checkable items) to create for a parameter.

Create a float parameter argument by passing to the static function:

- **`Unit`** used during the **`Workstep`** computation for this parameter.
- **`Measurement`** associated with the parameter that defines parameter's display unit for the current project unit system. The display unit is displayed in the parameter column name.

The computation unit is controlled by the plugin/workflow developers whereas the display unit is controlled by the users of the plugin/workflow. The user can then choose/edit the unit current system to choose which is its preferred display unit for this measurement.

See "Techlog measurements" section in *Ocean for Techlog Developer Guide - Basics* for more information on how units are managed in Techlog.

Note: The **`WorkstepArgument::getFloatValue`** function returns an **`Absent::MissingValue`** when it is called on a float **`ParameterWorktepArgument`** with a display unit (defined by the **`Measurement`** and **`Project`** unit system) that can't be converted to the computation unit (**`Unit argumentUnit`**).

Output arguments store the result of the workstep computation into the corresponding variables bound to the arguments. Add the outputs to the workstep using the **`createOutputVariable`** static public method of the **`OutputWorkstepArgument`** domain object class.

```
class OutputWorkstepArgument : public WorkstepArgument
{
public:
```

```
  static OutputWorkstepArgument createOutputVariable(const
    QString &argumentName, const QString &variableName,
    VariableDataFormat format, quint64 defaultColumnCount,
    const Unit &argumentUnit, const  QString
    &variableDescription, Workstep workstep);
  ...
}
```

The **`OutputWorkstepArgument`** static create function uses the same **`Unit`** (argumentUnit) used during the computation for this output.

The display unit of the **`Variable`** generated in output of the **`Workstep`** computation and bound to the **`OutpuWorkstepArgument`** is determined as follows:

- using the output variable name & computation unit, it finds the corresponding family
- gets the measurement from the family
- gets the display unit from the measurement (based on the current unit system) and uses that as output unit

If one of these steps fails, the computation unit passed to the function is used. The argument package needed for the Gamma Ray Average workstep is:

1) Input arguments:

   One input argument bound to a variable with gamma ray family.

2) Ouput arguments:

   One output argument containing the gamma ray average values per zone resulting of the workstep computation.

3) Parameter arguments:

   The workstep creates lithology zones following gamma ray average values per zone computed in output. The limit values that determine the lithology are stored in three parameter arguments named Sand, Shaly sand and Shale.

Workstep arguments are only defined at workstep creation time, like in the initialization of the activity. They cannot be added later, like in a slot method.

This example shows the Gamma Ray Average workstep arguments creation:

```
void WorkstepActivity::createWorkstepArguments(Workstep
workstep)
{
  // Input argument
  InputWorkstepArgument::createInputVariableByFamilyName(
  "Gamma Ray", "gAPI", "Gamma Ray", workstep);
  // Output argument
  OutputWorkstepArgument::createOutputVariable("Gamma Ray Avg",
  "GR_AVG", VariableDataFormatFloat, 1, "gAPI",
  "Gamma Ray Average", workstep);
  // create float parameters (upper limits of gamma ray lithology
  // values for sand, shaly sand and shale)
  ParameterWorkstepArgument::createFloatParameter("Sand",
  "gAPI", TechlogMeasurement::getGammaRay(), 20,
```

```
  Absent::MissingValue, Absent::MissingValue, "Parameters",
  workstep);
  ParameterWorkstepArgument::createFloatParameter("Shaly sand",
  "gAPI", TechlogMeasurement::getGammaRay(), 60,
  Absent::MissingValue, Absent::MissingValue, "Parameters",
  workstep);
  ParameterWorkstepArgument::createFloatParameter("Shale",
  "gAPI", TechlogMeasurement::getGammaRay(), 100,
  Absent::MissingValue, Absent::MissingValue, "Parameters",
  workstep);
}
```

Workstep signals

Once a workstep object is created, it listens to the workflow manager events by subscribing to the signals of the **`Workstep`** class.

```
class Workstep : public DomainObject
{
public:
  ...
  enum EventType
  {
    WorkingSetChanged,
    ParameterChanged,
    InputChanged,
    OutputChanged,
    CustomWorkstepPropertyChanged,
    Compute,
    ComputeDone,
    ComputeCancelled
  };
}
```

This is usually done in the **`run`** method of the plug-in activity. The signals are emitted whenever:

- **`WorkingSetChanged`** – a new dataset is added or removed from the workstep
- **`InputChanged`** – the variable bound to an input argument in the inputs tab of the workstep has changed.
- **`OutputChanged`** – properties like variable name, family, or unit of an output argument in the output tab of the workflow manager have changed.
- **`ParameterChanged`** – a parameter value in the parameters tab of the workstep has changed.
- **`CustomWorkstepPropertyChanged`** – the value of a custom workstep property has changed.

- **`Compute`** – the workstep computation has been run by the user clicking on the run computation button of the workflow manager.
- **`ComputeDone`** – the workstep computation is finished.
- **`ComputeCancelled`** – the workstep computation is cancelled programmatically or by the end-user.

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

```
#include "tsdkworkingsetchangedargs.h"
#include "tsdkinputchangedargs.h"
#include "tsdkoutputchangedargs.h"
#include "tsdkparameterchangedargs.h"
#include "tsdkcustomworksteppropertychangedargs.h"
#include "tsdkcomputeargs.h"
#include "tsdkcomputedoneargs.h"
#include "tsdkcomputecancelledargs.h"
```

```
private slots:
  void onWorkingSetChanged(
  const Slb::Ocean::Techlog::WorkingSetChangedArgs& args);

  void onInputChanged(
  const Slb::Ocean::Techlog::InputChangedArgs& args);

  void onOutputChanged(
  const Slb::Ocean::Techlog::OutputChangedArgs& args);

  void onParameterChanged(
  const Slb::Ocean::Techlog::ParameterChangedArgs& args);

  void onCustomWorkstepPropertyChanged(
  const Slb::Ocean::Techlog::CustomWorkstepPropertyChangedArgs&
args);

  void onCompute(const Slb::Ocean::Techlog::ComputeArgs & args);

  void onComputeDone(
  const Slb::Ocean::Techlog::ComputeDoneArgs& args);

  void onComputeCancelled(
  const Slb::Ocean::Techlog::ComputeCancelledArgs& args);
```

The Gamma Ray Average workstep subscribes to the **`Compute`**, **`ComputeDone`**, **`WorkingSetChanged`** and **`ParameterChanged`** signals. The **`Workstep`** class inherits from the **`DomainObject`** base class and the connection between signals and slots is done through the **`connect()`** pattern. Subscribe to the

DomainObjectErased signal to be notified when the workflow manager containing the custom workstep is closed to stop the plug-in activity.

```
void WorkstepActivity::connectWorkstep(Workstep workstep)
{
  // subscribe to workstep erased signal
  workstep.connect(Workstep::DomainObjectErased, this,
  SLOT(onWorkstepErased(const
  Slb::Ocean::Techlog::DomainObjectErasedArgs&)));
  // subscribe to run computation signal
  workstep.connect(Workstep::Compute, this,
  SLOT(onCompute(const Slb::Ocean::Techlog::ComputeArgs&)));
  // subscribe to computation done signal
  workstep.connect(Workstep::ComputeDone, this,
  SLOT(onComputeDone(const
  Slb::Ocean::Techlog::ComputeDoneArgs&)));
  // subscribe to workingSet changed signal
  workstep.connect(Workstep::WorkingSetChanged, this,
  SLOT(onWorkingSetChanged(const
  Slb::Ocean::Techlog::WorkingSetChangedArgs&)));
  // subscribe to parameter changed signal
  workstep.connect(Workstep::ParameterChanged, this,
  SLOT(onParameterChanged(const
  Slb::Ocean::Techlog::ParameterChangedArgs&)));
}
```

The corresponding slot methods implement the logic as follows:

- **onCompute** – for each dataset the gamma ray average value per zone is calculated and a new GR_AVG variable is created in the project browser.
- **onComputeDone** – the gamma ray average value calculated for each zone is used to identify the formation following the Sand, Shaly Sand and Shale limits values defined by the user in the parameters tab. A dataset "Lithology" with type "interval dataset" (**WellZonation** object) is created in the parent well.
- **onWorkingSetChanged** – warns the user if two datasets belong to the same well. In this case the gamma ray variable in the last dataset is used to compute the lithology.
- **onParameterChanged** – checks that there is no overlap between Sand, Shaly sand and Shale parameter values.

Note: The slot signature must be respected; pass the argument type corresponding to the signal that you want to connect to. Otherwise, the connection fails at run time. Every signal argument class inherits from the **SignalArgs** base class. In this class, the **sender** method returns the strongly typed sender of the signal. This is the **DomainObject** instance the slot was connected to.

WorkingSetChanged signal

The **`WorkingSetChanged`** signal has a **`WorkingSetChangedArgs`** argument. It provides the list of items (zone or dataset) in the workstep through the **`changedWorkingSetItems`** method and the list of additional items added to the workstep through the **`addWorkingSetItems`** method. For a workstep scoped by zone if the top and bottom of a zone is changed, the signal is also emitted and the zone that has changed is returned by the **`changedWorkingSet`** method.

```
class WorkingSetChangedArgs : public SignalArgsT<Workstep>
{
public:
  ...
  QList<WorkingSetItem> addWorkingSetItems(const
    WorkingSetItemFilter filter) const;
  QList<WorkingSetItem> changedWorkingSetItems(const
    WorkingSetItemFilter filter) const;
  WorkingSet changedWorkingSet() const;
};
```

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

```
void WorkstepActivity::onWorkingSetChanged(const
Slb::Ocean::Techlog::WorkingSetChangedArgs& args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  // retrieve the complete working set of the workstep from the
signal argument
  WorkingSet workingSet = args.changedWorkingSet();
  QList<QString> wellNames;
  // Search for dataset from the same well and warn the user
  foreach (const WorkingSetItem &item,
  workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange))
  {
    QString wellName = item.well().name();

    if (!wellNames.contains(wellName))
      wellNames.append(wellName);
    else
    {
      qWarning() << "A dataset which belongs to the well "
      << wellName << " already exists in the workstep";

      qWarning() << "The gamma ray variable selected in the last
dataset of well " << wellName << " will be used to compute its
lithology";
    }
  }
```

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

In this scenario, the four last zones of Well2.LQC dataset added to the workstep emit the **`WorkingSetChanged`** signal. Those zones are returned by the **`addedWorkingSetItems`** method.

`changedWorkingSet` returns the complete working set.

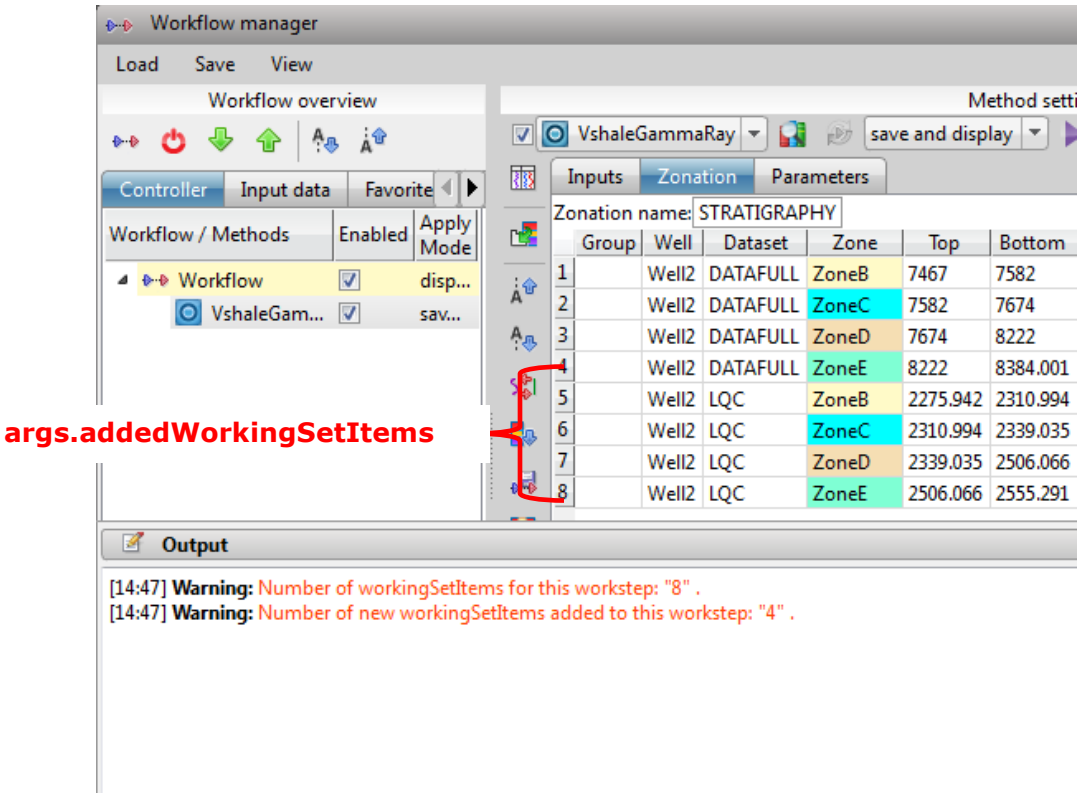


Figure 2-16 WorkingSetChanged signal arguments

If the end user changes the top or bottom values of a zone in the **Zonation** tab, then the **`Workingsetchanged`** signal is emitted and the zone that has been changed is returned by the **`changedWorkingSetItems`** method.

See the "Workstep data access" section on page 2-28 for more information on how access data in a workstep through workstep arguments (**`WorkstepArgument`**) and workstep items (**`WorkingSetItem`**).

InputChanged signal

The **`InputChanged`** signal uses the **`InputChangedArgs`** argument which gives the input argument that has been changed and for which items (zones of a dataset if workstep is scoped by zone).

```
class InputChangedArgs : public SignalArgsT<Workstep>
{
```

```
public:
  ...
  InputWorkstepArgument changedWorkstepArgument() const;
  QList<WorkingSetItem> changedWorkingSetItems(const
    WorkingSetItemFilter filter) const;
  WorkingSet changedWorkingSet() const;
};
```

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

```
void WorkstepActivity::onInputChanged(const
Slb::Ocean::Techlog::InputChangedArgs &args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  InputWorkstepArgument changedArgument =
  args.changedWorkstepArgument();

  WorkingSetItem changedItem = args.changedWorkingSetItems(
  WorkingSetItemFilterValidDataRange).first();

  Variable variable =
  changedArgument.boundVariable(changedItem);

  qWarning() << changedArgument.name()
    << " : input variable : " << variable.name();

  lock.release();
}
```

See the "Workstep data access" section on page 2-28 for more information on how access input variables in a workstep through workstep arguments (**`InputWorkstepArgument`**) and workstep items (**`WorkingSetItem`**).

OutputChanged signal

The **`OutputChanged`** signal uses the **`OutputChangedArgs`** argument that gives the output argument that has been changed.

Note: The argument returns the output argument where the user has changed a property as family, unit or variable name but does not return the property itself.

```
class OutputChangedArgs : public SignalArgsT<Workstep>
{
public:
  ...
  OutputWorkstepArgument changedWorkstepArgument() const;
  WorkingSet changedWorkingSet() const;
};
```

See the "Workstep data access" section on page 2-28 for more information on how to access output variables in a workstep through workstep arguments (**`OutputWorkstepArgument`**) and workstep items (**`WorkingSetItem`**).

ParameterChanged signal

The **`ParameterChanged`** signal uses the **`ParameterChangedArgs`** argument that gives the parameter argument that has been changed and for which items; depending of the workstep scope, they may be datasets or zones.

Note: Returning a list of items makes sense because a parameter value can be valid for several **`WorkingSetItem`**s in case of split zones (two zones with the same zone name and two differents top and bottom depths). The parameter value is shared by all zones with the same name at different depth intervals.

```
class ParameterChangedArgs : public SignalArgsT<Workstep>
{
public:
  ...
  ParameterWorkstepArgument workstepArgument() const;
  QList<WorkingSetItem> workingSetItems(const
    WorkingSetItemFilter filter) const;
  WorkingSet workingSet() const;
};
```

The slot handler accesses the needed informationfrom the signal argument.

```
void WorkstepActivity::parameterChanged(const
Slb::Ocean::Techlog::ParameterChangedArgs &args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  ParameterWorkstepArgument changedArgument =
  args.workstepArgument();

  // Iterate on all changed zones
  foreach (const WorkingSetItem &changedItem,
    args.workingSetItems(WorkingSetItemFilterValidDataRange))
  {
    Parameter parameter =
    changedArgument.boundParameter(changedItem);

    qWarning() << "parameter : " << parameter.name()
    << " at top index " << changedItem.first()
    << " and bottom index " << changedItem.last() ;
  }
  lock.release();
}
```

In this screenshot, ZoneB is split across two different depth intervals. The value of a parameter argument for this zone is changed and two items are returned by **`workingSetItems`** method of **`ParameterChangedArgs`** class.

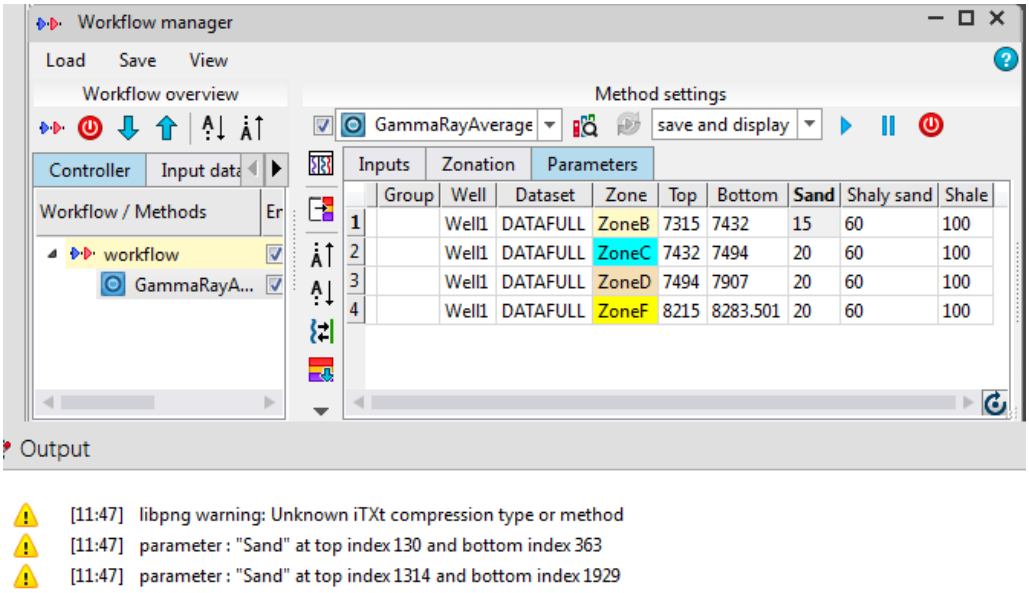


Figure 2-17 ParameterChanged signal arguments

See the "Workstep data access" section on page 2-28 for more information on how access parameter values in a workstep through workstep arguments (**`ParameterWorkstepArgument`**) and workstep items (**`WorkingSetItem`**).

CustomWorkstepPropertyChanged signal

The **`CustomWorkstepPropertyChanged`** signal uses a **`CustomWorkstepPropertyChangedArgs`** argument to give the custom workstep property that has been changed and its new value.

```
class CustomWorkstepPropertyChangedArgs :
      public SignalArgsT<Workstep>
{
public:
  ...
  QList<CustomWorkstepProperty>
    changedCustomWorkstepProperties() const;
};
```

See the "CustomWorkstepProperty class" section on page 2-51 for more information on how to add custom properties to a workstep.

Compute signal

The **`Compute`** signal has **`ComputeArgs`** argument that returns the working set processed during the computation.

```
class ComputeArgs : public SignalArgsT<Workstep>
{  ...
```

```
public:
  WorkingSet workingSet() const
};
```

ComputeDone signal

The **ComputeDone** signal has a **ComputeDoneArgs** that gives the **Workstep** instance the slot was connected to through the sender method inherited from the **SignalArgs** base class.

```
class ComputeDoneArgs : public SignalArgsT<Workstep>
{
};
```

Note: The **ComputeDone** signal isn't emitted if the computation is cancelled. See the "ComputeCancelled signal" section on page 2-28 for more information on how to handle end of the computation if it is cancelled.

ComputeCancelled signal

The **ComputeCancelled** signal has a **ComputeCancelledArgs** that gives the **Workstep** instance the slot was connected to through the sender method inherited from the **SignalArgs** base class.

```
class ComputeCancelledArgs : public SignalArgsT<Workstep>
{
};
```

Workstep data access

At computation time in the activity's **onCompute** slot receiver method, the values of the gamma ray variable bound to the input argument need to be read for each dataset that has been dropped in the workstep. This is done through two Ocean for Techlog classes which are **WorkingSetItem** and **WorkstepArgument**.

WorkingSet and WorkingSetItem classes

A **WorkingSet** belongs to a **Workstep** object and it is the data context in which a **Workstep** is processed when running the computation.

A **WorkingSet** relies on the workstep processing scope and may be composed of several **Dataset** or **Zone** instances (as list of **WorkingSetItems**) over which the **Workstep** runs.

Note: **WorkingSet** and **WorkingSetItem** are not domain objects and do not inherit from the **DomainObject** base class.

```
class WorkingSet : public IPrintable
{
public:
  Workstep workstep() const;
```

```
  QString zonationName() const;
  void setProgress(quint32 percentage);
  bool cancelled() const;
  const QList<WorkingSetItem> workingSetItems(const
    WorkingSetItemFilter filter) const;
  const QList<WorkingSetItem> usedWorkingSetItems(const
    WorkingSetItemFilter filter) const;
  const QList<WorkingSetItem> unusedWorkingSetItems(const
    WorkingSetItemFilter filter) const;
};
```

The **`WorkingSet`** class allows you to set the percentage of processing progress displayed in the Techlog progress bar (**`setProgress`** method) or get whether the computation of this working set was cancelled (**`isCancelled`** method).

The **`WorkingSet`** class returns the exhaustive list of **`WorkingSetItem`** objects through the **`workingSetItems`** method. **`usedWorkingSetItems`** and **`unusedWorkingSetItems`** functions respectively get the list of **`WorkingSetItems`** (datasets or zones) of the **`WorkingSet`** that are flagged as used (the default configuration after a dataset is dropped) and those that are flagged as not used (**Use** column of the **Inputs** tab set to **no** by the end user). The **`workingSetItems`** function returns all **`WorkingSetItem`**s, both used and not used.

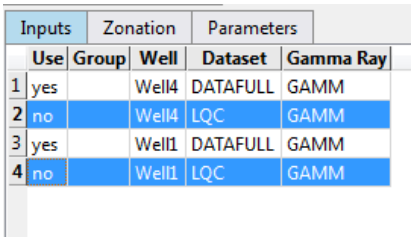


Figure 2-18 Used and not used workingSetItems

You can filter whether you want valid or unvalid zones in the list of returned **`WorkingSetItems`** by using the **`WorkingSetItemFilter`** enum argument:

- **`WorkingSetItemFilterInvalidDataRange`** – a zone is considered invalid when:
  - top and bottom depth values equal to 0
  - top depth value is greater than bottom depth value
  - the zone bottom depth value is higher or equal to dataset reference top depth value
  - the zone top depth value is lower or equal to dataset reference bottom depth value
- **`WorkingSetItemFilterValidDataRange`** – a zone is considered valid when it doesn't break rules listed above
- **`WorkingSetItemFilterNoFilter`** – returns valid and invalid zones.

A **`WorkingSetItem`** is an element of a **`WorkingSet`**. Depending on the **`WorkstepProcessingScope`** of the corresponding **`Workstep`**, a **`WorkingSetItem`** is of type **`Dataset`** or **`Zone`**.

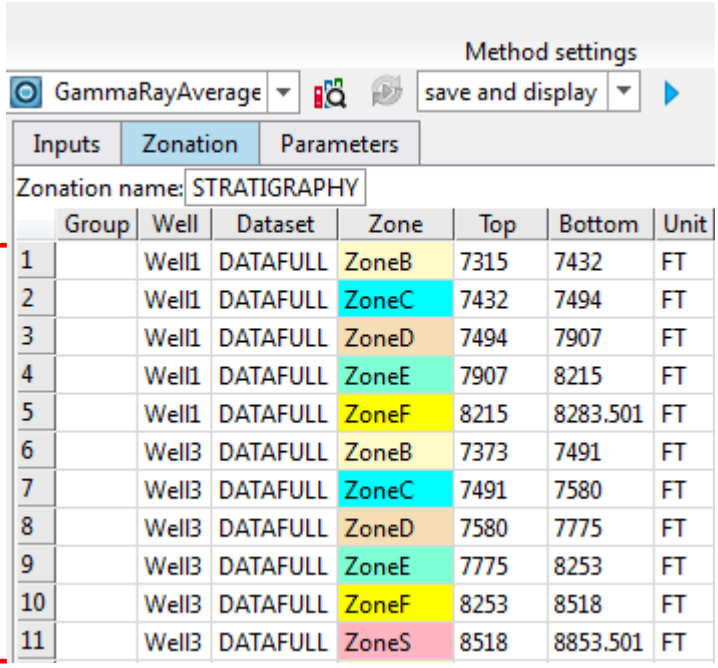
```
class WorkingSetItem : public IPrintable
{
public:
  bool isEmpty() const;
  WorkingSet workingSet() const;
  Well well() const;
  Dataset dataset() const;
  QString zoneName() const;
  bool isUsed() const;

  quint64 first() const;
  quint64 last() const;
  quint64 count() const;
  IndexIterator indexIterator() const;
};
```

If the **`WorkingSetItem`** is a **`Zone`** then you get the dataset and the well to which the zone belongs through the corresponding properties of the class. If the scope a dataset, then the **`Dataset`** property is the **`WorkingSetItem`** itself.

Note: Accessing the **`ZoneName`** property if the scope is dataset throws an exception.

These screenshots show this concept. If the **`Workstep`** is created with a dataset scope, you do not have access to the **Zonation** tab of the **Workflow manager** and you are only able to get the dataset and well from the **`WorkingSetItem`** object.

QList<WorkingSetItem>
workstep.workingSet().
workingSetItems()
=
List of zones



	Group	Well	Dataset	Zone	Top	Bottom	Unit
1		Well1	DATAFULL	ZoneB	7315	7432	FT
2		Well1	DATAFULL	ZoneC	7432	7494	FT
3		Well1	DATAFULL	ZoneD	7494	7907	FT
4		Well1	DATAFULL	ZoneE	7907	8215	FT
5		Well1	DATAFULL	ZoneF	8215	8283.501	FT
6		Well3	DATAFULL	ZoneB	7373	7491	FT
7		Well3	DATAFULL	ZoneC	7491	7580	FT
8		Well3	DATAFULL	ZoneD	7580	7775	FT
9		Well3	DATAFULL	ZoneE	7775	8253	FT
10		Well3	DATAFULL	ZoneF	8253	8518	FT
11		Well3	DATAFULL	ZoneS	8518	8853.501	FT

Figure 2-19 Workstep scoped by zone

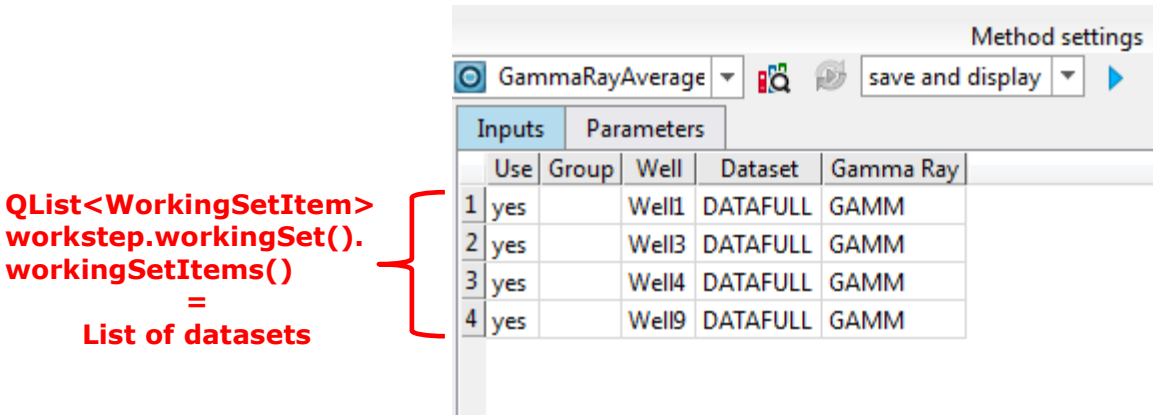


Figure 2-20 Workstep scoped by dataset

A **`WorkingSetItem`** object corresponds to an interval (**`Zone`** or **`Dataset`**) for which you have a top (**`first`**) and bottom (**`last`**) index and have access to the dataset indices for this interval through the **`indexIterator`** method.

This example accesses the indices.

```
void WorkstepActivity::iterateOnWorkingSetItems(WorkingSet
workingSet)
{
  foreach (const WorkingSetItem & item,
  workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange))
  {
    qWarning() << "Top index = " << item.first();
    qWarning() << "Bottom index = " << item.last();

    foreach(quint64 index, item.indexIterator())
    {
      qWarning() << "Index " << index << " of zone " <<
      item.zoneName();
      // …
    }
  }
}
```

This example is the slot receiver implementation of the **`WorkingSetChanged`** signal for the Gamma Ray Average workstep. The **`WorkingSet`** is retrieved from the signal argument **`WorkingSetChangedArgs`** and for each **`WorkingSetItem`** the method checks if the items (dataset) added to the **`Workstep`** do not belong to a well that already exists in the workstep. In the case of a duplicated well, the end user is warned in the Techlog output console that the gamma ray variable bound to the last dataset in the duplicated well will be used to compute the zonation **`Dataset`**.

```
void WorkstepActivity::onWorkingSetChanged(const
Slb::Ocean::Techlog::WorkingSetChangedArgs& args)
```

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

  // retrieve the complete working set of the workstep from the
signal argument
  WorkingSet workingSet = args.changedWorkingSet();
  QList<QString> wellNames;
  // Search for dataset from the same well and warn the user
  foreach (const WorkingSetItem &item,
  workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange))
  {
    QString wellName = item.well().name();

    if (!wellNames.contains(wellName))
      wellNames.append(wellName);
    else
    {
      qWarning() << "A dataset which belongs to the well "
      << wellName << " already exists in the workstep";

      qWarning() << "The gamma ray variable selected in the last
dataset of well " << wellName << " will be used to compute its
lithology";
    }
  }

  lock.release();
}
```

In this screenshot, when the LQC dataset is dropped in the **Workflow manager**, the end user is warned that there is already a dataset which belongs to Well1.

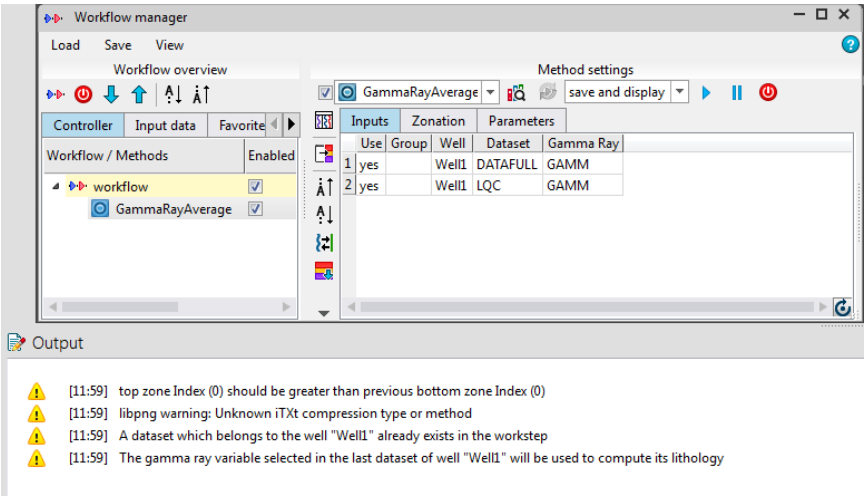


Figure 2-21 WorkingSetChanged signal emitted for GammaRayAverage workstep

WorkstepArgument domain object

The input, parameter and output arguments of a workstep are modeled using the **`InputWorkstepArgument`**, **`ParameterWorkstepArgument`** and **`OutputWorkstepArgument`** classes respectively, which all derive from the **`WorkstepArgument`** base class. It holds some fields common to each of those classes.

```
class WorkstepArgument : public DomainObject
{
public:
  ...
  void setDescription(const QString &description);

  WorkstepArgumentBoundObjectType boundObjectType(const
    WorkingSetItem& workingSetItem) const;
  Variable boundVariable(const WorkingSetItem& workingSetItem)
    const;
  bool isInterpolated(const WorkingSetItem &workingSetItem)
    const;
  Variable sourceVariable(const WorkingSetItem &workingSetItem)
    const;
};
```

The **`setDescription`** function allows to set a description to any derived objects as **`InputWorkstepArgument`**, **`ParameterWorkstepArgument`** and **`OutputWorkstepArgument`**. The description value passed to the **`setDescription`** function is showing up as a tooltip when the mouse hovers input or parameter column header.

You can resolve a workstep argument by its name or droid from the list of **`arguments`** which belong to a **`Workstep`** object using the **`get`** and **`find`** patterns.

Then **`WorkstepArgument`** methods are available to get or set the object bound to the argument for a **`WorkingSetItem`** object.

Input, parameter and output arguments can be bound to a **`Variable`** object for a given working set item using **`boundVariable`** accessor, while a parameter argument is bound to a **`Parameter`** object (constant value).

See the "ParameterWorkstepArgument " section on page 2-46 for more information on how to bind by **`Parameter`**.

Note: If an argument is unbound, then **`boundObjectType`** method returns **`WorkstepArgumentBoundObjectTypeUnbound`** enum value from **`WorkstepArgumentBoundObjectType`** enum class. Calling **`boundVariable`** on an unbound argument will throw an exception.

This example shows how to get each gamma ray variable bound to the input argument "Gamma Ray" in each dataset dropped in the workflow manager.

```
// Input
InputWorkstepArgument grArg = workstep.inputArguments()
.get("Gamma Ray");
// Output
OutputWorkstepArgument grAvgArg = workstep.outputArguments()
.get("Gamma Ray Avg");

// Iterate on all zones
foreach (const WorkingSetItem &workingSetitem,
  workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange))
{
  // check if the input argument is unbound
  if (grArg.boundObjectType(workingSetitem) ==
  WorkstepArgumentBoundObjectTypeUnbound)
                    continue;
  // get the variable bound to the input argument
  Variable gr = grArg.boundVariable(workingSetitem);
}
```

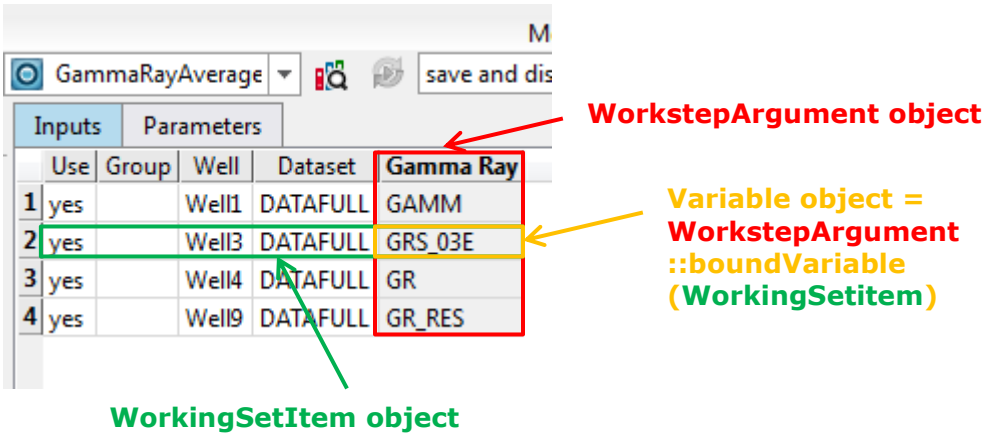


Figure 2-22 Variable bound to an input argument for a working set item

When you drop a dataset in the AWI, this dataset is the "working dataset": inputs are coming from this dataset and outputs are saved inside this dataset.

However, you can pick input variables from reference-compatible datasets. In this case the **`isInterpolated`** method returns true and the input workstep argument needs to be interpolated to fit the "working dataset".

A temporary **`Variable`** is created and is accessible through the **`Dataset::temporaryVariables()`** collection of the "working dataset". A call to **`boundVariable`** method returns this temporary variable while the **`sourceVariable`** method gets the selected **`Variable`** in the reference-compatible dataset.

See "Workstep capabilities" section on page 2-14 for more information on how to refuse or accept interpolated input variables in the workstep.

Read and write accessors are available for each data format. From the **`WorkstepArgument`** object you can get and set a value for a **`WorkingSetItem`** object (zone or dataset) at one row index and optionally one column index if the workstep agument is bound with an array variable. You can also get and set an array of values of the workstep argument, for a working set item, starting at a given index. In case of an array variable the vector of values can be one row of data, or several concatenated rows (in row-major).

By default, when you declare your input arguments, you specify units that you expect the variable values in. Ocean automatically does the conversion internally from the stored unit to the argument unit when you ask for data values through the read accessors (e.g. **`getFloatValue`**). This works well if the declared unit on the input argument and the bound variable unit are compatible but throws an exception if they are not.

There are many cases when you don't want to do any conversions because the data could be any variable (for example, cutoff variables). In that case, it is difficult to pick a unit for the input argument. You accept whatever user selects and do not need any conversion because you don't know what to convert to.

The conversion also fails for cases when you did not specify a unit and user selected a variable with defined unit. In this case there is no conversion factor from this variable unit to unitless. Use the **`getFloatValueUnconverted`** or **`getDoubleValueUnconverted`** to get the values in the same unit.

```
class WorkstepArgument : public DomainObject
{
public:
  float getFloatValue(const WorkingSetItem& workingSetItem,
    quint64 row, quint64 column) const;
  float getFloatValueUnconverted(const WorkingSetItem&
    workingSetItem, quint64 row, quint64 column) const;
  double getDoubleValue(const WorkingSetItem& workingSetItem,
    quint64 row, quint64 column) const;
  double getDoubleValueUnconverted(const WorkingSetItem&
    workingSetItem, quint64 row, quint64 column) const;
  QString getStringValue(const WorkingSetItem& workingSetItem,
    quint64 row, quint64 column) const;
```

```
  QVector<float> getVariableFloatValuesRange(const
    WorkingSetItem &workingSetItem, quint64 row, quint64 count)
    const;
  QVector<double> getVariableDoubleValuesRange(const
    WorkingSetItem &workingSetItem, quint64 row, quint64 count)
    const;
  void setFloatValue(float value, const WorkingSetItem&
    workingSetItem, quint64 row, quint64 column);
  void setDoubleValue(double value, const WorkingSetItem&
    workingSetItem, quint64 row, quint64 column);
  void setStringValue(const QString& value, const
    WorkingSetItem& workingSetItem, quint64 row, quint64 column);
  void setVariableFloatValues(const QVector<float>& values,
    const WorkingSetItem& workingSetItem, quint64 row);
  void setVariableDoubleValues(const QVector<double>& values,
    const WorkingSetItem& workingSetItem, quint64 row);
  void setVariableStringValues(const QVector<QString>& values,
    const WorkingSetItem& workingSetItem, quint64 row);
  ...
};
```

This example is the slot receiver implementation of the **`Compute`** signal for the Gamma Ray Average workstep; it retrieves the input and output arguments, computes the average gamma ray value for the current zone in the loop and all its row indexes and sets the result to the workstep output argument.

```
void WorkstepActivity::onCompute(const
Slb::Ocean::Techlog::ComputeArgs & args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  // get workingSet
  WorkingSet workingSet = args.workingSet();
  // Get the workstep
  Workstep workstep = workingSet.workstep();

  int zoneCount = workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange).count();
  int processedZones = 0;
  // Retrieve workstep arguments
  // Input
  InputWorkstepArgument grArg = workstep.inputArguments()
  .get("Gamma Ray");
  // Output
  OutputWorkstepArgument grAvgArg = workstep.outputArguments()
  .get("Gamma Ray Avg");

  // Iterate on all zones
  foreach (const WorkingSetItem &workingSetitem,
    workingSet.workingSetItems(
    WorkingSetItemFilterValidDataRange))
```

```
  {
    // check if the input argument is unbound
    if (grArg.boundObjectType(workingSetitem) ==
    WorkstepArgumentBoundObjectTypeUnbound)
                        continue;
    // get the variable bound to the input argument
    Variable gr = grArg.boundVariable(workingSetitem);
    // compute average Gamma Ray value in the current zone
    float avg = computeAvgValue(grArg, workingSetitem);
    // Read the variable values for current zone
    foreach(quint64 index, workingSetitem.indexIterator())
    {
      // Set the average Gamma ray value calculated for the zone
to workstep output argument
      grAvgArg.setFloatValue(avg, workingSetitem, index, 0);

      // compute and set the progress bar count value
      quint32 overallProgress = (quint32) (100 * (processedZones
+ (float)(index - workingSetitem.first()) /
workingSetitem.count()) / zoneCount);

      workingSet.setProgress(overallProgress);

      if (workingSet.isCancelled())
      {
        qDebug() << "GammaRayAverage computation cancelled";
        lock.release();
        return;
      }
    }
    processedZones++;
  }
  lock.release();
}

// This method is called to compute the average gamma ray value
for a zone
float WorkstepActivity::computeAvgValue(WorkstepArgument
workstepArg, WorkingSetItem item)
{
  float sumValues = Absent::MissingValue;
  int nbItems = 0;
  // Read the variable values for current zone
  foreach(quint64 index, item.indexIterator())
  {
    // Get the Gamma Ray input value at one row index of the zone
and add it to the sum of values
```

```
    float value = workstepArg.getFloatValue(item, index, 0);
    if (value != Absent::MissingValue)
    {
      if (sumValues == Absent::MissingValue)
        sumValues = value;
      else
        sumValues = sumValues + value;
      nbItems++;
    }
  }

  if (sumValues == Absent::MissingValue)
    return sumValues;

  return sumValues/nbItems;
}
```

This screenshot shows when each DATAFULL dataset added to the workstep, the GR_AVG output variable is created as the result of the workstep computation.

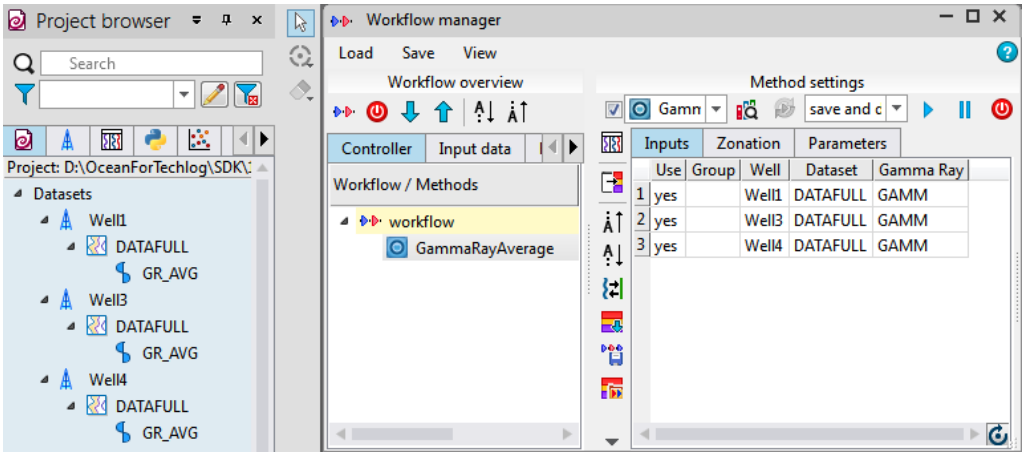


Figure 2-23 Compute signal emitted for GammaRayAverage workstep

The slot method of the **`ComputeDone`** signal:

- Uses the gamma ray average value per zone calculated by the workstep in the **`onCompute`** method and stored into the GR_AVG output variable to identify for each zone what is the formation regarding "Sand", "Shaly sand", and "Shale" parameters limits.
- creates a zonation **`Dataset`** object (interval dataset) with the formation name, top and bottom depth values of each zone. This is done in each parent well of the dataset containing the input gamma ray variable bound to the gamma ray input argument.

See the "Zonation creation" section on page 1-38 for more information on how to create a zonation dataset with Ocean for Techlog.

This example is the slot receiver implementation of the **`ComputeDone`** signal.

```
void WorkstepActivity::onComputeDone(const
Slb::Ocean::Techlog::ComputeDoneArgs& args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  Workstep workstep = args.sender();
  WorkingSet workingSet = workstep.workingSet();
  // …

  // get the gammma ray input argument
  InputWorkstepArgument grArg = workstep.inputArguments()
  .get("Gamma Ray");
  // get the parameter arguments
  ParameterWorkstepArgument sand =
  workstep.parameterArguments().get("Sand");
  ParameterWorkstepArgument shalySand =
  workstep.parameterArguments().get("Shaly sand");
  ParameterWorkstepArgument shale =
  workstep.parameterArguments().get("Shale");
  // get the gammma ray average output argument
  OutputWorkstepArgument grAvgArg = workstep.outputArguments()
  .get("Gamma Ray Avg");

  WorkingSetItem previousItem;
  QList<Zone> zones;
  bool firstZone = true;

  // delete sand, shaly sand and shale parameters if already are
in each zone of the trackItem displaying the gamma ray variable
  foreach (WorkingSetItem item,
  workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange))
  {

    // …

    // Create the "Lithology" WellZonation for each well in the
working set
    Variable ref = item.dataset().findReferenceVariable();
    float top = ref.getFloatValue(item.first());

    if (!firstZone)
      top = previousItem.dataset().findReferenceVariable().
      getFloatValue(previousItem.last());
    else
      previousItem = item;
```

```
    float bottom = ref.getFloatValue(item.last());

    float avg = grAvgArg.getFloatValue(item, 1, 0);

    QString itemWellDataset = item.well().name() + "." +
      item.dataset().name();
    QString previousItemWellDataset = previousItem.well().name()
+ "." + previousItem.dataset().name();

    if (previousItemWellDataset == itemWellDataset)
    {
      if (avg <= sand.getFloatValue(item, 0, 0)) // sand
      {
        zones.append(Zone("Sand", top, bottom));
      }
      else if (avg > sand.getFloatValue(item, 0, 0)
      && avg <  shalySand.getFloatValue(item, 0, 0)) // shaly sand
      {
        zones.append(Zone("Shaly sand", top,
        bottom));
      }
      else if (avg >= shalySand.getFloatValue(item, 0, 0)
      && avg < shale.getFloatValue(item, 0, 0)) // shale
      {
        zones.append(Zone("Shale", top, bottom));
      }

      previousItem = item;
      firstZone = false;
    }
    else
    {
      Well well = previousItem.well();
      ref = previousItem.dataset().findReferenceVariable();
      QString refFamily = "Measured Depth";
      QString refUnit = "ft";
      VariableDataFormat refFormat = VariableDataFormatFloat;
      if (!ref.isNull())
      {
         refFamily = ref.family();
         refUnit = ref.unit();
         refFormat = ref.format();
      }

      if (zones.count() != 0)
      {
```

```
        Dataset wellZonation =
        well.datasets().find("Lithology");

        if (wellZonation.isNull())
        // Create a lithology well zonation with same reference
family, unit and format than the variable reference of previous
workingsetItem dataset
        wellZonation = Dataset::create("Lithology",
        refFamily, refUnit, refFormat, well);

        wellZonation.setZones(zones,
        ZoneFamilyType::zoneFamilyName());

        zones.clear();
      }
    }
    firstZone = true;
  }

  Well well = workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange).last().well();

  Variable ref = workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange).last().dataset()
  .findReferenceVariable();

  QString refFamily = "Measured Depth";
  QString refUnit = "ft";
  VariableDataFormat refFormat = VariableDataFormatFloat;

  if (!ref.isNull())
  {
    refFamily = ref.family();
    refUnit = ref.unit();
    refFormat = ref.format();
  }

  if (zones.count() != 0)
  {
    Dataset wellZonation = well.datasets().find("Lithology");

    if (wellZonation.isNull())
    // Create a lithology well zonation with same reference family,
unit and format than the variable reference of previous
workingsetItem dataset
      wellZonation = Dataset::create("Lithology",
```

```
      refFamily, refUnit, refFormat, well);

    wellZonation.setZones(zones,
   ZoneFamilyType::zoneFamilyName());

   zones.clear();
  }

  // …

  lock.release();

  lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  Project project = Session::current().mainProject();
  // Set colors to GlobalZonation "Lithology" created by the
workstep computation
  ProjectZonationModel zonation = project.zonationModel();
  GlobalZonation globalZonation =
  zonation.findGlobalZonation("Lithology");

  if (!globalZonation.isEmpty())
  {
    foreach (GlobalZone zone, globalZonation.globalZones(
    ZoneFamilyType::zoneFamilyName()))
    {
      if (zone.name() == "Sand")
        zonation.setGlobalZoneColor("Lithology", "Sand",
        Qt::yellow);
      else if (zone.name() == "Shaly sand")
        zonation.setGlobalZoneColor("Lithology", "Sand",
        Qt::green);
      else if (zone.name() == "Shale")
        zonation.setGlobalZoneColor("Lithology", "Sand",
        Qt::yellow);
    }
  }
  lock.release();
}
```

This screenshot shows the result of the **`ComputeDone`** slot receiver. An interval dataset is created from the gamma ray average values computed previously for each zone of each dataset.

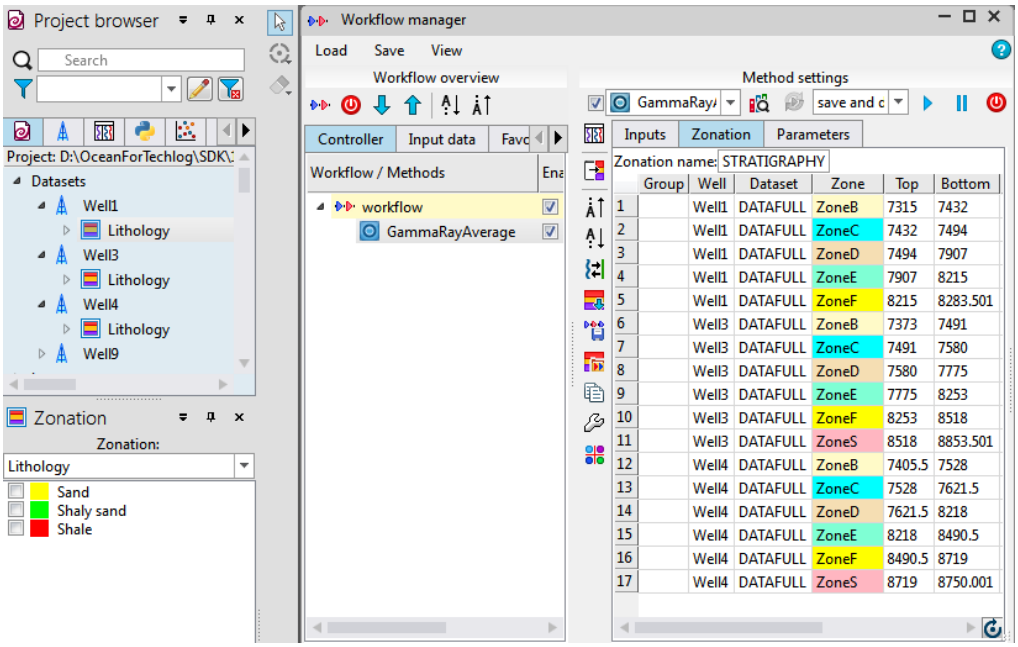


Figure 2-24 ComputeDone signal emitted for GammaRayAverage workstep

The slot method of the **`ParameterChanged`** signal is fired when a value for parameter (**`WorkstepArgument`**) and a zone (**`WorkingSetItem`**) is changed. The method checks that the new limit value set to the changed parameter for the changed item (zone) does not overlap another parameter limit value.

```
void WorkstepActivity::onParameterChanged(const
Slb::Ocean::Techlog::ParameterChangedArgs& args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  // retrieve the parent workstep from the workingSet
  Workstep workstep = args.workingSet().workstep();
  // get the item changed in the parameter tab of the AWI
  WorkingSetItem selectedItem = args.workingSetItems(
  WorkingSetItemFilterValidDataRange).first();

  // get the parameter argument changed
  ParameterWorkstepArgument changedArgument =
  args.workstepArgument();

  if (changedArgument.name() == "Sand")
  {
    ParameterWorkstepArgument paramShalySand =
    workstep.parameterArguments().get("Shaly sand");

    float shalySandValue =
    paramShalySand.getFloatValue(selectedItem, 0, 0);
```

```
    if (changedArgument.getFloatValue(selectedItem, 0, 0) >=
    shalySandValue)
                    {
      changedArgument.setFloatValue(shalySandValue - 1,
      selectedItem, 0, 0);
      qWarning() << "Sand limit value cannot be greater than Shaly
Sand.";
    }
  }
  else if (changedArgument.name() == "Shaly sand")
  {
    ParameterWorkstepArgument paramSand =
    workstep. parameterArguments().get("Sand");

    float sandValue = paramSand.getFloatValue(selectedItem, 0,
    0);

    if (changedArgument.getFloatValue(selectedItem, 0, 0) <=
    sandValue)
    {
      changedArgument.setFloatValue(sandValue + 1,
      selectedItem, 0, 0);
      qWarning() << "Shaly Sand limit value cannot be lower than
Sand.";
    }

    ParameterWorkstepArgument paramShale =
    workstep.parameterArguments().get("Shale");

    float shaleValue = paramShale.getFloatValue(selectedItem, 0,
0);

    if (changedArgument.getFloatValue(selectedItem, 0, 0) >=
    shaleValue)
    {
      changedArgument.setFloatValue(shaleValue - 1,
      selectedItem, 0, 0);
      qWarning() << "Shaly Sand limit value cannot be greater than
Shale.";
    }
  }
  else if (changedArgument.name() == "Shale")
  {
    ParameterWorkstepArgument paramShalySand =
    workstep.parameterArguments().get("Shaly sand");
```

```
    float shalySandValue =
    paramShalySand.getFloatValue(selectedItem, 0, 0);

    if (changedArgument.getFloatValue(selectedItem, 0, 0) <=
    shalySandValue)
    {
      changedArgument.setFloatValue(shalySandValue + 1,
      selectedItem, 0, 0);
      qWarning() << "Shale limit value cannot be lower than Shaly
Sand.";
    }
  }

  lock.release();
}
```

This screenshot shows that the user is warned if attempting to enter a Shaly sand limit for a zone lower than the Sand limit value.

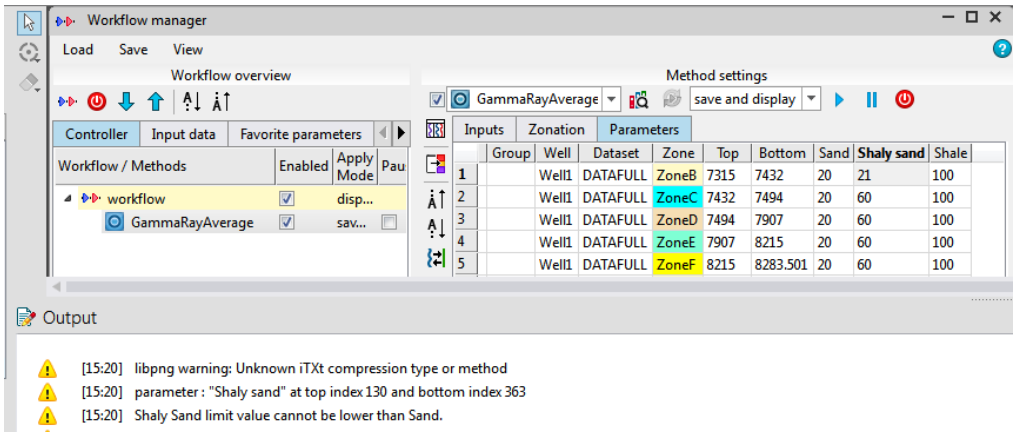


Figure 2-25 ParameterChanged signal emitted for GammaRayAverage workstep

InputWorkstepArgument domain object

The **`InputWorkstepArgument`** class represents the input argument of a workstep. This class inherits all public methods from the **`WorkstepArgument`** base class.

```
class InputWorkstepArgument : public WorkstepArgument
{
public:
  ...
  void setOptional( const bool isOptional );
  bool optional() const;

  void unbindVariable(const WorkingSetItem &workingSetItem);
  void setBoundVariable(const Variable& variable, const
       WorkingSetItem& workingSetItem);
```

```
  const QString familyName() const;
  const QString variableName() const;
  const QString aliasName() const;
  QList<QString> variableListNames() const;

  void setFamilyName(const Family &familyName);
  void setVariableName(const QString &variableName);
  void setAliasName(const QString &aliasName);
  void setVariableListNames(const QStringList
       &variableNameList);
};
```

Resolve a workstep input argument by its name or droid from workstep's list of **`inputArguments`** using the **`get`** and **`find`** patterns.

This example retrieves an input argument from the Gamma Ray Average workstep:

```
  InputWorkstepArgument grArg = workstep.inputArguments()
  .get("Gamma Ray");
```

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

- flag an input argument as **`optional`** (for example even if the input argument is unbound the workstep processing can be run because this input is optional)
- unbind the **`Variable`** bound to the input argument for a given **`WorkingSetItem`**
- change the **`Variable`** bound to the input argument for a given **`WorkingSetItem`**
- get the input argument **`familyName`**, **`variableName`**, **`aliasName`** or **`variableListNames`**, depending on how the argument is bound. This binding is defined at the input argument creation and may be changed after creation through the corresponding setters.

ParameterWorkstepArgument domain object

The **`ParameterWorkstepArgument`** class represents the workstep parameter argument. This class inherits all public methods from the **`WorkstepArgument`** base class.

```
class ParameterWorkstepArgument : public WorkstepArgument
{
public:
  ...
  Parameter boundParameter(const WorkingSetItem& workingSetItem)
    const;
  void setToConstant(const WorkingSetItem& workingSetItem);
  void setBoundVariable(const Variable& variable, const
    WorkingSetItem& workingSetItem);
```

```
  bool isParameterEnabled(const WorkingSetItem& workingSetItem)
    const;
  void setParameterEnabled(bool isEnabled, const
    WorkingSetItem& workingSetItem);
  QVariant defaultValue() const;
  QStringList possibleValues() const;
  void setPossibleValues(const QStringList& possibleValues)
    const;
  QStringList getStringListValue(const WorkingSetItem
    &workingSetItem) const;
  void setStringListValue(const QStringList &values, const
    WorkingSetItem &workingSetItem);
  Measurement measurement() const;
};
```

Resolve a workstep parameter argument can be resolved by its name or droid from the workstep's list of **`parameterArguments`** using the **`get`** and **`find`** patterns.

This example retrieves parameter argument from the Gamma Ray Average workstep:

```
  ParameterWorkstepArgument sand =
  workstep.parameterArguments().get("Sand");
```

Bind a parameter argument to **`a Parameter`** object (constant value) for a given **`WorkingSetItem`** using the **`setBoundParameter`** method.

See the "Parameter domain object" section on page 2-48 for more information on Parameter.

Note: Calling **`boundParameter`** on an unbound argument throws an exception.

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

- enable or disable the **`Parameter`** bound to the **`ParameterWorkstepArgument`** for a given **`WorkingSetItem`** (this throws an exception if the parameter is disabled and you try to a set a value through the **`Parameter`** object or bind the parameter argument to another **`Parameter`** for the given **`WorkingSetItem`**)
- get the **`defaultValue`** defined at the parameter argument creation (constant value).
- get and set the **`possibleValues`** of a string parameter argument.
- get and set list of string values for a multi-selection parameter.
- get the **`Measurement`** that defines the parameter's display unit.

A parameter argument can also be bound to a **`Variable`**. In the AWI the Techlog end user can switch a parameter from a constant to a variable using the corresponding menu item of the context menu:

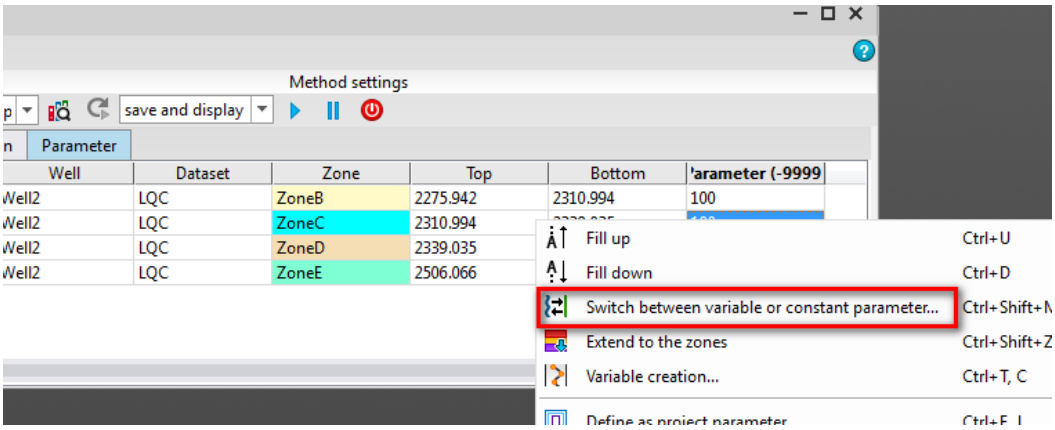


Figure 2-26 Switch between variable or constant parameter

This means that a **`ParameterWorkstepArgument`** can also be bound to a **`Variable`**.

- call the **`setBoundVariable`** method to switch the binding from a constant to a variable for a given **`WorkingSetItem`**
- call the **`setToConstant`** method to switch the binding from a variable to a constant for a given **`WorkingSetItem`** with **`defaultValue`** set at the **`ParameterWorkstepArgument`** creation

Note: Calling **`boundParameter`** on a **`ParameterWorkstepArgument`** bound to a **`Variable`** for a given **`WorkingSetItem`** throws an exception. Calling **`boundVariable`** on a **`ParameterWorkstepArgument`** bound to a **`Parameter`** (constant) for a given **`WorkingSetItem`** throws an exception.

Parameter domain object

You cannot create a **`Parameter`** object. The **`Parameter`** class exposes the constant value bound to a **`ParameterWorkstepArgument`** for a given **`WokingSetItem`** that can be displayed and used in some views.

```
class Parameter : public DomainObject
{
public:
  Well findWell() const;
  Dataset findDataset() const;
  GlobalZonation findGlobalZonation() const;
  GlobalZone findGlobalZone() const;
  void setGlobalZone( const GlobalZone& globalZone );

  QString group() const;
  void setGroup( const QString& group );
  QString family() const;
  void setFamily( const QString& family );
  QString unit() const;
  void setUnit(const QString & value);
```

```
  QString markerLabel() const;
  void setMarkerLabel(const QString & value);
  QString description() const;
  void setDescription(const QString & value);
  QVariant value() const;
  void setValue(const QVariant & value);
  float minValue() const;
  void setMinValue(const float & value);
  float maxValue() const;
  void setMaxValue(const float & value);
  QColor color() const;
  void setColor(const QColor & value);
};
```

Display a **`Parameter`** on the **`Logview`** as baselines (vertical line). In Techlog this functionality is available through the AWI toolbar **Display current layout parameters** button.

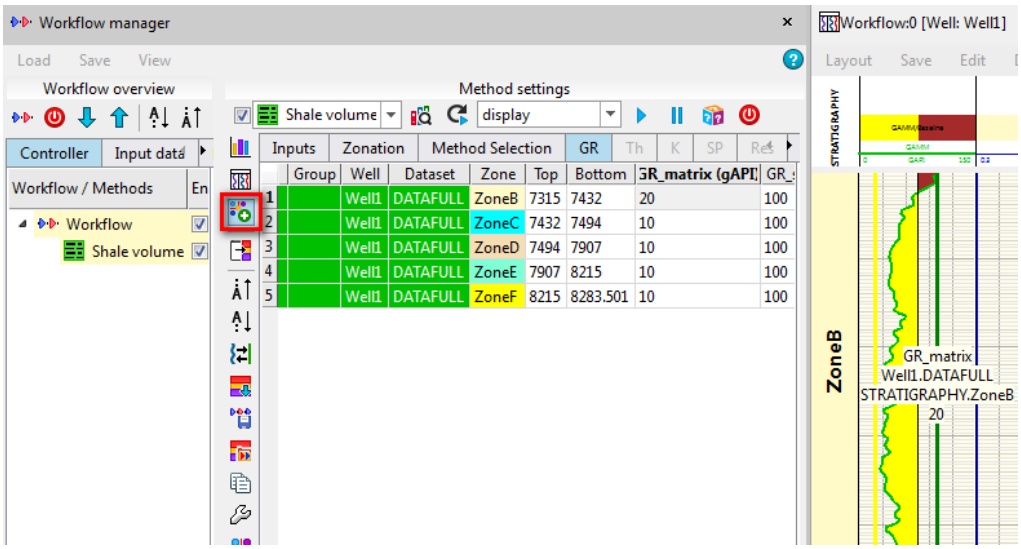


Figure 2-27 Display current layout parameters

See the "Workstep results display" section on page 2-54 for more information on how to display parameter bound to workstep argument for a zone in the AWI default logview.

If you hover the cursor over the baseline in the chart, the well, dataset, zonation and zone name associated with the **`Parameter`** appear. The corresponding APIs are available through the **`Parameter`** class. The value and global zone of the parameter can be set but the well, dataset and zonation properties are immutable.

See the "Logview" section in *Ocean for Techlog Developer Guide – Plots* for more information on how to visualize data into a **`Logview`**.

OutputWorkstepArgument domain object

The **`OutputWorkstepArgument`** class represents the workstep output argument. This class inherits all public methods from the **`WorkstepArgument`** base class.

```
class OutputWorkstepArgument : public WorkstepArgument
{
public:
  ...
  void setUsed( const bool isUsed );
  bool used() const;
  void setOutputVariableColumnCount(const quint64 columnCount,
    const WorkingSetItem& workingSetItem );
  void setOutputVariableColumnCountForScopeZone(const quint64
    columnCount, const Dataset& dataset );
  quint64 outputVariableColumnCount( const WorkingSetItem&
    workingSetItem ) const;
  const quint64 defaultColumnCount() const;
  const QString familyName() const;
  const QString variableName() const;
  Unit variableUnit() const;
  void setFamilyName(const Family &familyName);
  void setVariableUnit(const Unit &variableUnit);
};
```

Resolve a workstep output argument by its name or droid from the workstep's list of **`outputArguments`** using the **`get`** and **`find`** patterns.

This example retrieves an output argument from the Gamma Ray Average workstep:

```
  OutputWorkstepArgument grAvgArg = workstep.outputArguments()
  .get("Gamma Ray Avg");
```

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

- disable an output argument from the result computation by setting the **`used`** property to false. Workstep arguments are only defined at workstep creation. You can initialize an ouput argument turning off the **`used`** property and reactivate it later if it is required in output of the computation.
- set the number of columns of an output variable argument for a given **`WorkingSetItem`** using the **`outputVariableColumnCount`** property. The dimension of an output array variable is either set statically when the argument is declared, or set later if it depends on the data. If for instance the output variable dimension depends on some input variable, you call this method during the **`WorkingSetChanged`** event handler. Note that the dimension of an output may be different for the different datasets of the working set but must be the same for every zone of a given dataset (the function will throw an exception if you try to change it). For zone processing scope, use the **`setOutputVariableColumnCountForScopeZone`** function which prevents setting a different column count for a given dataset. It also throws an exception if used on an output variable of type double or string, since only float variables support arrays.
- get **`familyName`**, **`variableName`** and **`variableUnit`** (storage unit) of the output variable generated in output of the workstep computation and bound to the output argument. Those variable properties are defined at the output argument creation. Change the **`familyName`** and **`variableUnit`** after

creation through the corresponding setters, but the **`variableName`** is immutable.

CustomWorkstepProperty class

`CustomWorkstepProperty` objects give you the ability to extend the workstep properties. Custom workstep properties are not domain objects; they are containers that may nest recursively. They belong to a **`Workstep`**; retrieve them using the **`Workstep::customWorkstepProperties`** method.

Create a **`CustomWorkstepProperty`** using its constructor.

```
class CustomWorkstepProperty
{
public:
  ...
  CustomWorkstepProperty(const QString &name, const QString
    &tabName);
  void setDisplayName(const QString &value);
  const QString displayName() const;
  void setEditorType(const PropertyEditorType &value);
  const PropertyEditorType editorType() const;
  void setComboValueList(const QVariant &value);
  const QVariant comboValueList() const;
  void setValue(const QVariant &value);
  const QVariant value() const;
  void setParentPropertyName(const QString &value);
  const QString parentPropertyName() const;
  void setIcon(const QIcon &icon);
  void setTabName(const QString &value);
  const QString tabName() const;
  void setTabIcon(const QIcon &icon);
};
```

The enum **`PropertyEditorType`** determines the control type used to edit the custom workstep property. The possible values of a custom workstep property are presented to the end user through the control corresponding to the enum value, for example: ComboBox, ColorDialog, SpinBox, etc.

```
typedef enum PropertyEditorType
{
  PropertyEditorTypeGroup, // group node in the property tree
  PropertyEditorTypeSpinBox,
  PropertyEditorTypeDoubleSpinBox,
  PropertyEditorTypeBoolComboBox,
  PropertyEditorTypeDate,
  PropertyEditorTypeDateTime,
  PropertyEditorTypeTime,
  PropertyEditorTypeComboBox,
  PropertyEditorTypeLine,
```

```
  PropertyEditorTypeColorDialog,
  PropertyEditorTypeFontDialog,
  PropertyEditorTypePath,
  PropertyEditorTypePattern,
  PropertyEditorTypeLineWithCompletionList,
  PropertyEditorTypeLineForDouble,
  PropertyEditorTypeLink,
  PropertyEditorTypeComboCheckBox,
  PropertyEditorTypeTechplotDateTime,
  PropertyEditorTypeCount
};
```

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

- get and set the display name of the custom workstep property.
- get and set the editor type using a **`PropertyEditorType`** enumeration.
- when the **`editorType`** is a type of combo box, get and set the possible list of values for a custom workstep property through the **`comboValueList`** property. This method takes a list of **`QVariant`** values that are cast to the corresponding **`PropertyEditorType`** comboBox enum value format. **`PropertyEditorTypeBoolComboBox`** expects boolean values while **`PropertyEditorTypeComboBox`** expects string values.
- get and set the current value of the custom workstep property. This method takes a **`QVariant`** value that is cast to the corresponding **`PropertyEditorType`** enum value format which was set as **`editorType`**. **`PropertyEditorTypeSpinBox`** expects an integer value while **`PropertyEditorTypeColorDialog`** expects a color string value.
- get and set the parent property name to group the custom workstep property with another custom workstep property; the **`editorType`** must be set to **`PropertyEditorTypeGroup`**
- get and set the tab name where your custom workstep property is displayed in the Techlog **Properties** pane, when the workstep is selected in the **Workflow manager**
- set an icon for the tab and the custom workstep property. The icon must be in the plug-in folder.

Once the **`CustomWorkstepProperty`** object has its properties set, add it to the **`Workstep`** using the **`addCustomWorkstepProperty`** method.

Use public **`Workstep`** methods to find, update or remove a custom workstep property by its name.

```
class Workstep : public DomainObject
{
public:
  ...
  QList <CustomWorkstepProperty> customWorkstepProperties()
    const;
  CustomWorkstepProperty findCustomWorkstepProperty(const
    QString& propertyName) const;
```

```
  void addCustomWorkstepProperty(const CustomWorkstepProperty&
    property);
  void updateCustomWorkstepProperty(const
    CustomWorkstepProperty& property);
  void removeCustomWorkstepProperty(const
    CustomWorkstepProperty& property );
};
```

This example uses custom workstep properties.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();
// find the selected workflow from the current workspace activity
Workflow workflow = workspace.findSelectedWorkflow();
if (workflow.isNull()) workflow = Workflow::create("workflow",
workspace);
// Check for existence of the workstep name within the workflow
if (workflow.pluginWorksteps().contains("GammaRayAverage"))
{
  qWarning() << "The workflow already contains the gamma ray
average computation workstep";
  lock.release();
  return;
}

// create a workstep with a processing scope per zone
Workstep workstep = Workstep::create("GammaRayAverage",
WorkstepProcessingScopeZone, workflow);

// add custom properties
// Create a group property
CustomWorkstepProperty groupProperty =
CustomWorkstepProperty("MyGroup", "my custom tab");
groupProperty.setTabIcon(QIcon("ocean.png"));
groupProperty.setIcon(QIcon("organizeItems.png"));
groupProperty.setDisplayName("My Group");
groupProperty.setEditorType(PropertyEditorTypeGroup);
groupProperty.setExpanded(false);

// Create a custom property of type comboBox
CustomWorkstepProperty comboBoxProperty =
CustomWorkstepProperty("MyComboBox", "my custom tab");
comboBoxProperty.setParentPropertyName("MyGroup");
comboBoxProperty.setDisplayName("My ComboBox");
comboBoxProperty.setIcon(QIcon("comboFake.png"));
comboBoxProperty.setEditorType(PropertyEditorTypeComboBox);
comboBoxProperty.setComboValueList(QStringList() << "foo" <<
"bar");
```

```
comboBoxProperty.setValue("foo");

// Create a custom property of type color dialog
CustomWorkstepProperty colorProperty =
CustomWorkstepProperty("MyColor", "my custom tab");
colorProperty.setParentPropertyName("MyGroup");
colorProperty.setDisplayName("My Color");
colorProperty.setIcon(QIcon("colour.png"));
colorProperty.setEditorType(PropertyEditorTypeColorDialog);
colorProperty.setValue("Green");;

workstep.addCustomWorkstepProperty(groupProperty);
workstep.addCustomWorkstepProperty(comboBoxProperty);
workstep.addCustomWorkstepProperty(colorProperty);

lock.release();
```

This screenshot shows the result in the Techlog **Properties** when the workstep is selected in the **Workflow manager**:

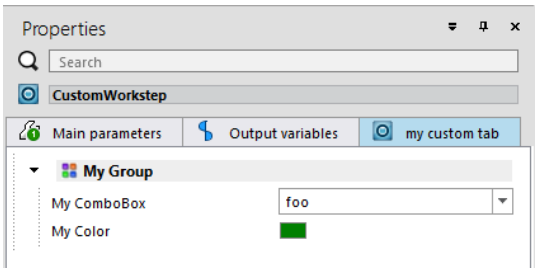


Figure 2-28 Custom workstep properties

Workstep results display

If the apply mode is set to "save and display" in the **Workflow manager** then the workstep input and output variables are displayed in the **Workflow manager**'s default logview after the workstep computation.

By default, when a **`Workstep`** object is created the method **`isApplyModeDisplay`** returns true if either the "save and display" or the "display" option is selected.

Retrieve the default logview instance using **`Workstep::findDefaultLogview`**.

```
class Workstep : public DomainObject
{
public:
  ...
  Logview findDefaultLogview() const;
  bool isApplyModeDisplay() const;
};
```

After the computation is completed (in the **`ComputeDone`** slot receiver), you add **`Parameter`** objects to the track containing the input variable by retrieving and using

the default logview instance. Those **`Parameter`** objects are the result of the workstep parameter arguments being bound to a value for each zone of the dataset.

This example adds add **`Parameter`** objects to the track.

```
void WorkstepActivity::computeDone(const
Slb::Ocean::Techlog::ComputeDoneArgs& args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  Workstep workstep = args.sender();
  WorkingSet workingSet = workstep.workingSet();

  // check if a default logview is displayed after workstep
computation
  if (!workstep.isApplyModeDisplay())
  {
    lock.release();
    return;
  }

  // get the default logview
  Logview logview = workstep.findDefaultLogview();
  // get the gammma ray input argument
  InputWorkstepArgument grArg = workstep.inputArguments()
  .get("Gamma Ray").cast<InputWorkstepArgument>();
  // get the parameter arguments
  ParameterWorkstepArgument sand = workstep.parameterArguments()
  .get("Sand").cast<ParameterWorkstepArgument>();
  ParameterWorkstepArgument shalySand =
  workstep.parameterArguments().get("Shaly sand")
  .cast<ParameterWorkstepArgument>();
  ParameterWorkstepArgument shale =
  workstep.parameterArguments().get("Shale")
  .cast<ParameterWorkstepArgument>();

  // ...

  // delete sand, shaly sand and shale parameters if already are
in each zone of the trackItem displaying the gamma ray variable
  foreach (WorkingSetItem item,
  workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange))
  {
    // Removing the parameters from the track containing the input
gamma ray variable
    foreach (NormalTrack track, logview.normalTracks())
    {
      foreach (TrackItem trackItem, track.trackItems())
```

```
      {
        Variable variable = trackItem.findVariable();

        if(!variable.isNull() && variable.name() ==
        grArg.boundVariable(item).name())
        {
          foreach(Parameter param, trackItem.parameters())
          {
            trackItem.removeParameter(param);
          }
        }
      }
    }
  }

  // ...

  // add parameters in each zone of the trackItem displaying the
gamma ray variable
  foreach (WorkingSetItem item,
  workingSet.workingSetItems(
  WorkingSetItemFilterValidDataRange))
  {
    foreach (NormalTrack track, logview.normalTracks())
    {
      if (track.findWell().name() == item.well().name())
      {
        foreach (TrackItem trackItem, track.trackItems())
        {
          Variable variable = trackItem.findVariable();

          if(!variable.isNull() && variable.name() ==
          grArg.boundVariable(item).name())
          {
            Parameter paramSand = sand.boundParameter(item);
            paramSand.setColor(Qt::yellow);
            trackItem.addParameter(paramSand);

            Parameter paramShalySand =
            shalySand.boundParameter(item);

            paramShalySand.setColor(Qt::green);

            trackItem.addParameter(paramShalySand);

            Parameter paramShale = shale.boundParameter(item);
```

```
            paramShale.setColor(Qt::red);
            trackItem.addParameter(paramShale);
          }
        }
      }
    }
  }

  lock.release();

  // ...

}
```

This screenshot shows the result. Changing a parameter value for a zone in the logview updates the corresponding parameter in the workstep **Parameters** tab.

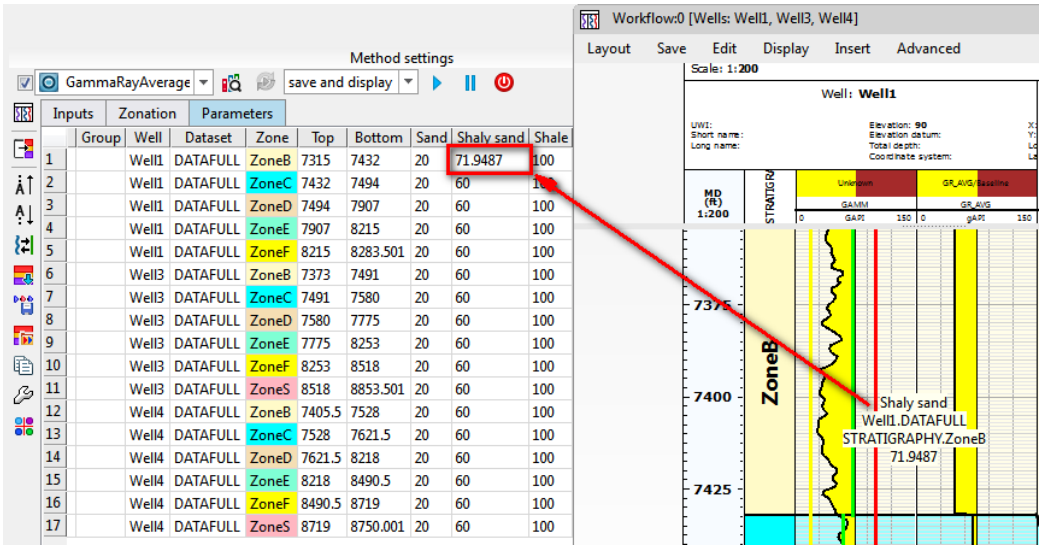


Figure 2-29 Default logview and workstep parameters

If the workstep's **`applyMode`** is set to **`WorkstepApplyModeDisplayOnly`**, the computed result is not saved in the project but it is shown in the AWI default logview using temporary variables. Retrieve those temporary variables from the input **`Dataset`** using the **`temporaryVariables`** methods. All the variables enumerated from this method have their **`isTemporary`** property set to true.

Save and restore an Ocean workstep

To save and restore an Ocean workstep in an AWI workflow, override the **`saveWorkstep`** and **`restoreWorkstep`** virtual methods in your plug-in activity derived from **`AbstractActivity`**.

```
class AbstractActivity : public QObject
{
public:
```

```
  virtual void saveWorkstep(const Workstep &workstep,
    QByteArray &data)
  virtual void restoreWorkstep(Workstep &workstep, const
    QByteArray &data)
  ...
};
```

When the end user clicks in the AWI **Save > Save workflow**, the **`saveWorkstep`** method is triggered.

If you do not have custom data (**`QWidget`** in a custom tab) for Techlog to save along with this **`Workstep`**, then overriding the **`saveWorkstep`** method without any specific implementation is enough to save the information in the native AWI tabs like inputs, parameters, zonation and outputs.

When you have custom data for Techlog to save along with the **`Workstep`** domain object, you must override the **`saveWorkstep`** virtual method and for each component of the GUI, get the value and add it to the **`QByteArray`** argument of the function.

This example activity creates a **`Workstep`** with a **`QWidget`** added to a custom tab. The **`QWidget`** contains two widgets: a line box and a combo box.

```
void WorkstepActivity::run()
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  Workspace workspace = Session::current().currentWorkspace();
  Workflow workflow = workspace.findSelectedWorkflow();
  if (workflow.isNull())
    workflow = Workflow::create("workflow", workspace);

  Workstep workstep = Workstep::create("GammaRayAverage",
  WorkstepProcessingScopeZone, workflow);

  createWorkstepArguments(workstep);
  connectWorkstep(workstep);
  createCustomWidget(workstep);

  lock.release();
}
```

```
void WorkstepActivity::createCustomWidget(Workstep workstep,
QByteArray data)
{
  // Create a custom UI and add it to a custom tab
  QLabel *label1 = new QLabel();
  label1->setText("Custom data 1");

  _comboBox = new QComboBox();
  _comboBox->addItem("value1");
  _comboBox->addItem("value2");
  _comboBox->addItem("value3");
  _comboBox->addItem("value4");
  _comboBox->addItem("value5");
```

```
  QHBoxLayout *hLayout1 = new QHBoxLayout();
  hLayout1->addWidget(label1);
  hLayout1->addWidget(_comboBox, 1);

  QLabel *label2 = new QLabel();
  label2->setText("Custom data 2");
  _lineEdit = new QLineEdit();

  QHBoxLayout *hLayout2 = new QHBoxLayout();
  hLayout2->addWidget(label2);
  hLayout2->addWidget(_lineEdit, 1);

  QVBoxLayout *vLayout = new QVBoxLayout();
  vLayout->addLayout(hLayout1);
  vLayout->addLayout(hLayout2);

  QWidget* widget = new QWidget();
  widget->setLayout(vLayout);

  // restore widget states if saved workflow
  if (!data.isEmpty())
  {
    QString savedString = data;
    QStringList states = savedString.split("|");
    if (!states.at(0).isEmpty())
    {
      int index = _comboBox->findText(states.at(0));
      _comboBox->setCurrentIndex(index);
    }
    if (!states.at(1).isEmpty())
    {
      _lineEdit->setText(states.at(1));
    }
  }

  workstep.addCustomWidgetInTab(widget, "My custom data tab");
}
```

When the end user saves the workflow that contains the Ocean **`Workstep`**, then the overridden **`saveWorkstep`** method is called and the widget states in the custom tab are saved into the **`QByteArray`**.

```
void WorkstepActivity::saveWorkstep(const Workstep &workstep,
QByteArray &data)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL( lock );
  qWarning() << "saving widget state into workstep = " <<
workstep.name();

  QString savedString;
  if (!_comboBox.isNull())
    savedString += _comboBox->currentText();
  savedString += "|";
```

```
  if (!_lineEdit->text().isEmpty())
    savedString += _lineEdit->text();
  data = savedString.toUtf8();

  lock.release();
}
```

Then the user restores the saved workflow from the Techlog project browser, then the overridden **`restoreWorkstep`** method is called. The state of the workstep is restored except that you still need to reconnect the signals to the **`Workstep`**. The **`QByteArray`** argument containing the custom widget's states is passed to the **`createCustomWidget`** method.

```
void WorkstepActivity::restoreWorkstep(Workstep &workstep, const
QByteArray &data)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL( lock );

  qWarning() << "restoring object workflow =" << workstep.name();

  connectWorkstep(workstep);
  createCustomWidget(workstep, data);

  lock.release();
}
```

This screenshot shows widgets with their custom data restored to the **`Workstep`**.

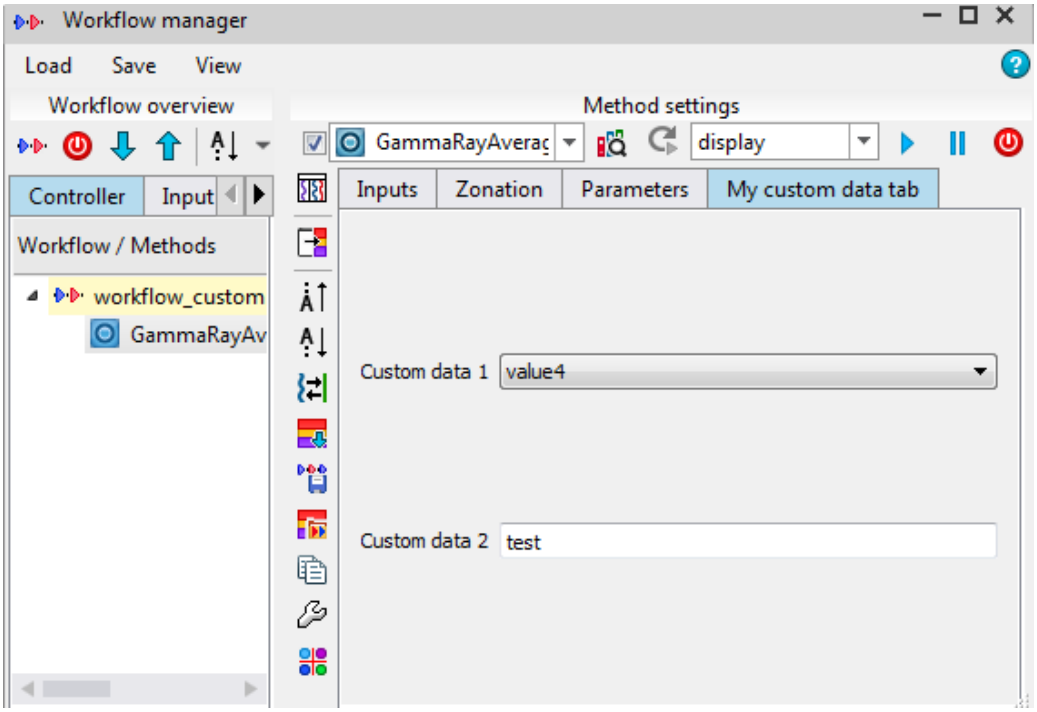


Figure 2-30 Custom data in a workstep

Extend workflow manager with custom actions

Ocean API enables you to extend the workflow manager with custom button actions.

You can create either push buttons or toggle buttons that can be located in the workflow manager mousebar or in the right mouse click context menu for a given parameter of an Ocean workstep.

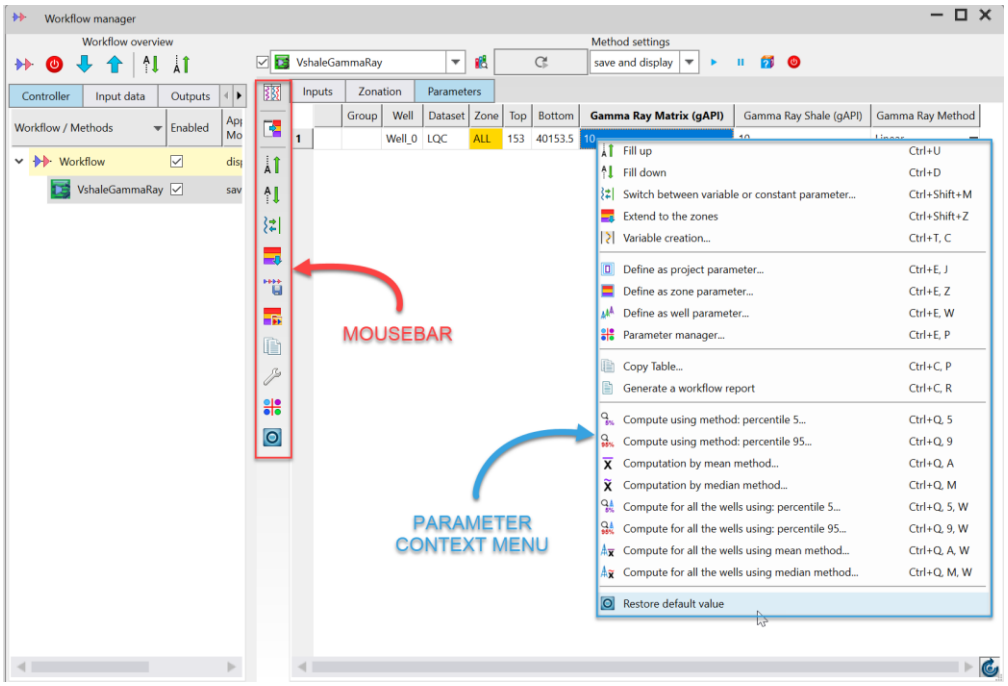


Figure 2-31 Actions in the workflow manager

WorkflowManagerAction object

Ocean exposes the ability to add different types of action items to the Techlog workflow manager.

1. Append the workflow manager toolbar with a new action item, defining its type (simple click or toggle button), state (checked or unchecked for a toggle button), tooltip and icon.
2. Create an action item to the right mouse click context menu of a workstep parameter.

These action items are handled by two different Ocean classes grouped under the **`WorkflowManagerAction`** base class. The class diagram shows these different classes:

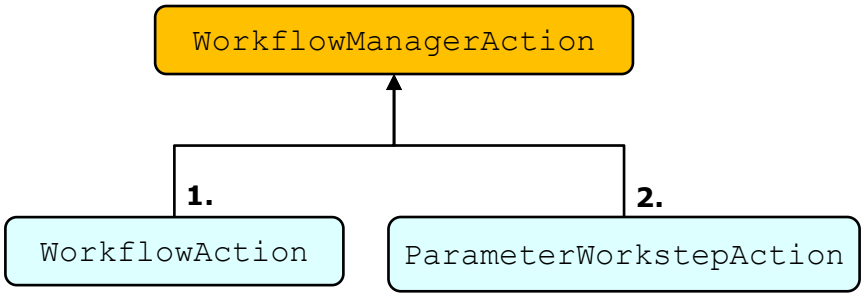


Figure 2-32 WorkflowManagerAction class diagram

The **WorkflowManagerAction** class holds properties common to all action items.

```
class WorkflowManagerAction: public DomainObject


{
public:
  bool isCheckable() const;
  bool isChecked() const;
  bool isEnabled() const;
  bool isVisible() const;
  QString text() const;
  QString toolTip() const;
  QIcon icon() const;
};
```

The **WorkflowManagerAction** class allows you to:

- Check if it is a simple click or toggle button.
- For toggle button check if it is toggle on or off.
- Check if the button is enabled or disabled.
- Gets text, tooltip and icon sets to the button.

The **WorkflowManagerAction** class holds common signals to all derived objects as **WorkflowAction** and **ParameterWorkstepAction**. Those objects can subscribe on these events.

```
class WorkflowManagerAction: public DomainObject
{
public:
 …
 enum EventType
 {
   WorkflowManagerActionHovered,
   WorkflowManagerActionToggled,
   WorkflowManagerActionTriggered
 };
}
```

The signals are emitted whenever:

- **WorkflowManagerActionHovered** – mouse hovers over the **WorkflowManagerAction** button (mouse over event).
- **WorkflowManagerActionToggled** – **isChecked** property value of a checkable **WorkflowManagerAction** has changed.
- **WorkflowManagerActionTriggered** – **WorkflowManagerAction** 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 "tsdkworkflowmanageractiontriggeredargs.h"
#include "tsdkworkflowmanageractionhoveredargs.h"
#include "tsdkworkflowmanageractiontoggledargs.h"
```

```
private slots:
  void onWorkflowManagerActionTriggered(const
Slb::Ocean::Techlog::WorkflowManagerActionTriggeredArgs& args);
  void onWorkflowManagerActionToggled(const
Slb::Ocean::Techlog::WorkflowManagerActionToggledArgs& args);
  void onWorkflowManagerActionHovered(const
Slb::Ocean::Techlog::WorkflowManagerActionHoveredArgs& args);
```

Any object derived from the **WorkflowManagerAction** base class can connect to all of these signals. But it makes sense to connect an action to the **WorkflowManagerActionToggled** signal with **isCheckable** sets to **true**.

Note: **WorkflowManagerActionToggled** 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 **WorkflowManagerActionHovered** signal. Simple click action connects to **WorkflowManagerActionTriggered** signal. Toggle action connects to **WorkflowManagerActionToggled** signal.

```
void WorkstepActivity::connectWorkflowManagerActions(Workstep
workstep)
{
  ParameterWorkstepArgument paramGRMatrixArg =
  workstep.getParameterWorkstepArgument("Gamma Ray Matrix");

  ParameterWorkstepAction paramActionGRMatrix =
      ParameterWorkstepAction::create("Action on GR Matrix",
      QIcon("ocean.png"), paramGRMatrixArg);

  CONNECT(paramActionGRMatrix, WorkflowManagerAction,
    WorkflowManagerActionTriggered, this, WorkstepActivity,
    onWorkflowManagerActionTriggered);
```

```
  WorkflowAction workflowAction =
   WorkflowAction::create("Workflow Toggle button",
   QIcon("ocean.png"), workstep.workflow());

  workflowAction.setCheckable(true);

  CONNECT(workflowAction, WorkflowManagerAction,
  WorkflowManagerActionTriggered, this, WorkstepActivity,
  onWorkflowManagerActionTriggered);

  CONNECT(workflowAction, WorkflowManagerAction,
  WorkflowManagerActionHovered, this, WorkstepActivity,
  onWorkflowManagerActionHovered);
}
```

WorkflowManagerActionHovered and **WorkflowManagerActionTriggered** signals include arguments that give the **WorkflowManagerAction** instance the slots were connected to through the **sender** method inherited from the **SignalArgs** base class. The arguments are modeled respectively in Ocean through **WorkflowManagerActionHoveredArgs** and **WorkflowManagerActionTriggeredArgs** classes.

```
class WorkflowManagerActionHoveredArgs : public
SignalArgsT<WorkflowManagerAction>

{
};
```

```
class WorkflowManagerActionTriggeredArgs : public
SignalArgsT<WorkflowManagerAction>

{
};
```

WorkflowManagerActionToggled 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 if its state is checked or unchecked.

This argument is modeled in Ocean through **ActionToggledArgs** class.

```
class WorkflowManagerActionToggledArgs : public SignalArgsT<
WorkflowManagerAction>

{

public:

  bool isChecked() const;
};
```

WorkflowAction object

A **WorkflowAction** object is instancied through **create** static method of the class passing as arguments the tooltip of the action, the icon that will be displayed for the button in the workflow manager mousebar and the parent **Workflow** instance.

```
class WorklowAction : public WorkflowManagerAction
{
public:
  static WorkflowAction create(const QString &tooltip,
  const QIcon &icon, Workflow workflow);

  void setCheckable(bool checkable);
  void setChecked(bool checked);
  void setEnabled(bool enabled);
  void setVisible(bool visible);
  void setToolTip(const QString &tooltip);
  void setIcon(const QIcon &icon);

};
```

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

- Turn the action item to a toggle button passing a boolean true value to the **setCheckable** function. Default value is false meaning that the action item is created in the workflow manager mousebar as a simple click button.
- Check or uncheck a toggle button through the **setChecked** function.
- Enable or disable the **WorkflowAction** (button is greyed out) through the **setEnabled** function.
- Hide or show the **WorkflowAction** through the **setVisible** function.
- Change after the **WorkflowAction** creation the icon and tooltip.

ParameterWorkstepAction object

A **ParameterWorkstepAction** object is instancied through **create** static method of the class passing as arguments the text of the action, the icon that will be displayed for the button in RMC context menu of the parameter and the parent **ParameterWorkstepArgument** instance.

```
class ParameterWorkstepAction : public WorkflowManagerAction
{
public:
  static ParameterWorkstepAction create(const QString &text,
  const QIcon &icon, ParameterWorkstepArgument
  parameterWorkstepArgument);

  void setEnabled(bool enabled);
  void setVisible(bool visible);
  void setText(const QString &text);
  void setToolTip(const QString &tooltip);
  void setIcon(const QIcon &icon);

};
```

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

- Enable or disable the **ParameterWorkstepAction** (button is greyed out) through the **setEnabled** function.
- Hide or show the **ParameterWorkstepAction** through the **setVisible** function.
- Sets a tooltip to the **ParameterWorkstepAction.**
- Change after the **ParameterWorkstepAction** creation the icon and text.

Note: Unlike **WorkflowAction**, a **ParameterWorkstepAction** can only be displayed as a simple click button.

Workflow manager action use case

The function below is called after a plug-in workstep and its arguments (inputs, parameters and outputs) have been created in an Ocean activity. See "Workstep creation" on page 2-9 and "Workstep arguments" on page 2-16 for more information on how to create a workstep and its arguments.

A **ParameterWorkstepAction** is added to each **ParameterWorkstepArgument** of the **Workstep**. A **WorkflowAction** is added to the **Workflow** mousebar.

Each of those actions connect to the **WorkflowManagerActionTriggered** signal.

```
void WorkstepActivity::createWorkflowManagerActions(Workstep
workstep)
{
  foreach(ParameterWorkstepArgument paramArg,
          workstep.parameterArguments())
  {
    ParameterWorkstepAction paramAction =
      ParameterWorkstepAction::create("Restore default value",
      QIcon("ocean.png"), paramArg);

    paramAction.setToolTip(QString("Restore default value for
%1").arg(paramArg.name()));

    CONNECT(paramAction, WorkflowManagerAction,
    WorkflowManagerActionTriggered, this, WorkstepActivity,
    onWorkflowManagerActionTriggered);
  }

  WorkflowAction workflowAction =
   WorkflowAction::create("Restore parameters default values",
   QIcon("ocean.png"), workstep.workflow());

  CONNECT(workflowAction, WorkflowManagerAction,
  WorkflowManagerActionTriggered, this, WorkstepActivity,
  onWorkflowManagerActionTriggered);
}
```

The slot receiver of the **`WorkflowManagerActionTriggered`** signal is responsible to restore:

- The default parameter value of the **`ParameterWorkstepArgument`** linked to a **`ParameterWorkstepAction`** sender.
- All default parameters values in all worksteps of the **`Workflow`** if the sender is a **`WorkflowAction`**.

```
void WorkstepActivity::onWorkflowManagerActionTriggered(const
Slb::Ocean::Techlog::WorkflowManagerActionTriggeredArgs& args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  WorkflowManagerAction workflowManagerAction = args.sender();

  if (workflowManagerAction.isA<ParameterWorkstepAction>())
  {
    ParameterWorkstepAction paramAction =
      workflowManagerAction.cast<ParameterWorkstepAction>();

    ParameterWorkstepArgument paramArg =
      paramAction.parameterWorkstepArgument();

    WorkingSet workingSet = paramArg.workstep().workingSet();
    foreach(const WorkingSetItem & item,
     workingSet.workingSetItems(WorkingSetItemFilterNoFilter))
    {
      if (paramArg.boundObjectType(item) ==
      WorkstepArgumentBoundObjectTypeVariable)
        paramArg.setToConstant(item);

      Parameter parameter = paramArg.boundParameter(item);
      parameter.setValue(paramArg.defaultValue());
    }
  }

  if (workflowManagerAction.isA<WorkflowAction>())
  {
    WorkflowAction workflowAction =
    workflowManagerAction.cast<WorkflowAction>();

    Workflow workflow = workflowAction.workflow();
    foreach(Workstep workstep, workflow.pluginWorksteps())
    {
      foreach(ParameterWorkstepArgument paramArg,
      workstep.parameterArguments())
```

```
      {
        WorkingSet workingSet = workstep.workingSet();
        foreach(const WorkingSetItem & item,
    workingSet.workingSetItems(WorkingSetItemFilterNoFilter))
        {
          if (paramArg.boundObjectType(item) ==
          WorkstepArgumentBoundObjectTypeVariable)
            paramArg.setToConstant(item);

          Parameter parameter = paramArg.boundParameter(item);
          parameter.setValue(paramArg.defaultValue());
        }
      }
    }
  }
  lock.release();
}
```