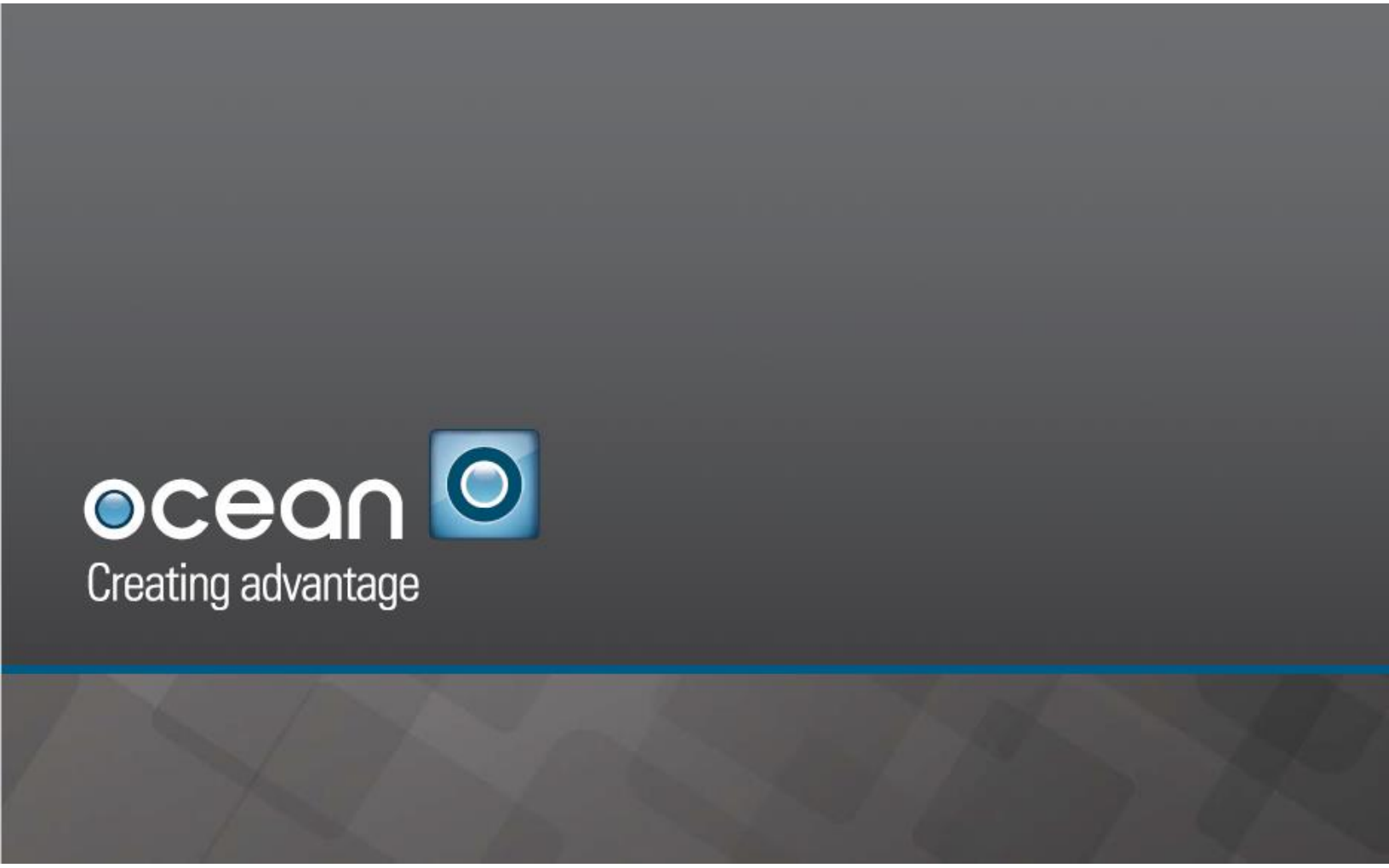
Plug-in Domain Object - Importer & Exporter

Volume 5



Ocean Software Development Framework for Techlog
Version 2023

Contents

1 Plug-in domain object

In This Chapter

Plug-in domain object overview

Ocean allows you to customize and extend the Techlog application by creating new data objects called *plug-in domain objects*. Plug-in domain objects can contribute to implementations for a use in Techlog and in the followingTechlog behaviors:

Be added to the root level of the Project tree:
Use `Slb::Ocean::Techlog::Project`

Be added to some of the Techlog data domain objects in the tree:
Use `Slb::Ocean::Techlog::DataDomainObject`

Provide a display name, icon for the plug-in domain object in the tree:
Implement `Slb::Ocean::Techlog::PluginDomainObject` macros

Relationship between plug-in domain object and data domain object:
Use `Slb::Ocean::Techlog::DataDomainObjectLink`

Notify you when the plug-in domain object changes:
Use `Slb::Ocean::Techlog::DomainObject` events

Use `Slb::Ocean::Techlog::PluginDomainObject` events

This guide explains how to implement a plug-in domain object with Ocean and the object to take part in the Techlog behaviors described above.

Ocean classes used for plug-in domain object implementation

All Ocean classes represented in the class diagram below take part in the implementation of a plug-in domain object.

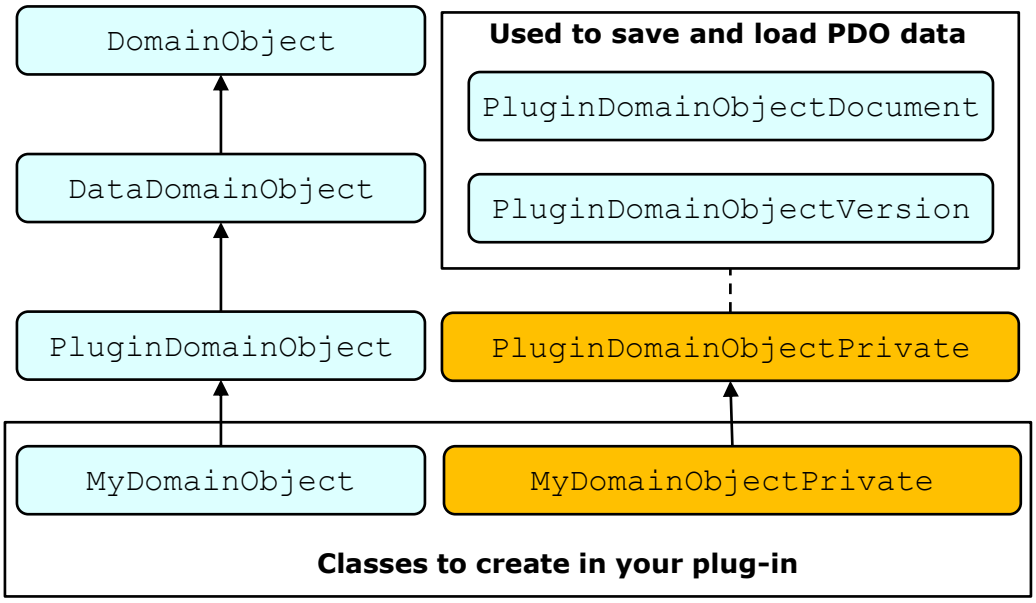


Figure 1-1 Ocean plug-in domain object classes

The blue classes are public interfaces while the orange classes are private for plug-in domain object consumers. Create two classes:

- One derived from the PluginDomainObject class (plug-in domain object public interface or facade)
- One derived from PluginDomainObjectPrivate class (plug-in domain object private implementation or PIMPL)

Plug-in domain object logic is implemented in the plug-in domain object private class as well as serialization and deserialization of plug-in domain object data in the Techlog project. The plug-in domain object facade contains only getters and setters that map the getters and setters of the PIMPL. This separated architecture allows the plug-in domain object developer to share its PDO with another plug-in developer to build its plug-in domain object in a separate DLL and provide this DLL plus an export of the plug-in domain facade header file to the plug-in domain object consumer.

If the plug-in domain object logic is changed, the plug-in domain object consumer gets the new build of the plug-in domain object DLL and there is no need to change the plug-in code.

How to implement a plug-in domain object

1) Implement a class deriving from **`PluginDomainObjectPrivate`** ocean class: this is the PIMPL (Private Implementation). It is the place where you implement the logic behind your getters and setters.
2) Override in this class **`PluginDomainObjectPrivate`** virtual functions named **`serialize`** and **`deserialize`**. They contain the code to serialize and deserialize your plug-in domain object data in the Techlog project.
3) Implement a class deriving from **`PluginDomainObject`** ocean class: this is your plugin domain object facade. It contains getters and setters that map the getters and setters of the class derived from **`PluginDomainObjectPrivate`** class. There is no logic in those methods, only some calls to the corresponding getters and setters of the PIMPL.

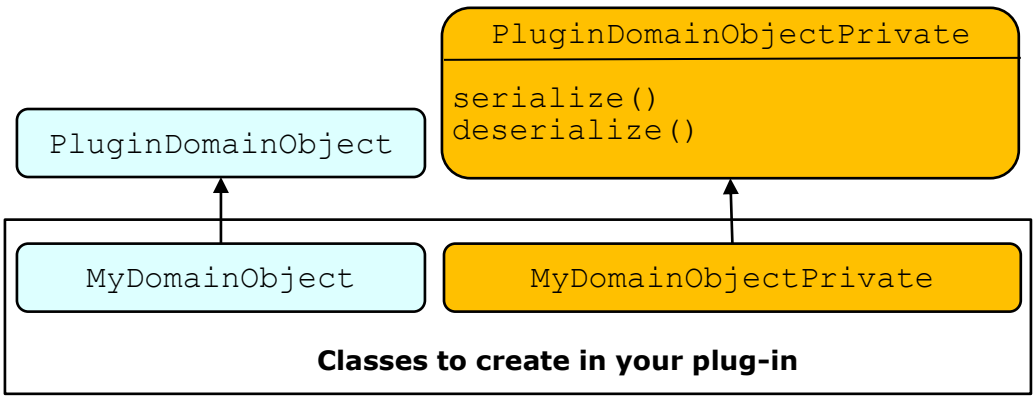


Figure 1-2 Ocean plug-in domain object implementation

Plug-in domain object private implementation (PIMPL)

The plugin domain object PIMPL must derive from **`PluginDomainObjectPrivate`** base class. This is the base class for all private implementations of the plug-in domain object.

```
class PluginDomainObjectPrivate
{
protected:
 virtual PluginDomainObjectVersion serialize() const =0;
 virtual bool deserialize(const PluginDomainObjectVersion
 &version)=0;
}
```

In the header file, declare your **`PluginDomainObjectPrivate`** object by using the macro **`PLUGIN_DOMAIN_OBJECT_PRIVATE_DECLARE`** and any getters or setters that are used in the plug-in domain object public interface.

This example declares a basic plug-in domain object PIMPL with its getters and setters.

You must declare serialization and deserialization functions that are used respectively to save and restore your plug-in domain object data.

```
#pragma once

#include "tsdkplugindomainobjectprivate.h"

class RectanglePrivate :
public Slb::Ocean::Techlog::PluginDomainObjectPrivate
{
  PLUGIN_DOMAIN_OBJECT_PRIVATE_DECLARE(RectangleFacade);

private:
  QPointF _center;
  QSizeF _size;
  QImage _image;
  QString _storageFormat;

public:
  const QPointF center() const;
  void setCenter(QPointF center);
  void setSize(QSizeF size);
  const QSizeF size() const;
  void setImage(QImage image);
  const QImage image() const;
  void setStorageFormat(QString storageFormat);
  const QString storageFormat() const;

protected:
  Slb::Ocean::Techlog::PluginDomainObjectVersion serialize()
  const override;
  bool deserialize(
  const Slb::Ocean::Techlog::PluginDomainObjectVersion&
  version) override;
};
```

In the source file, implement your **`PluginDomainObjectPrivate`** object by using the macro **`PLUGIN_DOMAIN_OBJECT_PRIVATE_DEFINE`** and the logic of getters or setters that are called from the plug-in domain object public interface.

In the example below the **`RectanglePrivate`** class is defined by propeties as:

- **`center`** that contains the center coordinates of the rectangle
- **`size`** that contains width and height of the retangle
- **`image`** that contains the **`QImage`** that fills the rectangle
- **`storageFormat`** contains the format used to save above properties in the Techlog project (used at the serialization time)

```
#pragma once
#include "RectangleFacade.h"
#include "RectanglePrivate.h"
#include "tsdkplugindomainobjectversion.h"
#include <qdom.h>

using namespace Slb::Ocean::Techlog;

PLUGIN_DOMAIN_OBJECT_PRIVATE_DEFINE(RectangleFacade,
RectanglePrivate, PluginDomainObjectPrivate);

const QPointF RectanglePrivate::center() const
{
  return _center;
}

void RectanglePrivate::setCenter(QPointF center)
{
  _center = center;
}

const QSizeF RectanglePrivate::size() const
{
  return _size;
}

void RectanglePrivate::setSize(const QSizeF size)
{
  _size = size;
}

const QImage RectanglePrivate::image() const
{
  return _image;
}
```

```
void RectanglePrivate::setImage(const QImage image)
{
  _image = image;
}

const QString RectanglePrivate::storageFormat() const
{
  return _storageFormat;
}

void RectanglePrivate::setStorageFormat(const QString
storageFormat)
{
  _storageFormat = storageFormat;
}
```

The most important step of the **`PluginDomainObjectPrivate`** implementation is the serialization and the deserialization of the plug-in domain object. It introduces two new Ocean objects:

- **`PluginDomainObjectDocument`** to save and load the data
- **`PluginDomainObjectVersion`** to define the persistence of the plug-in domain object

PluginDomainObjectDocument class

The **`PluginDomainObjectDocument`** saves your plug-in domain object data in the Techlog project at the serialization time. It can be instanciated from the **`PluginDomainObjectPrivate`** object through public functions.

```
class PluginDomainObjectPrivate
{
public:
 PluginDomainObjectDocument mainDocument() const;
 void removeMainDocument() const;

 PluginDomainObjectDocument addAttachedDocument(const QString
 &documentName) const;
 QList<PluginDomainObjectDocument> attachedDocuments() const;
 PluginDomainObjectDocument findAttachedDocument(const QString
 &documentName) const;
 PluginDomainObjectDocument getAttachedDocument(const QString
 &documentName) const;
 void removeAttachedDocument(const QString &documentName) const;
 void removeAllDocuments() const
```

```
}
```

A **mainDocument** is available during the life of your plug-in domain object and you can instanciate a PluginDomainObjectDocument through mainDocument function to save or load plug-in domain object data. The best practice is to hold in this **mainDocument** non opaque data using JSON or XML storage format.

You can add **PluginDomainObjectDocument** during the serialization of your plug-in document object through the **addAttachedDocument** function. The **attachedDocuments** are used to store opaque data. An opaque data type is a data type whose concrete data structure is not defined in an interface such as images, documents or any binary large object (BLOB).

Note: Although an attached document stores opaque data, you must give the content type at the moment that sets the **PluginDomainObjectDocument** content.

You can retrieve a **PluginDomainObjectDocument** from the **attachedDocuments** collection by its name using **findAttachedDocument** and **getAttachedDocument**. If the name of the **PluginDomainObjectDocument** does not exist in the collection, the find pattern returns an empty **PluginDomainObjectDocument** to test through **PluginDomainObjectDocument::isEmpty()** function.

The **PluginDomainObjectDocument** class provides separate API's to make a clear distinction between opaque and non opaque data.

```
class PluginDomainObjectDocument
{
public:
  bool isEmpty() const;
  QString name() const;
  bool exists() const;

  QDomDocument domDocument() const;
  void setContent(const QDomDocument &xmlContent);
  void setContent(const QDomDocument &xmlContent,
  const QString &contentType);

  QJsonObject jsonObject() const;
  void setContent(const QJsonObject &jsonContent);

  QByteArray content() const;
  void setContent(const QByteArray &content,
  const QString &contentType);

  QString contentType() const;
}
```

For non opaque data, use **setContent** dedicated functions to set the content to your **PluginDomainObjectDocument** as **QDomDocument** (XML content) and **QJSonObject** (JSON content). Getters called at the deserialization time are respectively **domDocument()** function for XML content and **jsonObject()** function for JSON content.

Opaque data are set to the **PluginDomainObjectDocument** through a **QByteArray** and its **contentType** passed to the **setContent** function.

The **contentType** function returns:

- "text/xml" string for **QDomDocument** (XML content)
    - an overload of the **setContent** function for **QDomDocument** allows you to specify another XML content type that is not "text/xml". Use the "+xml" suffix for XML **contentType**.
- "application/json" string for **QJsonObject** (JSON content)
- the **QString contentType** passed to the **setContent** function that accepts **QByteArray**

PluginDomainObjectVersion class

The **PluginDomainObjectVersion** defines the persistence of the plug-in domain object and to handle the backward and forward compatibility of a plug-in domain object.

The **PluginDomainObjectVersion** object encapsulates minor and major versions passed to the object through its constructor.

```
class PluginDomainObjectVersion
{
public:
  PluginDomainObjectVersion(quint32 major, quint32 minor);

  quint32 majorVersion() const;
  void setMajorVersion(const quint32 &majorVersion);
  quint32 minorVersion() const;
  void setMinorVersion(const quint32 &minorVersion);
}
```

Data are saved and loaded (serialization/deserialization) through a **PluginDomainObjectDocument** for a given version modeled with the **PluginDomainObjectVersion**. Every plugin domain object is saved with its type that contains the version.

Serialization / Deserialization

The following example shows how to serialize the **PluginDomainObject** properties in a non opaque document. To show both XML and JSON contents serialization, save the **PluginDomainObject** with the storage format to use.

```
PluginDomainObjectVersion RectanglePrivate::serialize() const
{
  if (_storageFormat == "XML")
```

```
  {
    QDomDocument xmlDocument("Rectangle");
    QDomElement root = xmlDocument.createElement("Rectangle");

    xmlDocument.appendChild(root);

    QDomElement center = xmlDocument.createElement("Center");
    center.setAttribute("X", _center.x());
    center.setAttribute("Y", _center.y());
    root.appendChild(center);

    QDomElement size = xmlDocument.createElement("Size");
    size.setAttribute("Width", _size.width());
    size.setAttribute("Height", _size.height());
    root.appendChild(size);

    mainDocument().setContent(xmlDocument);
  }

  if (_storageFormat == "JSON")
  {
    QJsonObject jsonObject;
    jsonObject.insert("X", _center.x());
    jsonObject.insert("Y", _center.y());
    jsonObject.insert("Width", _size.width());
    jsonObject.insert("Height", _size.height());
    mainDocument().setContent(jsonObject);
  }
  return PluginDomainObjectVersion(2016, 1);
}
```

The serialize protected function is typed and has to return a **`PluginDomainObjectVersion`**. Here the **`PluginDomainObject`** version is 2016.1.

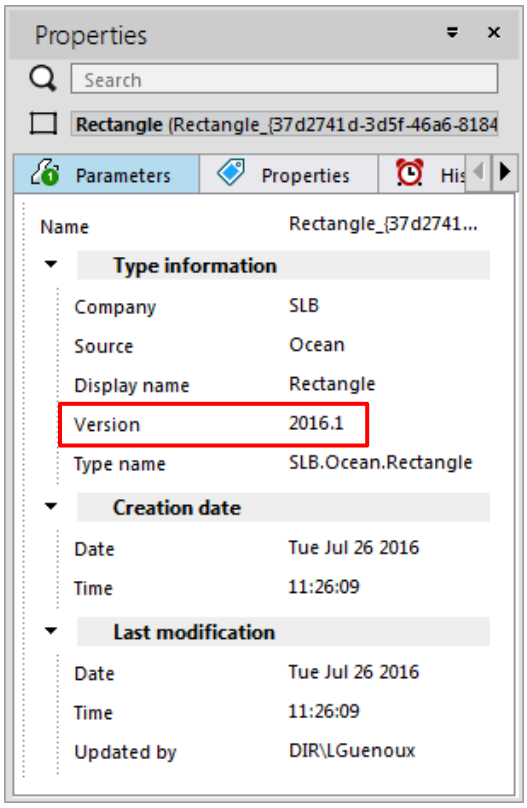


Figure 1-3 Plug-in domain object version in properties editor

A new image property is introduced in version 2016.2 of the rectangle plug-in domain object. An image is considered as opaque data and it is serialized through a **`QByteArray`** in an additional attached document named "image".

```
PluginDomainObjectVersion RectanglePrivate::serialize() const
{
  if (_storageFormat == "XML")
  {
    QDomDocument xmlDocument("Rectangle");
    QDomElement root = xmlDocument.createElement("Rectangle");

    xmlDocument.appendChild(root);

    QDomElement center = xmlDocument.createElement("Center");
    center.setAttribute("X", _center.x());
    center.setAttribute("Y", _center.y());
    root.appendChild(center);

    QDomElement size = xmlDocument.createElement("Size");
    size.setAttribute("Width", _size.width());
    size.setAttribute("Height", _size.height());
    root.appendChild(size);

    mainDocument().setContent(xmlDocument);
```

```
  }

  if (_storageFormat == "JSON")
  {
    QJsonObject jsonObject;
    jsonObject.insert("X", _center.x());
    jsonObject.insert("Y", _center.y());
    jsonObject.insert("Width", _size.width());
    jsonObject.insert("Height", _size.height());
    mainDocument().setContent(jsonObject);
  }

  QByteArray bytes;
  QDataStream stream(&bytes, QIODevice::ReadWrite);
  stream << _image;

  PluginDomainObjectDocument pluginDomainObjectDocument =
  findAttachedDocument("image");
  if (pluginDomainObjectDocument.isEmpty())
    pluginDomainObjectDocument = addAttachedDocument("image");
  pluginDomainObjectDocument.setContent(bytes,
  "application/octet-stream");

  return PluginDomainObjectVersion(2016, 2);
}
```

The **`deserialize`** protected function gives as argument a **`PluginDomainObjectVersion`** object to handle the compatibility for both versions of the rectangle plug-in domain object.

The following example show how to handle the deserialization of the two versions of the rectangle plug-in domain object with and without image.

```
bool RectanglePrivate::deserialize(const
PluginDomainObjectVersion& version)
{
  // support only 2016 major version of the plug-in domain object
  if (version.majorVersion() != 2016)
    return false;

  if (mainDocument().contentType() == "text/xml")
  {
    QDomDocument xmlDocument = mainDocument().domDocument();
    QDomElement root =
xmlDocument.elementsByTagName("Rectangle").at(0).toElement();

    QDomElement center =
    root.elementsByTagName("Center").at(0).toElement();
    _center.setX(center.attribute("X").toDouble());
```

```
    _center.setY(center.attribute("Y").toDouble());

    QDomElement size =
root.elementsByTagName("Size").at(0).toElement();
    _size.setWidth(size.attribute("Width").toDouble());
    _size.setHeight(size.attribute("Height").toDouble());

    _storageFormat = "XML";
  }

  if (mainDocument().contentType() == "application/json")
  {
    QJsonObject jsonObject = mainDocument().jsonObject();
    _center.setX(jsonObject.value("X").toDouble());
    _center.setX(jsonObject.value("Y").toDouble());
    _size.setWidth(jsonObject.value("Width").toDouble());
    _size.setHeight(jsonObject.value("Height").toDouble());

    _storageFormat = "JSON";
  }

  if (version.minorVersion() == 2)
  {
    PluginDomainObjectDocument pluginDomainObjectDocument =
    findAttachedDocument("image");
    if (pluginDomainObjectDocument.isEmpty())
      return false;

    QByteArray mainBytes = pluginDomainObjectDocument.content();
    QDataStream stream(&mainBytes, QIODevice::ReadOnly);
    QImage value;
    stream >> value;
    _image = value;
  }

  return true;
}
```

Plug-in domain object public implementation (facade)

The plugin domain object facade must derive from **`PluginDomainObject`** base class. This is the base class for all public implementations of the plug-in domain object.

In the header file, declare your **`PluginDomainObject`** object by using the macro **`PLUGIN_DOMAIN_OBJECT_DECLARE`** and any getters or setters that map the getters and setters of the **`RectanglePrivate`** class derived from the

PluginDomainObjectPrivate class. The getters and setters are exposed in the facade to plug-in domain object consumers.

To get an Ocean native **DomainObject** behavior, it is recommended to declare a **create** static function that is used to create the plug-in domain object with its name and parent passed as arguments to the function.

See the "Accessing Domain Objects" section in *Ocean for Techlog Developer Guide - Basics* for more information on **DomainObject** common patterns.

Example:

```
pragma once

#include "tsdkplugindomainobject.h"
#include "RectanglePrivate.h"

class RectangleFacade : public
Slb::Ocean::Techlog::PluginDomainObject
{
  PLUGIN_DOMAIN_OBJECT_DECLARE(RectangleFacade,
RectanglePrivate);

public:
  static RectangleFacade create(const QString& name,
  Slb::Ocean::Techlog::Project parent);

  void setCenter(QPointF center);
  const QPointF center() const;
  void setSize(QSizeF size);
  const QSizeF size() const;
  void setImage(QImage image);
  const QImage image() const;
  void setStorageFormat(QString storageFormat);
  const QString storageFormat() const;
};
```

The **PluginDomainObject** class holds several constructors. One constructor for each **PluginDomainObject** possible parent type. Possible parents are **Project**, **Well**, **Dataset**, **Variable** and **PluginDomainObject**.

```
class PluginDomainObject : public DataDomainObject
{
protected:
  PluginDomainObject(const QString &pluginDomainObjectName,
  const Project &parent, const PluginDomainObjectType &type);

  PluginDomainObject(const QString &pluginDomainObjectName,
  const Well &parent, const PluginDomainObjectType &type);

  PluginDomainObject(const QString &pluginDomainObjectName,
  const Dataset &parent, const PluginDomainObjectType &type);
```

```
  PluginDomainObject(const QString &pluginDomainObjectName,
  const Variable &parent, const PluginDomainObjectType &type);

  PluginDomainObject(const QString &pluginDomainObjectName,
  const PluginDomainObject &parent, const PluginDomainObjectType
  &type);

  PluginDomainObjectPrivate & myD();
  const PluginDomainObjectPrivate & myConstD() const;
}
```

The **myD()** protected function is called in setters of the **PluginDomainObject** facade to return the nonconst private implementation (PIMPL) of this **PluginDomainObject**.

Note: Calling this function automatically tags this instance as modified to serialize it and save it in the Techlog process. Only call this function from functions that modify the object (setters) to avoid performance issues due to excessive and unnecessary serialization.

The **myConstD()** protected function is called in getters of the **PluginDomainObject** facade to return the const private implementation (PIMPL) of this **PluginDomainObject**. Only use this function with any function that does not modify this instance (such as getters).

In the source file that implements your **PluginDomainObject**, call severals macros to ease the **PluginDomainObject** implementation.

```
PLUGIN_DOMAIN_OBJECT_ICON(RectangleFacade, "rectangle_32.png");
PLUGIN_DOMAIN_OBJECT_DISPLAY_NAME(RectangleFacade,
"Rectangle");
PLUGIN_DOMAIN_OBJECT_TYPE(RectangleFacade, "SLB", "Ocean",
"Rectangle");
PLUGIN_DOMAIN_OBJECT_DEFINE(RectangleFacade, RectanglePrivate,
PluginDomainObject);
```

The macros generate boilerplate code such as:

- creates the **PluginDomainObjectType** composed of company (vendor name), source (project) and plug-in domain object name values
- registers the plugin domain object instance with the type, icon and display name

The PluginDomainObjectType is unique by the values passed to **PLUGIN_DOMAIN_OBJECT_TYPE** macro. The **PluginDomainObjectVersion** returns by the serialize function of **PluginDomainObjectPrivate** implementation is saved with the **PluginDomainObjectType**.

Values passed to the macros are displayed in the Parameters tab of the properties editor when the plug-in domain object is selected in the Techlog project browser.

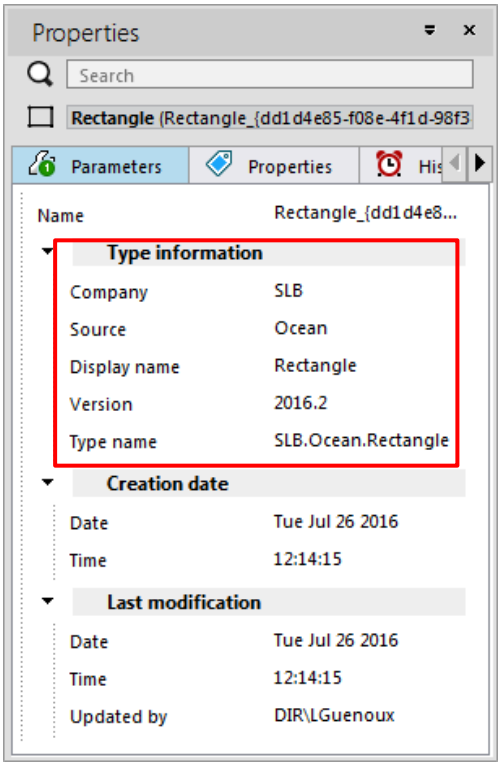


Figure 1-4 Plug-in domain object type and version in properties editor

In the **`create`** static method implementation, **`name`** and **`parent`** arguments are passed to the corresponding **`PluginDomainObject`** constructor following the **`parent`** type.

Note: A **`PluginDomainObject`** is unique by its name in the collection that belongs to the **`parent`**.

Note: The **`PluginDomainObjectType`** is created by the **`PLUGIN_DOMAIN_OBJECT_TYPE`** macro and there is no need to pass the **`PluginDomainObjectType`** to the **`PluginDomainObject`** constructor.

The following example shows how to implement the rectangle plug-in domain object facade.

```
#include "RectangleFacade.h"

using namespace Slb::Ocean::Techlog;

PLUGIN_DOMAIN_OBJECT_ICON(RectangleFacade, "rectangle_32.png");
PLUGIN_DOMAIN_OBJECT_DISPLAY_NAME(RectangleFacade,
"Rectangle");
PLUGIN_DOMAIN_OBJECT_TYPE(RectangleFacade, "SLB", "Ocean",
"Rectangle");
PLUGIN_DOMAIN_OBJECT_DEFINE(RectangleFacade, RectanglePrivate,
PluginDomainObject);
```

```
RectangleFacade RectangleFacade::create(const QString& name,
Project parent)
{
  RectangleFacade rectangle(name, parent);
  return rectangle;
}

const QPointF RectangleFacade::center() const
{
  return myConstD().center();
}

void RectangleFacade::setCenter(QPointF center)
{
  myD().setCenter(center);
}

const QSizeF RectangleFacade::size() const
{
  return myConstD().size();
}

void RectangleFacade::setSize(const QSizeF size)
{
  myD().setSize(size);
}

const QImage RectangleFacade::image() const
{
  return myConstD().image();
}

void RectangleFacade::setImage(const QImage image)
{
  myD().setImage(image);
}

const QString RectangleFacade::storageFormat() const
{
  return myConstD().storageFormat();
}

void RectangleFacade::setStorageFormat(const QString
storageFormat)
{
  myD().setStorageFormat(storageFormat);
```

```
}
```

Access and create a plug-in domain object in your Ocean plug-in

A **PluginDomainObject** is accessible from the **Project** domain object.

```
class Project : public DomainObject
{
public:
  DomainObjectCollection<PluginDomainObject>
  pluginDomainObjects() const;

  PluginDomainObject findPluginDomainObject(const QString
  &pluginDomainObjectName) const;

  PluginDomainObject getPluginDomainObject(const QString
  &pluginDomainObjectName) const;
  …
};
```

And any domain object derived from **DataDomainObject** base class as **Well**, **Dataset**, **Variable** and **PluginDomainObject**.

```
class DataDomainObject : public DomainObject
{
public:
  DomainObjectCollection<PluginDomainObject>
  pluginDomainObjects() const;

  PluginDomainObject findPluginDomainObject(const QString
  &pluginDomainObjectName) const;

  PluginDomainObject getPluginDomainObject(const QString
  &pluginDomainObjectName) const;
  …
};
```

The **pluginDomainObjects** function is a templated function that returns a collection of **PluginDomainObject** by type that belongs to a Project or a **DataDomainObject** parent object.

You can get or find a **PluginDomainObject** by its name (unique by its name in the **pluginDomainObjects** collection) and cast it to the **PluginDomainObject** type.

The following example shows how to restore the retangle plug-in domain objects saved under the main parent **Project** and draw those rectangles in a **GraphicsScene** added to a **CustomPlot** through Ocean **Image2d** domain objects.

See the "Plots" section in *Ocean for Techlog Developer Guide - Plots* for more information on **GraphicsScene**, **CustomPlot** and **Image2d** domain objects.

```
void OceanActivity::restoreRectanglesFromProject(const
GraphicsScene &graphicsScene)
{
  if ((graphicsScene.isNull()) || (graphicsScene.isErased()))
return;
```

```
  foreach(Image2d image2d,
  graphicsScene.items().toListFilteredByType<Image2d>())
  {
    if (!image2d.isErased())
      image2d.erase();
  }

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

  foreach(const PluginDomainObject& pluginDomainObject,
  project.pluginDomainObjects<RectangleFacade>())
  {
    RectangleFacade rectangle =
    pluginDomainObject.cast<RectangleFacade>();

    Image2d image2d = Image2d::create(graphicsScene);
    image2d.setCenter(rectangle.center());
    image2d.setSize(rectangle.size());
    image2d.setImage(rectangle.image());

    image2d.connect(DomainObject::DomainObjectErased, this,
    SLOT(onRectangleErased(const
    Slb::Ocean::Techlog::DomainObjectErasedArgs&)));
  }
}
```

Note: Some of the returned plug-in domain objects may not be readable by the plugin (for example due to version incompatiblity) and thus casting them into the specific facade may fail.

A **`PluginDomainObject`** is created through its **`create`** static function declared and implemented in the **`PluginDomainObject`** facade.

The following example shows how the rectangle plug-in domain object is created. The plug-in allows you to draw an **`Image2d`** in a **`GraphicsScene`** added to a **`CustomPlot`** and saves the properties of the **`Image2d`** (center coordinates, width, height and **`QImage`**) into a rectangle plug-in domain object when you release the mouse button.

```
void OceanActivity::saveRectangle()
{
  Image2d image2d = _currentImage2d.at(0);

  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  QString path = _imageLineEdit->text();
  QImage qimg(path);
```

```
  if (qimg.load(path))
    image2d.setImage(qimg);

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

  QString rectangleName = "Rectangle_" +
  image2d.droid().toString();

  RectangleFacade rectangle =
  RectangleFacade::create(rectangleName, project);

  rectangle.setStorageFormat(
  _storageFormatComboBox->currentText());
  rectangle.setCenter(image2d.center());
  rectangle.setSize(image2d.size());
  rectangle.setImage(image2d.image());

  rectangle.connect(DomainObject::DomainObjectErased, this,
  SLOT(onRectangleErased(const
  Slb::Ocean::Techlog::DomainObjectErasedArgs&)));

  lock.release();

  _widget->close();
}
```

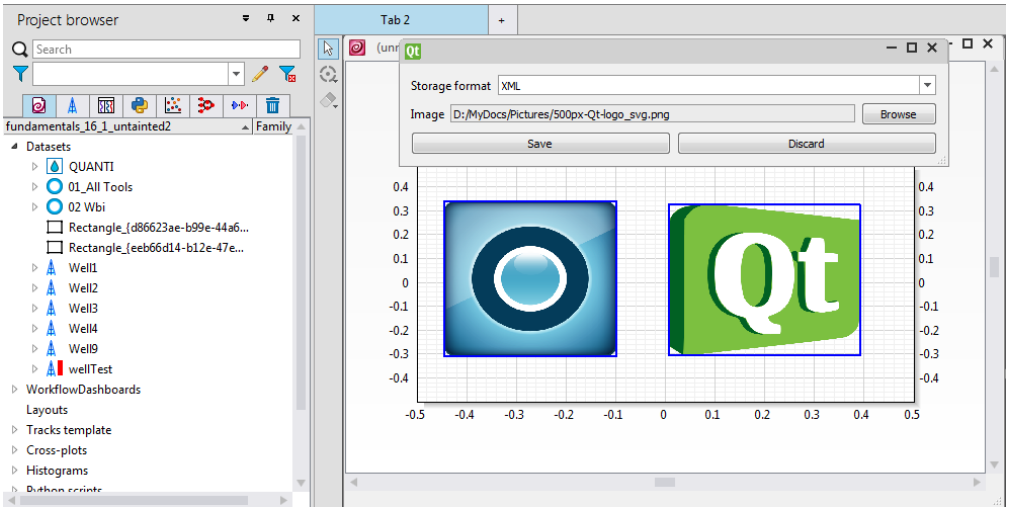


Figure 1-5 Rectangle plug-in domain object

Plug-in domain object signals

As the **PluginDomainObject** class derived from the **DomainObject** base class, a **PluginDomainObject** can subscribe on **DomainObject** generic events and be notified when the **PluginDomainObject** is changed, deleted or renamed.

```
class DomainObject
{
protected:
  ...
  enum EventType
   {
      DomainObjectChanged,
      DomainObjectErased,
      DomainObjectRenamed
   };
}
```

The **PluginDomainObject** class implements is own signals as well.

```
class PluginDomainObject : public DataDomainObject
{
protected:
  ...
  enum EventType
   {
      PluginDomainObjectCreated,
      PluginDomainObjectDataChanged
   };
}
```

A **PluginDomainObject** object can subscribe to those signals through the **connect** method inherited from the **DomainObject** base class.

- **PluginDomainObjectCreated** signal is emitted when a child **PluginDomainObject** is added to the **PluginDomainObject**.
- **PluginDomainObjectDataChanged** signal is emitted when the **PluginDomainObject** bulk data size is changed.

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

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

```
private slots:
  void onPluginDomainObjectCreated(const
Slb::Ocean::Techlog::PluginDomainObjectCreatedArgs &args);
  void onPluginDomainObjectDataChanged(const
Slb::Ocean::Techlog::PluginDomainObjectDataChangedArgs &args);
```

This example connects to those signals.

```
void OceanActivity::run()
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);
  Project project = Session::current().mainProject();

  foreach(const PluginDomainObject& pluginDomainObject,
     project.pluginDomainObjects<RectangleFacade>())
  {
    RectangleFacade rectangle =
     pluginDomainObject.cast<RectangleFacade>();

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

    rectangle.connect(
     PluginDomainObject::PluginDomainObjectDataChanged, this,
     SLOT(onPluginDomainObjectDataChanged(const
     Slb::Ocean::Techlog::PluginDomainObjectDataChangedArgs
     &)));
  }
  lock.release();
}
```

PluginDomainObjectDataChanged signal

The **`PluginDomainObjectDataChanged`** signal includes a **`PluginDomainObjectDataChangedArgs`** argument that gives the affected object (**`PluginDomainObject`** where the data has been changed).

```
class PluginDomainObjectDataChangedArgs : public
SignalArgsT<PluginDomainObject>
{
};
```

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

```
void OceanActivity::onPluginDomainObjectDataChanged(const
Slb::Ocean::Techlog::PluginDomainObjectDataChangedArgs &args)
{
  PluginDomainObject changedPDO =
    args.sender().cast<PluginDomainObject>();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
changedPDO);
```

```
  qWarning() << "PluginDomainObject \"" << changedPDO.name()
    << "\" bulk data has changed";

  lock.release();
}
```

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 OceanActivity::onPluginDomainObjectCreated(const
Slb::Ocean::Techlog::PluginDomainObjectCreatedArgs &args)
{
  PluginDomainObject newPDO = args.newPluginDomainObject();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, newPDO);

  PluginDomainObject parentPDO =
    args.sender().cast<PluginDomainObject>();

  qWarning() << "A new child PluginDomainObject \""
    << newPDO.name() << "\" to the parent PluginDomainObject "
    << "\"" << parentPDO.name() << "\"";

  lock.release();
}
```

Relationship between plug-in domain object and data domain object

Ocean allows you to specify a reference from the **PluginDomainObjectPrivate** object to any object derived from **DataDomainObject** base class as **Well**, **Dataset**, **Variable** and **PluginDomainObject**. The relationship between the two objects is modeled in Ocean with **DataDomainObjectLink** object.

```
class DataDomainObjectLink
```

```
{
public:
  DataDomainObjectLink(const QString &name,
  const DataDomainObject &linkedDataDomainObject);

  DataDomainObjectLink(const QString &name,
  const Droid &linkedDataDomainObjectDroid);

  bool isEmpty() const;
  QString name() const;
  DataDomainObject findLinkedDataDomainObject() const;
};
```

A **`DataDomainObjectLink`** is a non domain object built though its constructor passing the **`name`** of this link and the linked **`DataDomainObject`** instance or droid.

This **`DataDomainObjectLink`** can be added, retrieved, removed and listed from the **`PluginDomainObjectPrivate`** through dedicated public functions.

```
class PluginDomainObjectPrivate
{
public:
  QList<DataDomainObjectLink> getLinks() const;

  void addLink(const DataDomainObjectLink &link);
  void setLinks(const QList<DataDomainObjectLink> &links);

  void removeLink(const QString &linkName);

  DataDomainObjectLink findLink(const QString &linkName) const;
  DataDomainObjectLink findLink(const Droid &linkedObjectDroid)
  const;

  bool linkExists(const QString &linkName) const;
  bool linkExists(const Droid &linkedObjectDroid) const;

  void clearLinks();
};
```

The following example shows how to create links between the rectangle plug-in domain object and wells in the Techlog project.

```
  lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  PluginDomainObject pluginDomainObject =
  project.findPluginDomainObject<RectangleFacade>
  (rectangleName);

  if (pluginDomainObject.isNull())
  {
    lock.release();
    return;
  }
```

```
  RectangleFacade myRectangle =
  pluginDomainObject.cast<RectangleFacade>();

  int nbLink = 0;
  foreach(Well well, project.wells())
  {
    DataDomainObjectLink link(QString("link%1").arg(nbLink),
    well.droid());

    myRectangle.myD().addLink(link);

    well.connect(DomainObject::DomainObjectRenamed, this,
    SLOT(onWellRenamed(const
    Slb::Ocean::Techlog::DomainObjectRenamedArgs&)));

    nbLink++;
  }

  lock.release();
```

In this example the plug-in listens to the well-renamed event, so if a well is renamed, the **`DataDomainObjectLink`** created between the plug-in domain object and the well is not valid anymore and must be removed. The following example shows how to retrieve the **`DataDomainObjectLink`** for the well renamed and remove it.

Note: If the **`DataDomainObject`** referenced object is erased, the **`DataDomainObjectLink`** is automatically removed with it.

```
void OceanActivity::onWellRenamed(const
Slb::Ocean::Techlog::DomainObjectRenamedArgs &args)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  DomainObject domainObject = args.sender();
  Well well = domainObject.cast<Well>();

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

  foreach(const PluginDomainObject& pluginDomainObject,
  project.pluginDomainObjects<RectangleFacade>())
  {
    RectangleFacade rectangle =
    pluginDomainObject.tryCast<RectangleFacade>();

    if (rectangle.myD().linkExists(well.droid()))
    {
      QString linkName =
      rectangle.myD().findLink(well.droid()).name();
```

```
      rectangle.myD().removeLink(linkName);

      qWarning() << "The link " << linkName
      << " has been removed " << rectangle.name()
      << " and " << well.name();
    }
  }

  lock.release();
}
```

2 Import / Export extensibility

In This Chapter

Import / Export overview

Techlog can integrate many types of data, including log data, point data, seismic, deviation files, and image files. Techlog also supports a variety of the most common formats, including LAS, GeologASCII, DLIS/LIS, ASCII file with the wizard, core images, TechCSV, and the Techlog format (XML).

In addition, you can export data or any object created in Techlog (such as plots and layouts) from Techlog to formats such as LAS, DLIS, GeologASCII, ASCII file, TechCSV, and the Techlog format (XML).

Techlog data import

You can import data in different ways. The method that you use depends on the type of file and your preference of a process.

- Drag the files to be imported into Techlog (LAS, DLIS, Techlog XML, CSV files).
- Select **Project** > **Import** from the main Techlog window.
- Select **Home** > **Import**

Whatever action opens the import buffer in the Project browser window. The import buffer stores in a temporary space all of the data that you choose to import.

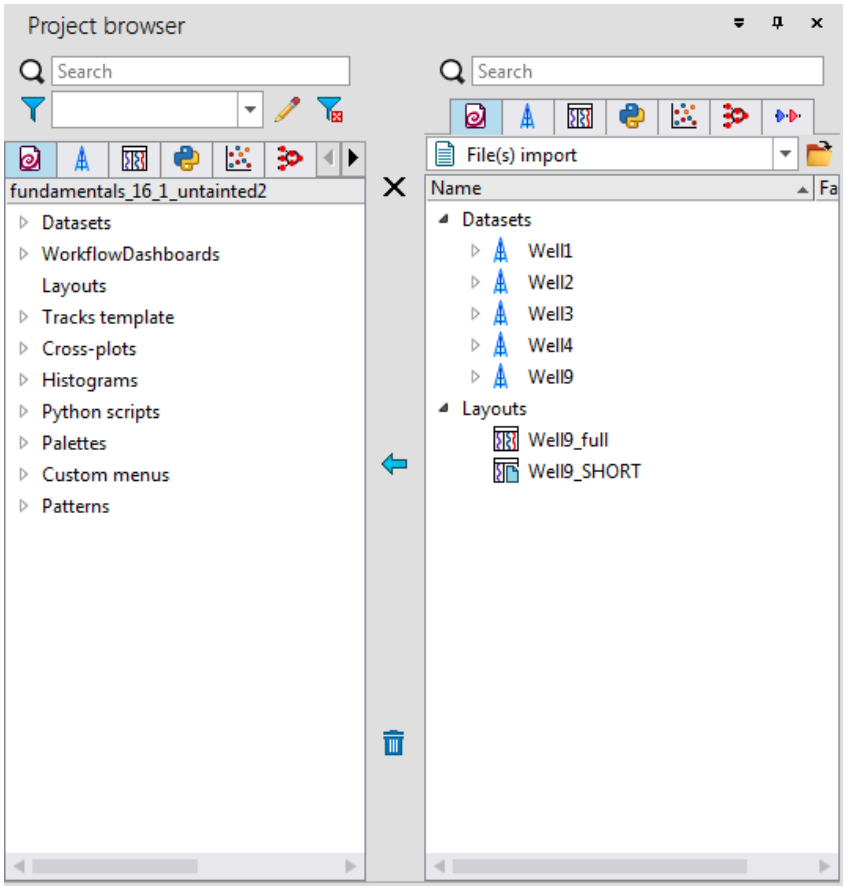
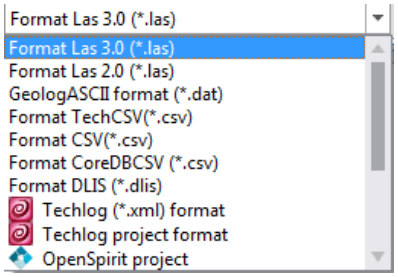


Figure 2-1 Project browser (left) and the import buffer (right)

Techlog data export

Techlog export features allow you to export data to standard formats as follows:

1. Click **Project** > **Export** to open the export buffer.
2. From the drop-down list, select the format in which you wish to export the data.



3. In the Project browser, select the objects to export and click  .

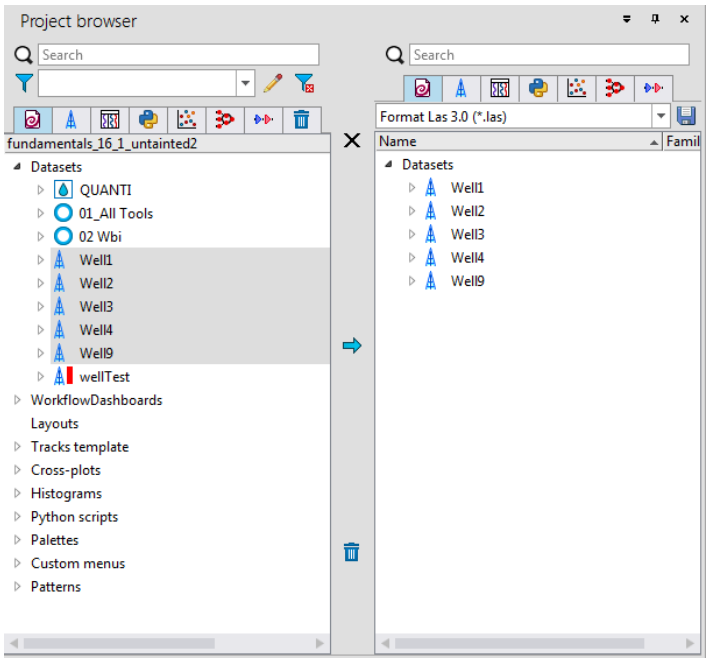


Figure 2-2 Project browser (left) and the export buffer (right)

Plug-in importer / exporter

Ocean provides public API to extend import and export functionalities of Techlog platform.

A plug-in can register its own import and export functions:

- ability to register to File import through the **`FileImporter`** class
- ability to register to Mime import through the **`MimeImporter`** class
- ability to register to Project export through the **`Exporter`** class

If the plug-in is activated when import or export buffer are opened, the importer and exporter registered by the plug-in are added to the drop-down list that contains all import and export formats available natively in Techlog.

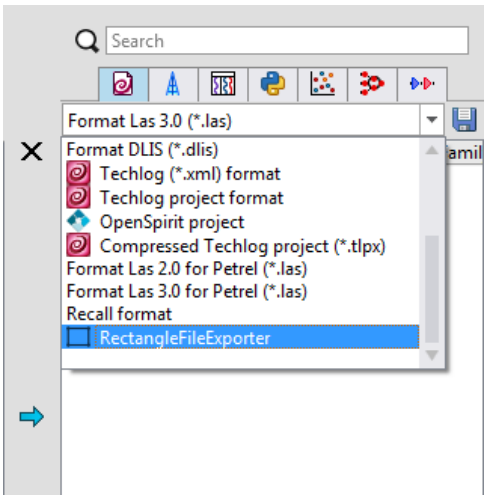


Figure 2-3 Plug-in exporter added to export formats list

Why differentiating File import from Mime import?

- You can drag and drop several different files into Techlog (single shot), some would be able to be processed by the platform itself, some others can be processed by different plug-ins.
- Mime import mainly concerns the drag and drop from an application (eg. Recall superview, OpenSpirit desktop) into Techlog. In such case, we expect only one "function" to perform the import.

1) Drop a file into Techlog to launch the plug-in that registers the importer able to import the data. If one native import function can process the action (eg. XML format).
2) Select the best importer from a list of importers (native or plug-in) that are able to perform the import.

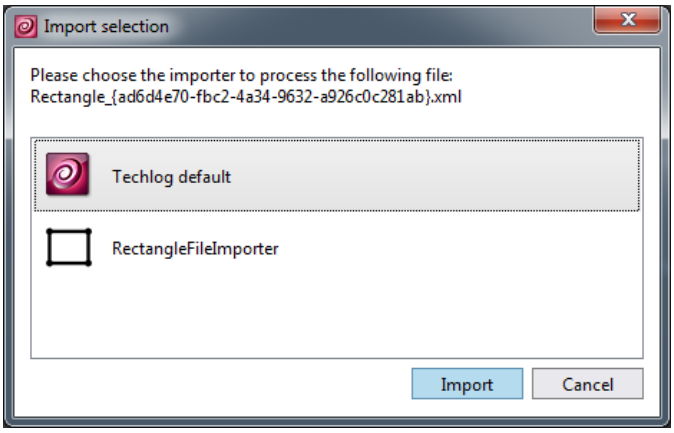


Figure 2-4 Import selection window

How to implement a plug-in importer / exporter

1) Implement a class deriving from:
    - **`FileImporter`** or **`MimeImporter`** Ocean class, depending of the type of plug-in importer that you want to create.
    - **`Exporter`** Ocean class for a plug-in exporter
2) Declare and implement a constructor for the class that takes as argument:
    - the identity of the importer/exporter (modeled in Ocean through **`ImportExportIdentity`** class)
    - for importers only, the list of file extensions accepted by the **`FileImporter`** or the mime format of the **`MimeImporter`**, and a regex pattern to read the first 1024 bytes of the file (not mandatory, can be an empty pattern)
3) Override **`doImport`** / **`doExport`** public function that is called once a file is triggered by Techlog and this importer/exporter instance has been selected.
4) Register the plug-in importer/exporter to Techlog in **`IPlugin::services_impl`** declared and overridden in the main plug-in class derived from **`PluginIdentity`** class.

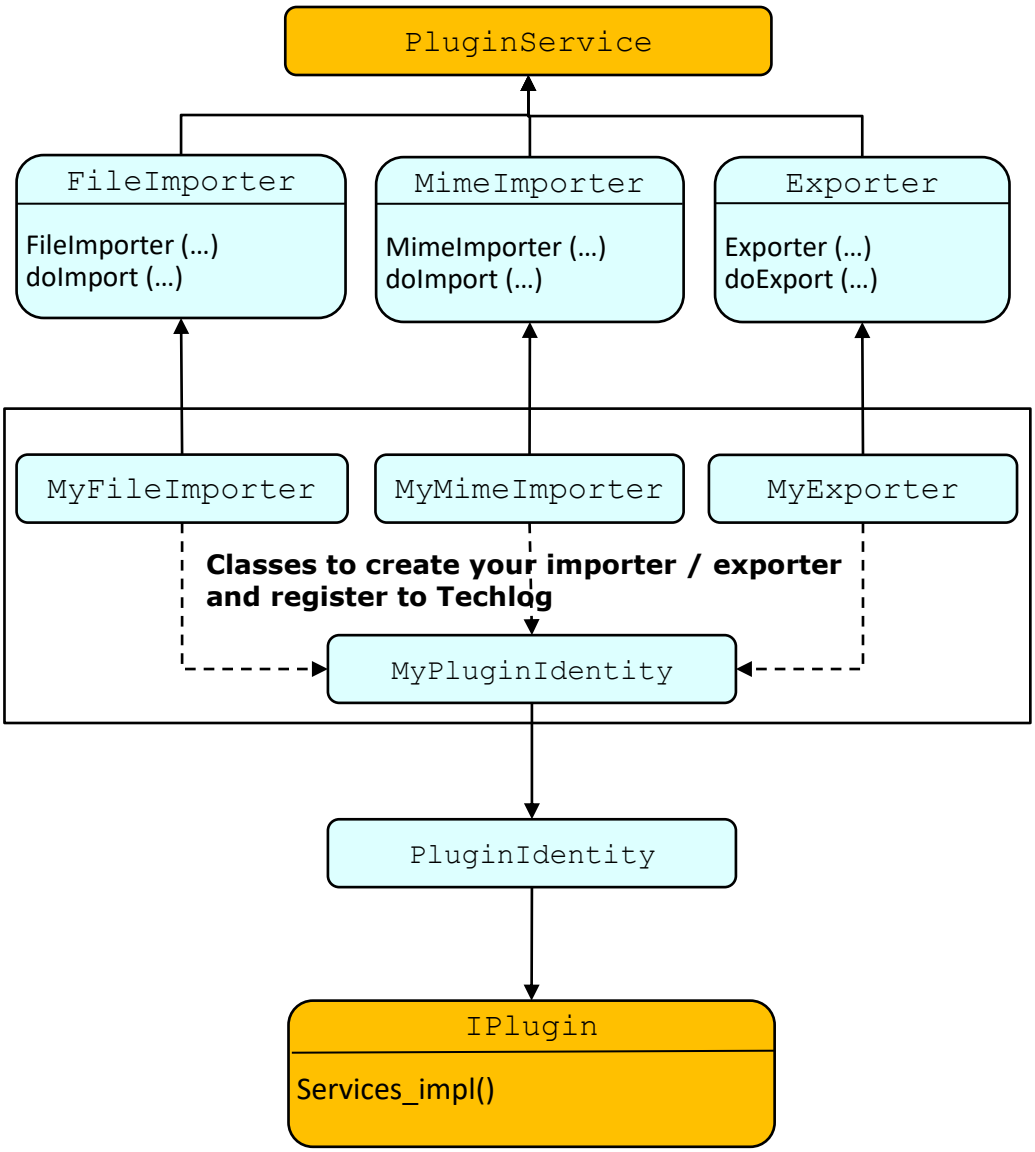


Figure 2-5 Ocean importer / exporter implementation

FileImporter implementation

The plug-in file importer must derive from the **`FileImporter`** class.

```
class FileImporter
{
protected:
  FileImporter(const ImportExportIdentity &identity, const
  QStringList &fileExtensions, const QRegExp &pattern);

  const ImportExportIdentity & identity() const;

  const QStringList & fileExtensions() const;
```

```
  const QRegExp & pattern() const;

  virtual ImportExportReturnStatus doImport(Project &project,
  const QString &filePath)=0;
}
```

It is recommended to declare a **`create`** static method that is used to create the identity of the importer through the **`ImportExportIdentity`** object and instantiate the **`FileImporter`** object (with its overidden constructor) that is registered as service into Techlog.

See the "ImportExportIdentity class" on page 2-15 for more information on how to declare a unique identity for your plug-in importer to be registered as service into Techlog.

The following example shows a plug-in importer header file:

```
#pragma once

#include "tsdkfileimporter.h"

class RectangleFileImporter : public
Slb::Ocean::Techlog::FileImporter
{
public:
  static QSharedPointer<RectangleFileImporter> create();

  virtual Slb::Ocean::Techlog::ImportExportReturnStatus
  doImport(Slb::Ocean::Techlog::Project& project, const QString&
  filePath) override;

private:
  RectangleFileImporter(const
  Slb::Ocean::Techlog::ImportExportIdentity& id,
  const QStringList& fileExtensions, const QRegExp& patten);
};
```

The example shows how to implement the **`create`** static method and instantiate the plug-in **`FileImporter`** unique by its **`uid`**, **`name`** and **`version`** passed to the **`ImportExportIdentity`** object. The XML extension is passed to **`fileExtensions`** list argument.

1) Drop a XML fileto launch the plug-in registering the importer to import the XML data.

```
#pragma once

#include "RectangleFileImporter.h"
#include "RectangleFacade.h"
#include <qdom.h>

using namespace Slb::Ocean::Techlog;
```

```
RectangleFileImporter::RectangleFileImporter(const
ImportExportIdentity& id, const QStringList& fileExtensions,
const QRegExp& patten): FileImporter(id, fileExtensions, patten)
{
}

QSharedPointer<RectangleFileImporter>
RectangleFileImporter::create()
{
  static const QUuid uid =
  QUuid("F0138A41-25ED-4BD6-AEF8-11AD9F51E546");
  static const QString name =
  QString::fromLatin1("RectangleFileImporter");
  static const QString version = QString::fromLatin1("1.0");
  static const QStringList extension = QStringList() <<
  QString::fromLatin1("xml");
  static const QRegExp pattern = QRegExp();

  return QSharedPointer<RectangleFileImporter>(new
  RectangleFileImporter(ImportExportIdentity(uid, name,
  QIcon("rectangle_32.png"), version), extension, pattern));
}
```

2) Override and implement the **`doImport`** function that is called once a file is triggered by Techlog.

The following example is a **`doImport`** implementation for the rectangle plug-in domain object. The Rectangle properties are stored in a XML file and the image that fills the rectangle into a PNG file.

See "How to implement a plug-in domain object" on page 1-3 for more information on rectangle plug-in domain object.

```
ImportExportReturnStatus
RectangleFileImporter::doImport(Project& project, const QString&
filePath)
{
  QFileInfo fileInfo(filePath);

   if(!FileImporter::fileExtensions()
   .contains(fileInfo.suffix(), Qt::CaseInsensitive))
    return ImportExportReturnStatus(
      ImportExportReturnStatus::StatusError,
QString::fromLatin1("The extension <b>%1</b> is not
supported").arg(fileInfo.suffix()));

  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  RectangleFacade rectangle =
```

```
  RectangleFacade::create(fileInfo.baseName(), project);
  rectangle.setStorageFormat("XML");

  QFile file(fileInfo.absoluteFilePath());
  if (!file.open(QIODevice::ReadOnly))
    return ImportExportReturnStatus(
      ImportExportReturnStatus::StatusError,
QString::fromLatin1("Cannot read the file
").arg(fileInfo.absoluteFilePath()));

  QDomDocument xmlDocument;
  if (!xmlDocument.setContent(&file))
  {
    file.close();
    return ImportExportReturnStatus(
      ImportExportReturnStatus::StatusError,
QString::fromLatin1("Unable to parse the <b>%1</b>
file").arg(fileInfo.completeBaseName()));
  }

  QDomElement root = xmlDocument.elementsByTagName("Rectangle")
  .at(0).toElement();

  QDomElement center =
  root.elementsByTagName("Center").at(0).toElement();
  rectangle.setCenter(
  QPointF(center.attribute("X").toDouble(),
  center.attribute("Y").toDouble()));

  QDomElement size =
  root.elementsByTagName("Size").at(0).toElement();
  rectangle.setSize(QSizeF(size.attribute("Width").toDouble(),
  size.attribute("Height").toDouble()));

  file.close();

  QString imagePath = fileInfo.absolutePath()
  + QDir::separator() + fileInfo.baseName() + ".png";

  QImage qimg(imagePath);
  if (qimg.load(imagePath))
    rectangle.setImage(qimg);

  lock.release();

  if (rectangle.isNull())
```

```
    return ImportExportReturnStatus(
      ImportExportReturnStatus::StatusError,
QString::fromLatin1("Unable to create the <b>%1</b> domain
object").arg(fileInfo.baseName()));

  return ImportExportReturnStatus(
    ImportExportReturnStatus::StatusSuccess);
}
```

The **doImport** function returns a status on the import modeled in Ocean with the **ImportExportReturnStatus** class.

See "ImportExportReturnStatus class" on page 2-16 for more information on import return status.

MimeImporter implementation

The plug-in mime importer has to derive from the **MimeImporter** class.

```
class MimeImporter
{
protected:
  MimeImporter(const ImportExportIdentity &identity, const
  QString &mimeFormat, const QRegExp &pattern);

  const ImportExportIdentity & identity() const;

  const QString & mimeFormat() const;

  const QRegExp & pattern() const;

  virtual ImportExportReturnStatus doImport(Project &project,
  const QByteArray &mimeContent)=0;
}
```

It is recommanded to declare a **create** static method that is used to create the identity of the mime importer through the **ImportExportIdentity** object and instantiate the **MimeImporter** object (with its overidden constructor) that is registered as service into Techlog.

See the "ImportExportIdentity class" on page 2-15 for more information on how to declare a unique identity for your plug-in importer to be registered as service into Techlog.

The following example shows a plug-in importer header file:

```
#pragma once

#include "tsdkmimeimporter.h"

class WMVMimeImporter : public Slb::Ocean::Techlog::MimeImporter
{
public:
  static QSharedPointer<WMVMimeImporter> create();
```

```
  virtual Slb::Ocean::Techlog::ImportExportReturnStatus
  doImport(Slb::Ocean::Techlog::Project& project, const
  QByteArray &mimeContent) override;

private:
  WMVMimeImporter(const
  Slb::Ocean::Techlog::ImportExportIdentity&
  id, const QString &mimeFormat, const QRegExp& patten);
};
```

The **`MimeImporter`** constructor requires the supported **`mimeFormat`**. When dropping from an external application into Techlog, the **`mimeFormat`** will be used to know if the plug-in is a potential candidate to import the file.

Note: A **`MimeImporter`** is not added to the drop-down list that contains all import formats available natively in Techlog. This list is for **`FileImporter`** only.

With the optional regex **`pattern`**, Techlog tries to find the specified pattern inside the mime content. It can be used if you want to allow to import a mime format with a special pattern such as a CSV file from Excel.

The **`doImport`** virtual method is called once a mime import is triggered by Techlog and this mime importer instance matches with the mime format/content.

The following example shows a MimeImporter implementation that allows to import WMV content from Windows Media Player application.

```
#pragma once

#include "WMVMimeImporter.h"
#include "WMVDomainObject.h"
#include <qdom.h>

using namespace Slb::Ocean::Techlog;

WMVMimeImporter::WMVMimeImporter(const ImportExportIdentity&
id, const QString &mimeFormat, const QRegExp& patten)
  : MimeImporter(id, mimeFormat, patten)
{
}

QSharedPointer<WMVMimeImporter> WMVMimeImporter::create()
{
  static const QUuid uid =
  QUuid("F0138A41-25ED-4BD6-AEF8-11AD9F51E547");
  static const QString name =
  QString::fromLatin1("WMVMimeImporter");
  static const QString version = QString::fromLatin1("1.0");
  static const QString mimeFormat =
  QString::fromLatin1("video/x-ms-wmv");
  static const QRegExp pattern = QRegExp();
```

```
  return QSharedPointer<WMVMimeImporter>(new

WMVMimeImporter(ImportExportIdentity(uid, name,
                        QIcon("wmv_32.png"), version),
mimeFormat, pattern));
}

ImportExportReturnStatus WMVMimeImporter::doImport(Project&
project, const QByteArray &mimeContent)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  Droid *droid = new Droid();
  WMVDomainObject wmvDomainObject =
WMVDomainObject::create(QString("WMV_%1").arg(droid->toString(
)),project);
  wmvDomainObject.setFileContent(mimeContent);

  lock.release();

  return ImportExportReturnStatus(
  ImportExportReturnStatus::StatusSuccess);
}
```

The **doImport** function returns a status on the import modeled in Ocean with the **ImportExportReturnStatus** class.

See "ImportExportReturnStatus class" on page 2-16 for more information on import return status.

Exporter implementation

The plug-in exporter has to derive from the **Exporter** class.

```
class Exporter
{
protected:
  Exporter(const ImportExportIdentity &identity);

  const ImportExportIdentity & identity() const;

  virtual ImportExportReturnStatus doExport(Project &project)=0;
}
```

It is recommended to declare a **create** static method to create the identity of the exporter through the **ImportExportIdentity** object and instantiate the **Exporter** object (with its overidden constructor) that is registered as a service into Techlog.

See the "ImportExportIdentity class" on page 2-15 for more information on how to declare a unique identity for your plug-in exporter to be registered as service into Techlog.

The following example shows a plug-in exporter header file:

```
#pragma once

#include "tsdkexporter.h"
#include "RectangleFacade.h"

class RectangleFileExporter : public
Slb::Ocean::Techlog::Exporter
{
public:
  static QSharedPointer<RectangleFileExporter> create();

  virtual Slb::Ocean::Techlog::ImportExportReturnStatus
  doExport(const Slb::Ocean::Techlog::Project& project)
  override;

private:
  RectangleFileExporter(const
  Slb::Ocean::Techlog::ImportExportIdentity& id);
};
```

The example shows how to implement the **`create`** static method and instantiate the plug-in **`Exporter`** unique by its **`uid`**, **`name`** and **`version`** passed to the **`ImportExportIdentity`** object.

```
#pragma once

#include "RectangleFileExporter.h"
#include "tsdkplugindomainobject.h"
#include "RectangleFacade.h"
#include "tsdkprogressdialog.h"
#include <qdom.h>

using namespace Slb::Ocean::Techlog;

RectangleFileExporter::RectangleFileExporter(const
ImportExportIdentity &id)
  : Exporter(id)
{
}

QSharedPointer<RectangleFileExporter>
RectangleFileExporter::create()
{
  static const QUuid uid =
  QUuid("66B28F06-7B8A-4BEB-B646-207E53DD9599");
  static const QString name =
  QString::fromLatin1("RectangleFileExporter");
```

```
  return QSharedPointer<RectangleFileExporter>(
    new RectangleFileExporter(ImportExportIdentity(uid, name,
   QIcon("rectangle_32.png"))));
}
```

Override and implement the **`doExport`** function that is called once a file is triggered by Techlog and this exporter instance has been selected.

The following example shows the **`doExport`** implementation for the rectangle plug-in domain object. The properties of the rectangle plug-in domain object as center and size are exported to an XML file and the image that fills the rectangle to a PNG file.

See "How to implement a plug-in domain object" on page 1-3 for more information on rectangle plug-in domain object.

```
ImportExportReturnStatus RectangleFileExporter::doExport(const
Project& project)
{
  Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

  QString destFolder = QFileDialog::getExistingDirectory();

  if (destFolder.isEmpty())
    return ImportExportReturnStatus(
      ImportExportReturnStatus::StatusError,
      QString::fromLatin1("Operation canceled"));

  QString fileName;

  foreach(const PluginDomainObject& pluginDomainObject,
  project.pluginDomainObjects<RectangleFacade>())
  {
    RectangleFacade rectangle =
    pluginDomainObject.cast<RectangleFacade>();

    fileName = rectangle.name();

    rectangle.image().save(destFolder + QDir::separator() +
    fileName + ".png");

    QDomDocument xmlDocument("Rectangle");
    QDomElement root = xmlDocument.createElement("Rectangle");
    xmlDocument.appendChild(root);

    QDomElement center = xmlDocument.createElement("Center");
    center.setAttribute("X", rectangle.center().x());
    center.setAttribute("Y", rectangle.center().y());
    root.appendChild(center);
```

```
    QDomElement size = xmlDocument.createElement("Size");
    size.setAttribute("Width", rectangle.size().width());
    size.setAttribute("Height", rectangle.size().height());
    root.appendChild(size);

    QFile toFile(destFolder + QDir::separator() + fileName +
    ".xml");
    if (toFile.open(QIODevice::WriteOnly))
    {
      QTextStream stream(&toFile);
      stream << xmlDocument.toString();
    }
    else
    {
      lock.release();
      toFile.close();
      return ImportExportReturnStatus(
        ImportExportReturnStatus::StatusError, QString("Failed
to save file: %1").arg(fileName.isEmpty() ? "Empty file name" :
fileName));
    }

    toFile.close();
  }

  lock.release();

  return ImportExportReturnStatus(
    ImportExportReturnStatus::StatusSuccess);
}
```

The **doExport** function returns a status on the export modeled in Ocean with the **ImportExportReturnStatus** class.

See "ImportExportReturnStatus class" on page 2-16 for more information on export return status.

ImportExportIdentity class

The **ImportExportIdentity** class is a structure to describe the identity of an importer or exporter.

```
class ImportExportIdentity
{
public:
  ImportExportIdentity(const QUuid &uid, const QString &name,
  const QIcon &icon=QIcon(), const QString
  &version=QString::null);

  const QUuid & uid() const;
  const QString & name() const;
```

```
  const QIcon & icon() const;
  const QString & version() const;
}
```

ImportExportIdentity object members are:

- the **uid** is the unique identifier of the importer/exporter

  Note: To generate a unique identifier in Visual Studio, go to **Tools** menu -> **Create GUID** item -> **static const struct GUID** option.

- the **name** of the importer/exporter that is displayed in the import/export buffer available formats list
- the **icon** of the importer/exporter that is displayed close to the **name** in the import/export buffer available formats list

  Note: The minimum icon size allowed is 32x32. If the image size passed to the **QIcon** is lower than 32x32 the **ImportExportIdentity** constructor throws an exception.

- the **version** of the importer/exporter

ImportExportIdentity member values are set through the constructor of the class. The **icon** and **version** can be passed as null.

Note: Several plugins can register the same importer/exporter unique by the **ImportExportIdentity** (same **uid**, same **name** and same **version**). If a Plugin registers an importer/exporter with the same **uid** than another one which does not have the same **name** or the same **version**, the import/export registration faisl and a message is displayed.

ImportExportReturnStatus class

The **ImportExportReturnStatus** class is a structure to provide information on the return status of the **doImport** and **doExport** functions.

See "FileImporter implementation" on page 2-6 for more information on how to implement a **doImport** function.

```
class ImportExportReturnStatus
{
public:
  ImportExportReturnStatus(Status status, const QString
  &errorMessage=QString::null);

  Status status() const;
  const QString & errorMessage() const;
}
```

ImportExportIdentity object members are:

- the **status** enum value (success or error) of import/export
- the **errorMessage** that is an optional string passed to the **ImportExportReturnStatus** constructor
  - the **errorMessage** is displayed to the user in case of failure during the import/export

Register importer / exporter

FileImporter, MimeImporter and Exporter are registered into Techlog in the **`IPlugin::services_impl`** that is declared and overridden in your main plug-in class derived from the **`PluginIdentity`** class.

See the "Writing your first plug-in" section in *Ocean for Techlog Getting Started Guide* for more information on virtual methods to be overridden in the main plug-in class.

```
class IPlugin
{
protected:
  virtual QList<QSharedPointer<PluginService>> services_impl()
  const;
  …
};
```

The following example shows how to add importer and exporter to the list of **`PluginService`** known by Techlog when the plug-in is activated in the Techlog module manager.

```
QList<QSharedPointer<Slb::Ocean::Techlog::PluginService>>
VolumePluginDomainObjects::services_impl() const
{
  static QList<QSharedPointer<PluginService>> services;

  if (!services.isEmpty())
    return services;

  // Register the Import/Export services
  services << RectangleFileExporter::create();
  services << RectangleFileImporter::create();
  services << WMVMimeImporter::create();
  return services;
}
```