Getting Started with Ocean for Petrel

For Geoscientists and Software Developers

Table of Contents

WELCOME TO OCEAN FOR PETREL

Ocean is an application development framework with the capability to work across data domains. It provides services, components, and a common graphical user interface that enables efficient integration between applications. It allows application developers to interact with Ocean applications like Petrel.

Ocean applications are loaded dynamically as .NET assemblies. These assemblies, the building blocks of Ocean, contain modules. Plug-ins organizes and contains modules.

Ocean for Petrel provides Visual Studio extension for developer to start building plug-in guided by wizards and developer tools. Refer to wizard and developer tool documentation.

Ocean Architecture

The Ocean architecture consists of three levels: the Core, the Services, and the product family. For model-centric applications, the product family is Petrel. Ocean modules are managed by the Core layer. They interact with all levels of the framework, as shown in Figure 1:
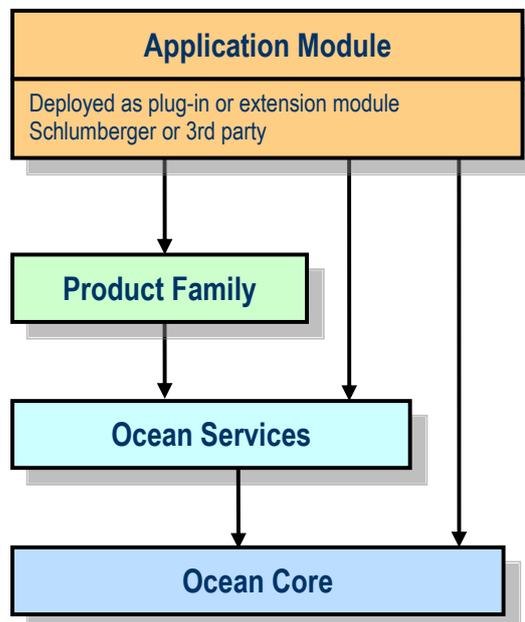
Figure 1: Ocean Architecture

The Ocean Core plays the role of the basic infrastructure. It manages Ocean modules and registers services, both the services pre-loaded by the product family as well as services that are added dynamically via the application programming interface (API). The Ocean Core manages the data sources provided by the product family or any external data source that could be defined by any module. It also performs event management and basic message logging.

The Ocean Services are a set of application independent utilities. They are modules that benefit from being standardized across product families. An example is the Coordinate Service – a utility for converting between projection and geodetic coordinate systems. The Ocean Services layer only depends on the Ocean Core or on other Ocean Services.

The product family is the host for Ocean applications and is the environment in which the Ocean module needs to run. The product family provides:

- the domain objects and their data source

- the graphical environment in which the applications will display their data

- a common look and feel for all application user interface components

Application modules connect to all software layers as well as to the .NET framework. Application modules can register their own services with the Ocean Core and benefit from services registered by other modules. All applications built on the Ocean framework are designed in a similar fashion, but they rely on a product family to build and run.

Access to the Petrel Data Domain

The application can access data in the following domains provided by Ocean for Petrel:

- Analysis (for data analysis)

- FaultModel (for fault modeling)

- Stratigraphy (for stratigraphy)

- Well (for Petrophysics and Geology applications)

- Prestack, Seismic, SeismicGeobody (for Geophysics)

- Shapes (for generic geometric objects)

- FrameworkModeling (for Structural modeling)

- StructuralGeology (for Structural geology)

- PillarGrid, UnstructuredPillarGrid (for Geomodeling)

- Simulation (for Reservoir Evaluation)

This is not an exhaustive list.
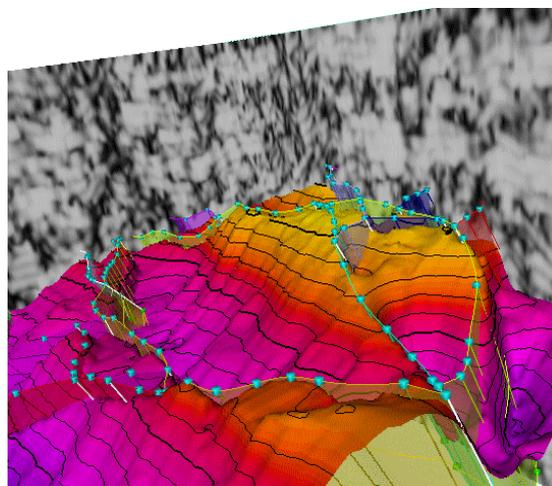
Figure 2: Surface with fault model and seismic data

Ocean for Petrel UI Infrastructure

Ocean provides the capability to extend Petrel's user interface functionality. The Ocean for

Petrel Application Programming Interface (API) supports:

- Windows:
    - Adding custom windows
- Renderers:
    - Adding renderers for domain objects (native and custom) in different windows
- Interactions:
    - Adding custom window modes to define interactions in different windows
    - Object picking in different windows for its manipulation
- Menus and toolbars:
    - Adding new menus to Petrel window or extending Petrel menus
    - Adding new toolbars with custom tools
    - Extending Petrel toolbars with custom tools
- Petrel project explorer:
    - Adding custom objects in Petrel tree in a particular hierarchy
    - Adding processes and workflows in the Petrel process diagram and workflow editor

This is not an exhaustive list.

The Ocean Plug-in and Module

An Ocean module is an extension to the product family. Plug-in programmers create modules that behave just like any standard part of Petrel. The modules are compiled into assemblies. Modules are combined in a plug-in that provides identity and support information. Plug-ins are installed by the Petrel Plug-in Manager found under the Help menu of Petrel. The Plug-in Manager reads a .pip (Plug-in Installation Package) file that is deployed for the plug-in and uses it to define the plug-ins, and their modules, loaded when Petrel is started.

The following figure shows a sample Ocean plug-in with modules.
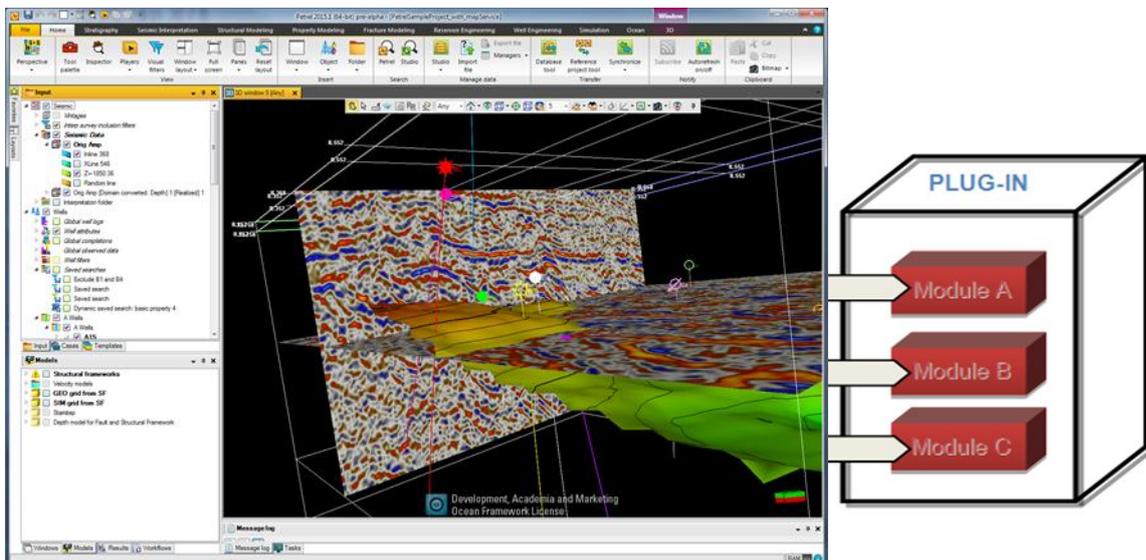
Figure 3: Ocean Modules

The Ocean module has a defined lifecycle with certain requirements and restrictions that allow the clean integration into the product family. Its lifecycle phases serve to license the module, initialize and integrate it into the product family, add presentation interfaces, and remove all these when the module is unloaded. The appearance and interaction is seamless and allows the product family to treat the module as native code.

An Ocean module uses the functionalities provided by the product family, the Ocean Services, the Ocean Core, and also the .NET architecture. It may also use third party application assemblies and other modules.

Ocean for Petrel modules are built with Visual Studio 2011 or Visual Studio 2012 for .NET, with the help of the Ocean for Petrel wizard. The wizard takes care of the interaction with the Core as well as some low-level access to functionality provided by the Petrel product family.

Plugin class

An Ocean plugin derives from the abstract **Plugin** class, defined in the **Slb.Ocean.Core** namespace. During Petrel startup, the Ocean Core will load plug-ins and their contained modules as defined in the PluginManagerSettings.xml file. The Plugin class contains information about the plug-in and its identity. The following is an example module:

```
using Slb.Ocean.Core;
using Slb.Ocean.Core;
public class MyPlugin : Plugin
{ ...
  public override string Name { get { return "Plugin Demo"; } }
  public override PluginIdentifier PluginId
  { get { return new PluginIdentifier("PluginDemo", new Version("1.0.0.0")); } }
  public override IEnumerable<ModuleReference> Modules
  {
    get { yield return new ModuleReference(typeof(DemoModule));
          yield return new ModuleReference(typeof(DemoModule2)); }
  }
  public override IEnumerable<PluginIdentifier> Dependencies
  { get { return new List<PluginIdentifier>(); } }
  public override string Description { get { return "Example plug-in"; } }
  public override string AppVersion { get { return "2014.0"; } }
  public override string Author { get { return "Joe Smith"; } }
  public override string ImageResourceName
  { get { return "MyPlugin.MyImage.bmp"; } } // used if Plugin is licensed
  public override string DeploymentFolder
  { get { return "OceanTraining"; } }
}
```

IModule Interface

An Ocean module implements the **IModule** interface. **IModule** is defined in the **Slb.Ocean.Core** namespace.

The **IModule** interface defines five methods of the module lifecycle phases and inherits from **IDisposable**. The phases and their methods are:

- Construction (default constructor)
- Initialization (**Initialize**)
- Integration (**Integrate**)
- Presentation integration (**IntegratePresentation**)
- Disintegration (**Disintegrate**)
- Disposal (**Dispose**)

During the product startup, the Ocean Core will load modules as defined in a configuration file for the product family. The Ocean Core will call the default constructor to instantiate each module. The constructor can be used to initialize any private fields, and acquire any resources the module needs.

The second phase of the module lifecycle is the execution of the **Initialize** method. The main purpose of **Initialize** is registration of the services provided by the module with the Ocean **ServiceLocator** class. When initialization is complete the services registered are available for consumption by all Ocean modules.

The **Integrate** method is the first point at which a module may consume services registered with Ocean. This is because all modules have been through the initialize phase where their services are registered with Ocean. The Ocean Core **ServiceLocator** class is used to look up services.

The **IntegratePresentation** method of **IModule** is the phase in the module lifecycle in which user interface components (menus, toolbars, context menus, windows etc.) are added to the product family.

When the product family begins to shut down, the **IModule Disintegrate** method is called on each module. **Disintegrate** has the responsibility of cleaning up any presentation elements installed by the module.

Dispose comes from the **IDisposable** pattern and must be implemented as part of the **IModule** implementation. The purpose of **Dispose** is to free any unmanaged resources and free any licenses acquired by the module.

The following is an example module:

```csharp
using Slb.Ocean.Core;
using Slb.Ocean.Coordinates;
public class MyModule : IModule
{
  // Perform initializations of private properties, resources.
  public Module ()
  { ...  }
  // Initialize services provided by the module
  public Initialize ()
  {
    // Create the service
    object FooBar = new FooBarService();
    // Add the service by its type
    CoreSystem.Services.AddService(typeof(FooBarService), FooBar);
    ...
  }
  // Integrate services provided by other modules
```

```csharp
public Integrate ()
{
  // Find the coordinate system service
  ICoordinateService coordService;
  coordService = CoreSystem.GetService<ICoordinateService>();
  ...
}
// Add to user interface of product family
public IntegratePresentation ()
{
  // Define the button to add
  PetrelButtonTool btn = new PetrelButtonTool("MyButton",
                                             PetrelImages.Cyan,
                                             ButtonClick);
 // Add the button to the Insert menu of the Petrel menubar
  WellKnownMenus.Tools.AddTool(btn);
  ...
}
// Remove components added to the user interface. Remove services
public Disintegrate ()
{
  // Remove the button added to the Insert menu
  WellKnownMenus.Tools.RemoveTool(btn);
  // Remove our service
  CoreSystem.Services.RemoveService(typeof(FooBarService));
  ...
}
// Dispose of any unmanaged resources
public Dispose ()
{ ...   }
```

WRITING YOUR
FIRST PLUG-IN

Ocean for Petrel allows application developers to extend the Petrel functionality with new custom processes and workflows. This chapter describes the procedure of creating a simple process.

Writing the Plug-in

In your first plug-in, you will write an Ocean plug-in with a module that adds a process to Petrel. The process will print the names of all Seismic cubes in the current project.

A process in Petrel is an interactive operation performed on data. The process is available from the Petrel process diagram. The operation could create new data or update existing data. For example, the operation of using a set of points to create a well path would be referred to as a process. In this example, the set of points is the input, and the well path is the output.

A workstep is a single processing step. It differs from a process in that it does not require manual interaction when executing. This is why it can take part in a workflow. Worksteps are available from the Petrel workflow editor; they also take input data and parameters, perform some processing, and produce output data.

A process can be created from a workstep, so you can write code one time that is available in both the process diagram and the workflow editor.

For your first plug-in, you will use the Ocean Wizard to create a process from a workstep.

There are three main steps for creating your first plug-in. Each step will be detailed in the sections that follow. The steps are:

- Run the Ocean for Petrel Wizard in Visual Studio to create the plug-in and module.

- Inspect the files created by the Wizard.

- Modify the code to add the processing logic.

For details on the installation and use of the Ocean Wizard, please refer to the appendix of this document.

For details on processes and worksteps, please see the Workflow chapter in the *Ocean Developer's Guide*.

Creating the Plugin, Module and Process with Visual Studio

To create the project, plugin, module, process, and workstep using Visual Studio:

Start Visual Studio.

Create a new project by selecting **File** > **New Project**. In the Project types area, under **Visual C#** project type, select **Ocean**. Then select the **Ocean Plug-in template**. Provide the name "ListSeismic" for the project. Click the **OK** button to start the Wizard. (See Figure 4.)
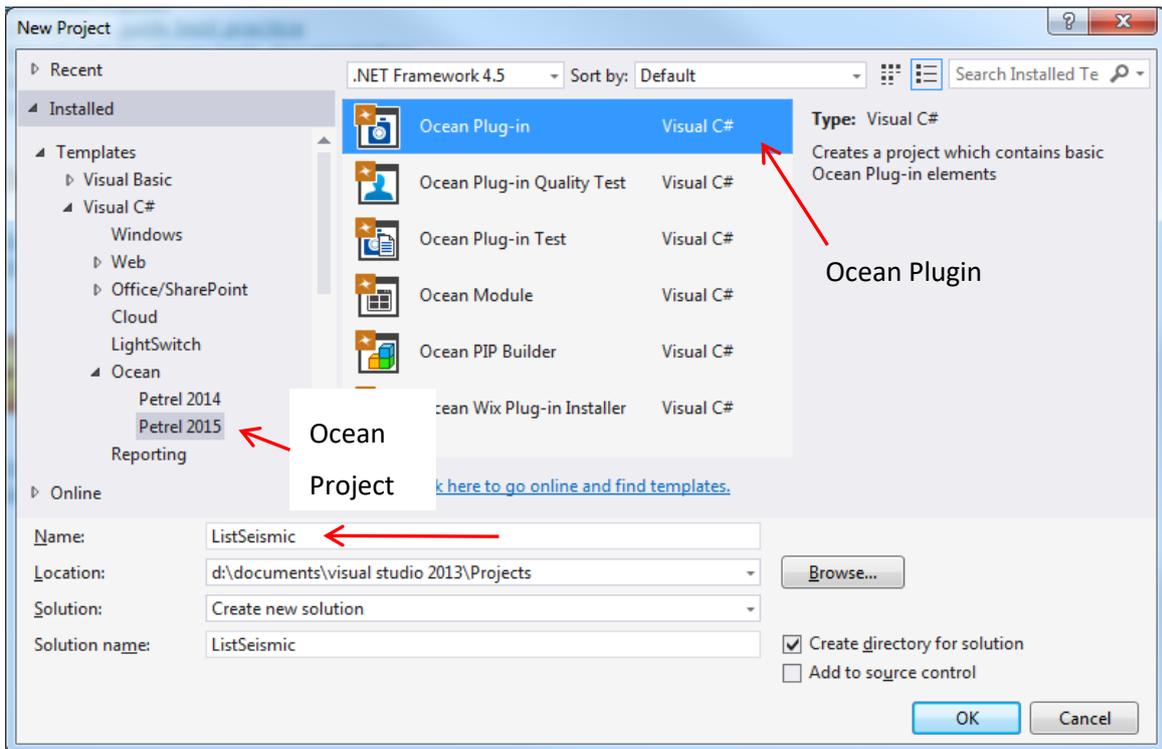
Figure 4: New Project Window

It is generally a good practice to use a descriptive plug-in name. Change the name of your plug-in to "ListSeismicPlugin". Change the "Author", "Contact", "Plugin URL", and "Description" fields as appropriate. Also, since we are adding a module to the plug-in turn on the "Create new module" and the "Register existing modules" checkboxes (See Figure 5.). Click Next in the dialog.
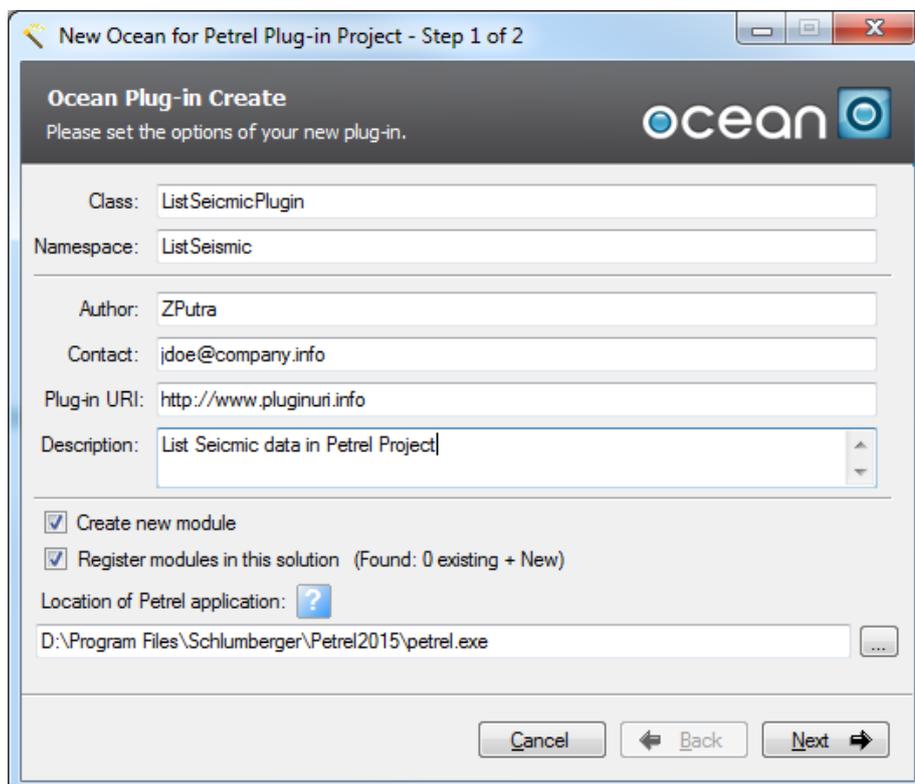
Figure 5: Plug-in Creation step 1 window

The wizard will create the name of the module from the namespace with ".Module" appended. Review the content (see Figure 6) and click Next in the dialog.

Figure 6: Plug-in Creation step 2 window

It is generally a good practice to use the name "Module" for the application module. Change the name of your module to "ListSeismicModule". Create a workstep in your module by checking the **New Workstep** box; give it the name "ListSeismicCubes". Click the **Next** button. (See Figure 7.)
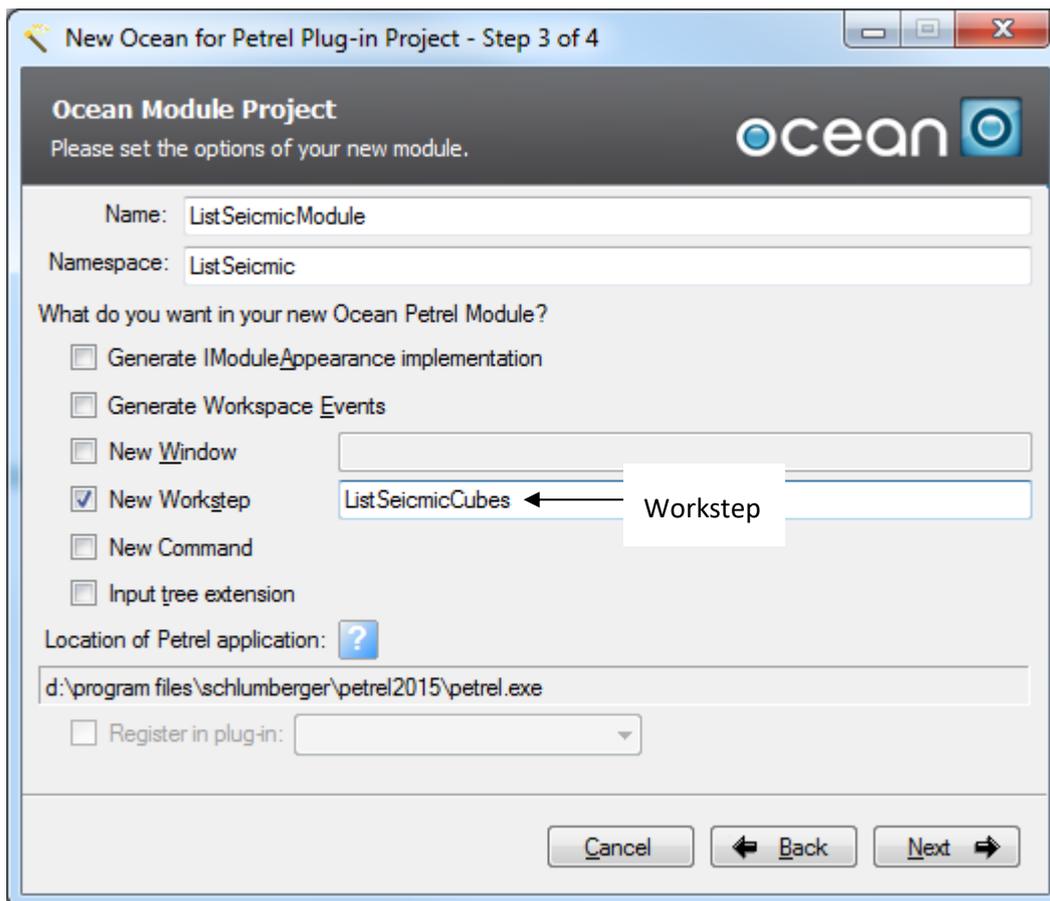
Figure 7: Ocean Module Project options

For your workstep, enter a **Short description** of "My First Ocean Process" and a **Long description** of "This process will print the names of all seismic cubes in the current project". Ensure that **Generate custom UI** is unchecked in order to use the default UI automatically generated by Ocean. This workstep should have its own process wrapper; ensure that **Generate process wrapper** is checked. Click the **Next** button. (See Figure 8.)
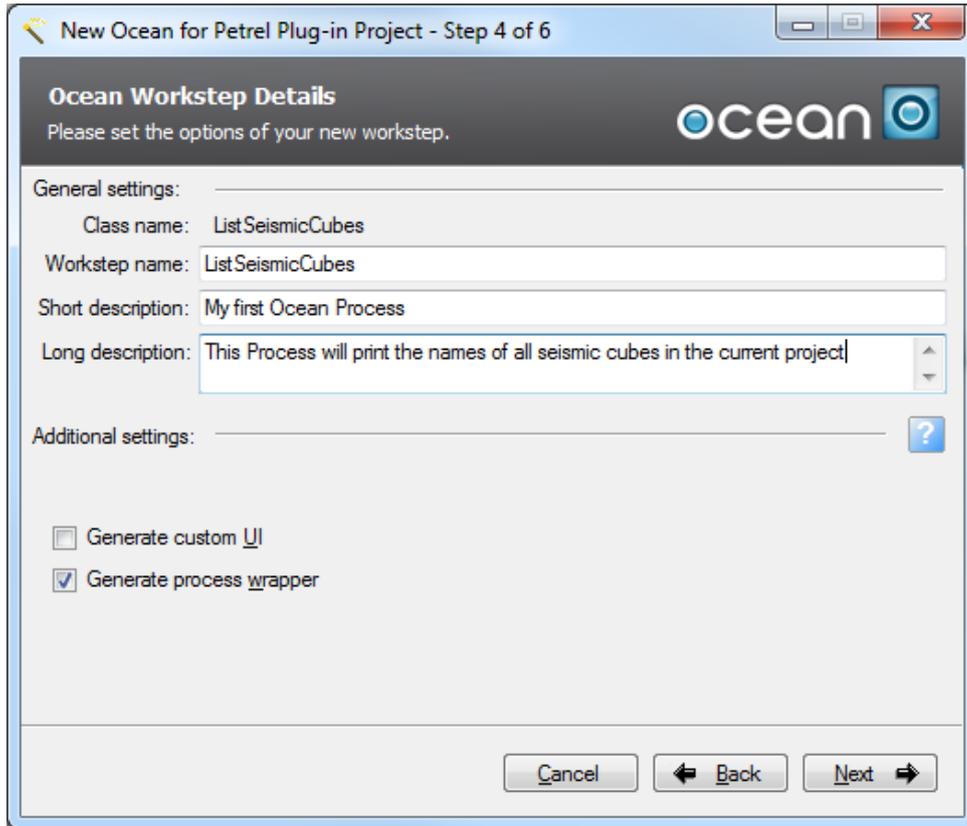
Figure 8: Ocean Workstep Details

Do not specify any workstep arguments; they are not needed. Click the **Next** button. (See Figure 9.)
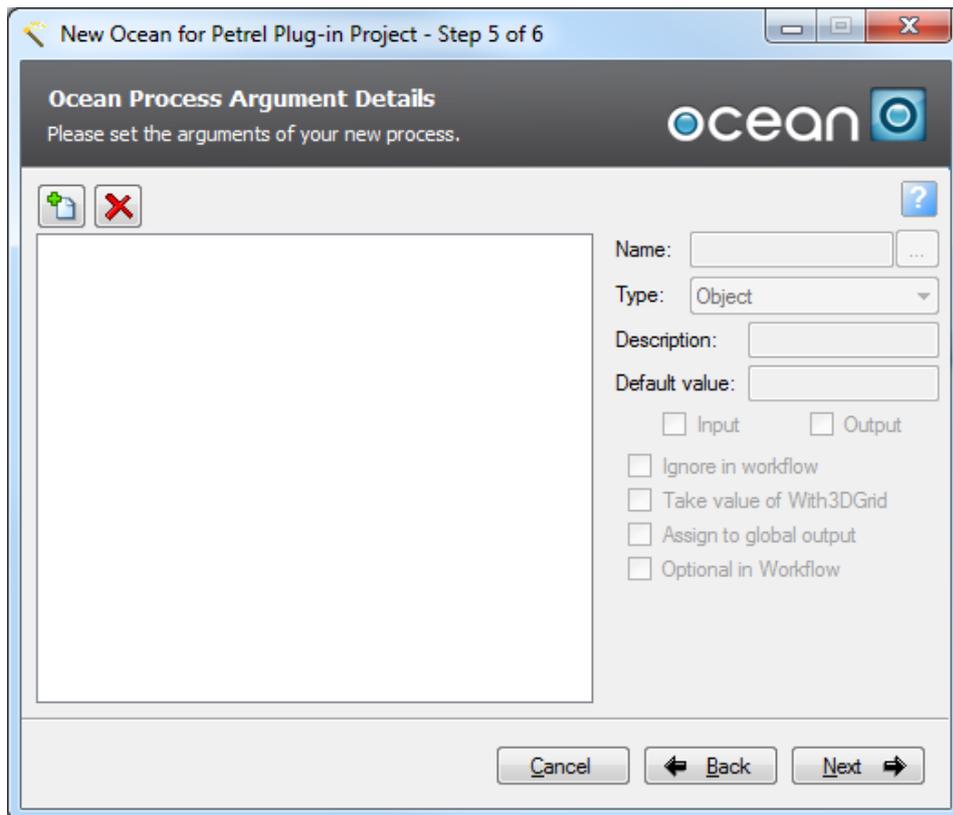
Figure 9: Ocean Workstep Argument Details

Review what the wizard will generate and click the Finish button. (See Figure 10.)
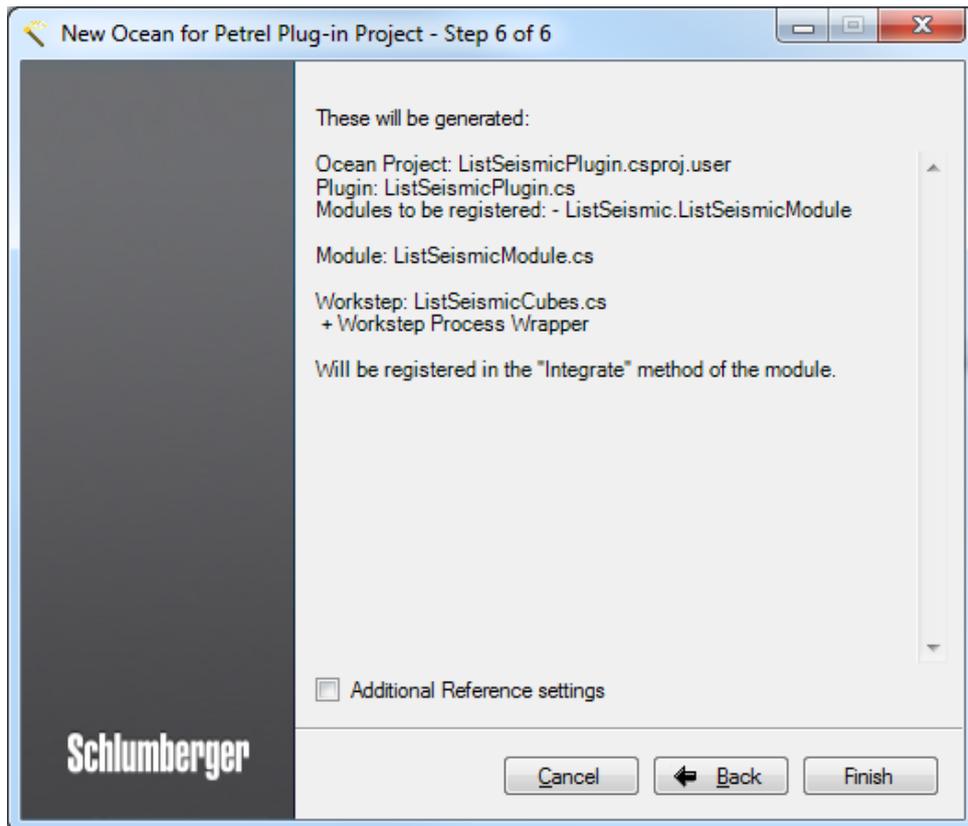
Figure 10: Ocean for Petrel Module Project settings

Inspecting the Files

The Ocean for Petrel Module Wizard will create a solution named "ListSeismic" with a project named "ListSeismic" in the Visual Studio Solution Explorer. The project will contain source files for the Plugin class that was created, the Module class that was created, as well as the ListSeismicCubes process that is added to the Process diagram. (See 11.)
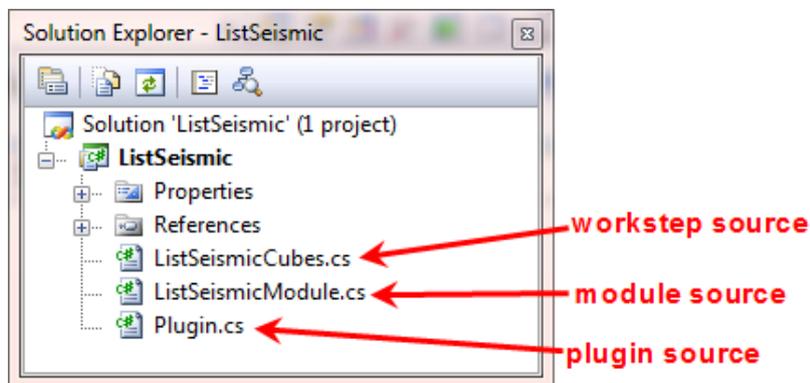
Figure 11: Example Project Source Files in Solution Explorer

The Wizard configures the ListSeismic project to make debugging simple. The assembly is output to the Petrel installation directory. The project command line property is set up to start Petrel. So debugging is simply a matter of building and starting the debugger.

The Wizard writes the basic code needed for the ListSeismicCubes plug-in. You will need to add code for your specialized algorithm in order to print the names of all Seismic Cubes in the current project.

Plugin

The **Plugin** class contains properties which provide identity to the plugin and is generated by the wizard from the values you entered and the Ocean version. Thes include **AppVersion**, **Author**, **Contact**, **Dependencies**, **Description**, **ImageResourceName**, **PluginUri**, **Modules**, **Name**, **PluginId**, and **Trust**.

Module

The **Module** class implements the **IModule** interface. The new process is added to the Process manager in the **Integrate** method automatically by the Wizard.

First the new ListSeismicCubes workstep is created, and then it is added to the Workflow editor. Finally, the process is created from the workstep using the **WorkstepProcessWrapper** convenience class. The new process is added to the process diagram using the **PetrelSystem.ProcessDiagram** API. Putting the pieces together, the Wizard generated the following **Integrate** method:

```
public void Integrate()
{
  ListSeismicCubes listseismiccubesInstance = new ListSeismicCubes();
  PetrelSystem.WorkflowEditor.Add(listseismiccubesInstance);
  PetrelSystem.ProcessDiagram.Add(new WorkstepProcessWrapper(
listseismiccubesInstance),"Plug-ins");
}
```

ListSeismicCubes

The **ListSeismicCubes** class is the workstep class that holds the processing algorithm. Previously, you saw how a process was created from it in the module.

ListSeismicCubes derives from **Workstep<>**, a generic class that includes the definition of the arguments. The wizard implements much of the class for you, including the argument creation and copy methods. Arguments creation is very basic. Arguments copy can be done with the **DescribedArgumentsHelper** static class.

The Wizard also implements two interfaces, **IAppearance** (for its appearance in the Process diagram) and **IDescriptionSource** (to display information in the Process UI). We do not need the setters for **Text** and **Image** properties, so you may delete them.

ListSeismicCubes also contains an **Arguments** class created by the Wizard, which in this case is empty since the algorithm has no arguments.

Writing the Algorithm Code

The Wizard wrote nearly all of the **ListSeismicCubes** class except for the custom algorithm code. It implemented the **IExecutorSource** interface and an instance of the **Executor** class. You will add the custom algorithm code to the **ExecuteSimple** method, which will be called when the workstep or process is executed.

```
public class ListSeismicCubes :  Workstep<ListSeismicCubes.Arguments>,
        IExecutorSource, IAppearance, IDescriptionSource
{
  #region IExecutorSource Members and Executor class
```

```
  public Slb.Ocean.Petrel.Workflow.Executor GetExecutor
      ( object argumentPackage, WorkflowRuntimeContext ctx )
  {
      return new Executor ( argumentPackage as Arguments, ctx );
  }
  public class Executor : Slb.Ocean.Petrel.Workflow.Executor
  {
    Arguments arguments;
    WorkflowRuntimeContext context;
    public Executor ( Arguments arguments, WorkflowRuntimeContext ctx )
    {
      this.arguments = arguments;
      this.context = ctx;
    }
    public override void ExecuteSimple( )
    {
      // TODO: Implement the workstep logic here.
    }
  #endregion
  ...
}
```

To write the algorithm code:

Access the APIs from the Seismic namespace by adding a reference in your project to the
Slb.Ocean.Data.dll, **Slb.Ocean.Petrel.dll** and to the **Slb.Ocean.Petrel.Seismic.dll**. You must also
add a reference to **Slb.Ocean.Petrel.Basics.dll**. These assemblies are available in the Public
folder of the Petrel installation directory. When adding reference to Ocean for Petrel public
assemblies, always set copy local reference property to false. This is to ensure successful
deployment of your plug-in to Petrel.

Add the following **using** statements for the required namespaces at the top with the other **using**
statements.

```
using System.Collections.Generic;
using Slb.Ocean.Petrel.DomainObject;
using Slb.Ocean.Petrel.DomainObject.Seismic;
```

Code the **ExecuteSimple** method. The work for the process is completed as follows:

Read the current project using the **PetrelProject.PrimaryProject** API. Use the static class
SeismicRoot to find the root of all domain objects in the seismic domain. It exposes a property
SeismicProject, which provides navigation to all seismic collections and interpretation
collections in the project. Parse through all the seismic collections and print the names of all
seismic cubes within them using the **InfoOutputWindow** method of the **PetrelLogger** class. Build
a dynamic list of surveys (**SeismicCollection**) to avoid recursive calls while providing a breadth
traversal of the survey tree. The **PetrelLogger** static convenience class provides static methods
to show a textual display of information to users. The **InfoOutputWindow** is used to issue an
information message that is intended for all end users to see, but without being highly
interruptive.

The following shows the complete Executor class:

```
using Slb.Ocean.Petrel.DomainObject;
public class Executor : Slb.Ocean.Petrel.Workflow.Executor
{
  Arguments arguments;
```

```
WorkflowRuntimeContext context;
public Executor ( Arguments arguments, WorkflowRuntimeContext context )
{
  this.arguments = arguments;
  this.context = context;
}
public override void ExecuteSimple( )
{
  // Get current primary project
  Project proj =  PetrelProject.PrimaryProject;
  // Get the root of all domain objects in the seismic domain
  SeismicRoot root = SeismicRoot.Get(proj);
  SeismicProject sProj = root.SeismicProject;

  // Find all seismic collections in the current project
  IEnumerable<SeismicCollection> col = sProj.SeismicCollections;
  List<SeismicCollection> listCol;
  listCol = new List<SeismicCollection>(col);
  for(int idx = 0; idx < listCol.Count; idx++)
  {
    SeismicCollection curr = listCol[idx];
    foreach (SeismicCollection sCol in curr.SeismicCollections)
    {
      listCol.Add(sCol);
    }
    // Find all Seismic cubes in the current collection
    foreach (SeismicCube cube in curr.SeismicCubes)
    {
      PetrelLogger.InfoOutputWindow(curr.Name +
                    " contains Seismic Cube " + cube.Name);
    }
  }
  return;
}
}
```

Running the Plug-in

You have just completed the modification of the **ExecuteSimple** method. In this section, you will finish building the solution and run Petrel with your plug-in.

Build your solution in Visual Studio. This will create the assembly for the plug-in. It will also create a plugin.xml file and merge its contents into the PluginManagerSettings.xml file. The PluginManagerSettings.xml file exists under your %AppData% directory for your account on your PC. Typically this is C:\Users\<username>\AppData\Roaming\Schlumberger\Petrel\2014\. This file is used when Petrel is started to define the plug-ins, and their associated modules, which will be loaded by the Plugin Manager.

Start Petrel.

In Petrel, open the demo project deployed with the Ocean SDK. Then go to the Process diagram window and find **ListSeismicCubes** (12).
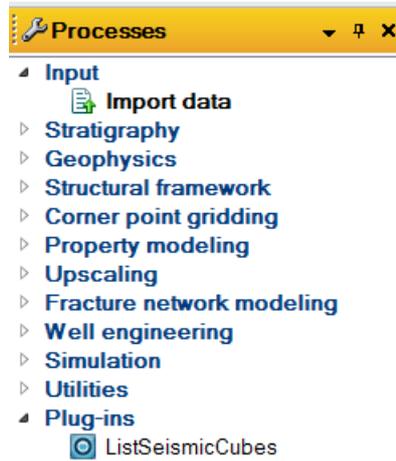
Figure 12: ListSeismicCubes Selected in Process Diagram

Double click **ListSeismicCubes** to start the process and display the **ListSeismicCubes** dialog. The short and long descriptions are shown in the top section of the dialog. There are no arguments for the process, so the lower section is empty, as shown in the following figure.
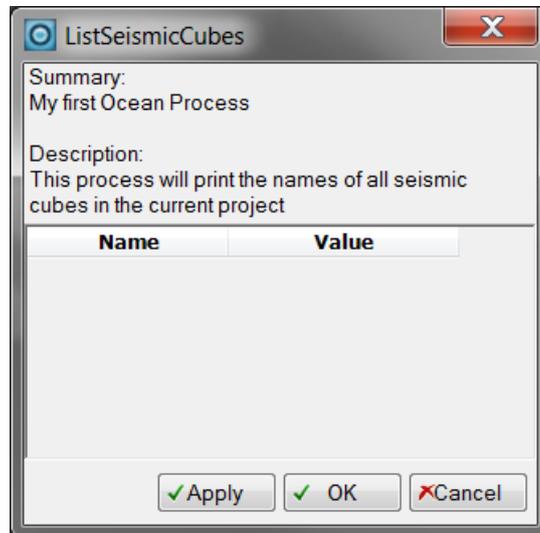
Figure 13: ListSeismicCubes Dialog

Click **Apply** to run the process. The **ListSeismicCubes** process will show the seismic cubes in the Petrel message log (Figure 14).
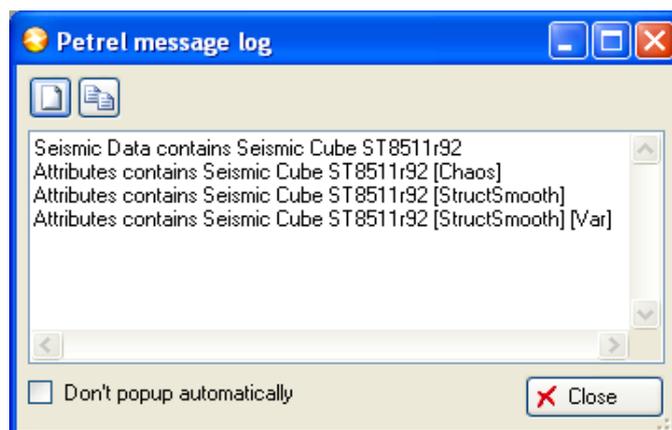
Figure 14: ListSeismicCubes Output Messages

Exit Petrel.

You have now written, built, and run your first Ocean plug-in. You can find further details about the Ocean Wizard and the functionality it offers in the Appendix of this document.

Using the Online Help

As you explore the Ocean APIs in the upcoming sections and in your development, you will want to look at the online documentation. The Ocean SDK includes online help exposing all the namespaces, classes, and interfaces implemented in the API.

Opening the Online Help

Ocean help should be integrated into Visual Studio Help and into F1 help when the Ocean SDK is installed. To select the online help, open Visual Studio, and select **View Help** from the **Help** menu as shown:
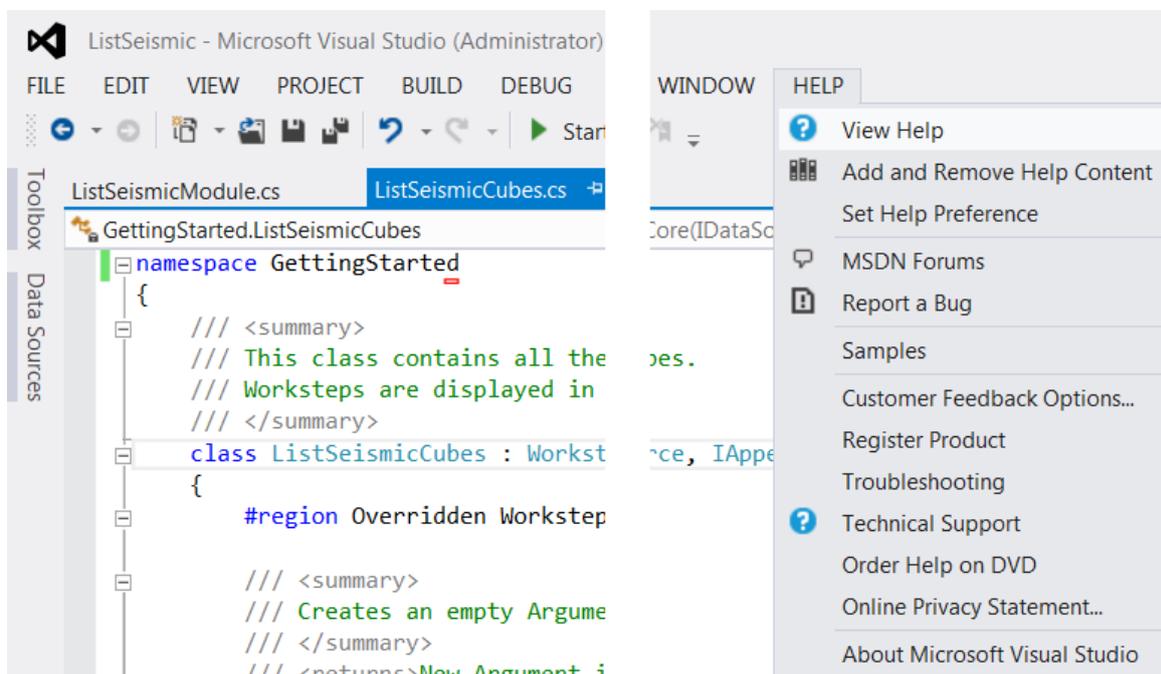
Hit F1 on a class, interface, property, method or other part of the Ocean API while you are in the Visual Studio editor window to bring up the corresponding help in a browser window, like you can for Microsoft APIs.

You can open the Ocean.chm file directly from the Documentation folder of the Ocean SDK installation directory or from the Ocean Start Page in Visual Studio.

Using IntelliSense

In Visual Studio, the Ocean library references (files with ".dll" extensions) allow the environment to select class members and methods from a list that is dynamically generated from the types of the variables in the code. This feature helps you select the correct class syntax without having to explicitly search for information in the Ocean help.

Accessing Class Definitions

While editing code, you can go directly to the class definition of the current variable declaration. To do this, right-click on the class name, and Visual Studio opens the class reference menu as shown in the following figure.
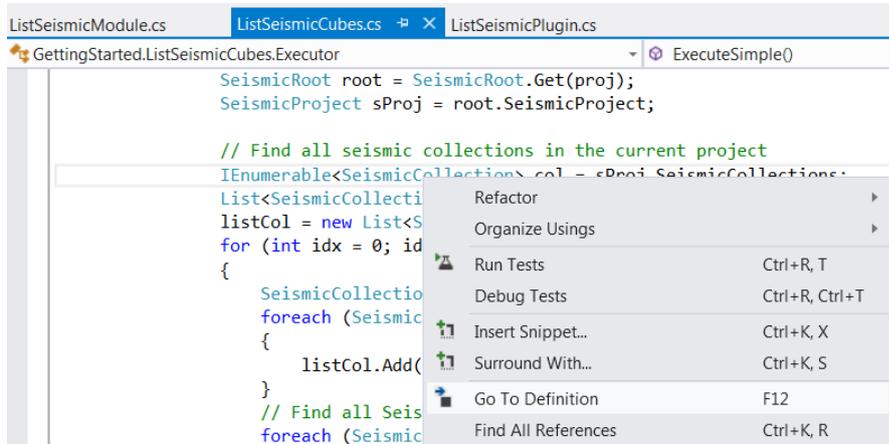
Select Go To Definition. The following editor opens in your Visual Studio environment.
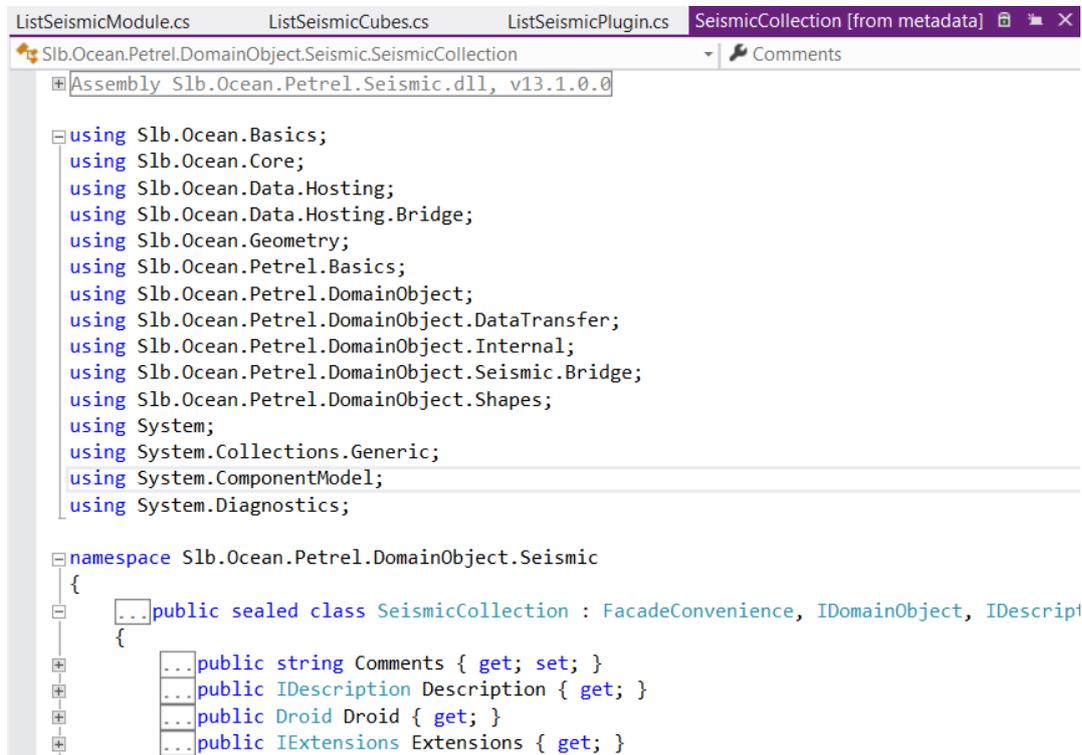
UNDERSTANDING THE PETREL DATA DOMAIN

Exposing Petrel's Data Model

Ocean for Petrel accesses data stored in Petrel Projects. The types of data that Ocean may access increase with each release as Petrel's data model expands and as more of the data is exposed through Ocean for Petrel. The Ocean for Petrel API documentation that accompanies the release contains the definitions of all exposed data types, their properties, and methods for access.

This section will help you understand the Petrel data domain as it relates to developing applications. Sample code is included to assist you in learning to use the API.

However, this document is only intended to help you get started using the Ocean for Petrel API, and not every feature or every domain is covered here. For a complete discussion of the Ocean for Petrel features, functionality and usage, refer to the *Ocean Developer's Guide*.

Entities and Properties

A project built by Petrel contains data elements that represent earth entities (a well, a surface, a reservoir). These entities are characterized by properties (such as logs along a well bore, height fields across a surface, or porosity distributions in a reservoir). Both types of data are exposed in Ocean by classes, called domain objects.

The entity instance carries the geometry of the object it represents. The property object only carries property values.

Templates

Each property object in Ocean has a template associated with it so that the application working with it has a way to determine how to display the property values. The template defines the units, scale, and color table used by the object. The templates are found in the **PetrelProject.WellKnownTemplates**.

User View of the Petrel Data Model

The Petrel Project organizes the user's view of the project in different trees:

- Input
- Models
- Results
- Cases
- Templates

- Processes

All the trees are accessible via the API. Data is organized in pseudo-folders, seen in the API as object collections. Each view is described in the following sections.

Input Data
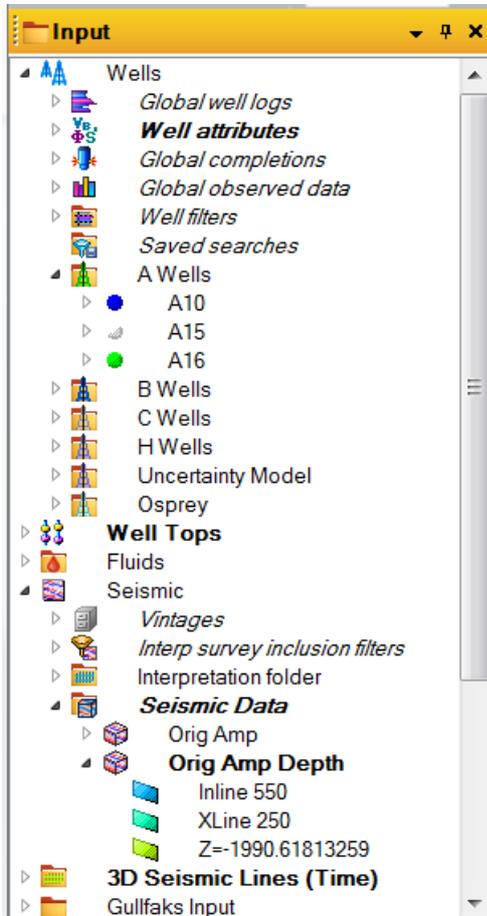
Figure 15: Input Data Tree

The Input Data function gathers all data that is needed to build a static model. This includes all data acquired and processed (well logs, geological markers, seismic data sets, seismic attributes) and all data that is interpreted from the field measurements (petrophysical properties, seismic time interpretations, velocity functions).

Most domain objects visible in the Input data tree are accessible in Ocean from the following specialized collections:

BoreholeCollection – This collection class contains boreholes, logs, and completions, as well as nested BoreholeCollections. See the Well data model for details.

MarkerCollection – This is the geology data model. These collections contain markers, horizons, zones, and fluid contacts.

SeismicCollection – This collection is for 3D and 2D seismic collections.

InterpretationCollection – This collection is for all seismic interpretation.

Collection – Other folders are accessible as generic collections. From these collections, the API can access point sets, polyline sets, and surfaces.

Models
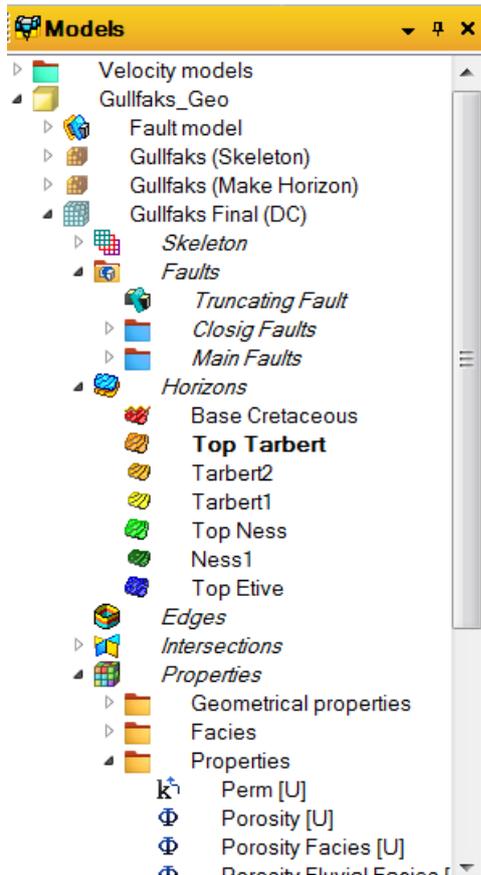
Figure 5: Models Data Tree

The Models data tree includes all elements of the pillar grid static model and the property objects that represent property distributions throughout the pillar grid cell array.

Domain objects visible in the Models data tree are accessible from the following collections:

ModelCollection. This top level container in the Models tree enumerates all the Models (pillar grids) in the tree. Model collections are not nested.

Grid.PropertyCollection. Specific to each model, the collection lists all property and fault property instances in that model.

Fault.Properties. This collection lists all fault properties, per fault, in a pillar grid model.

Results
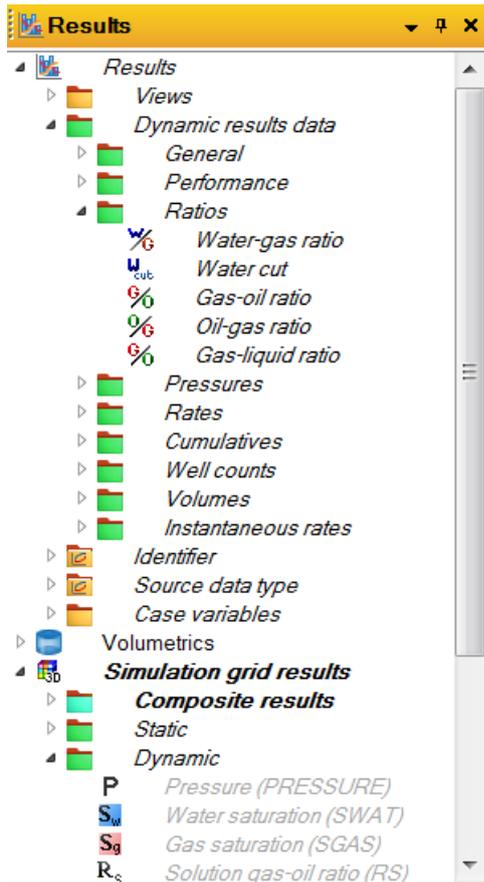
Figure 6: Results Tree

Results are simulation results imported back in Petrel to compare the static models with their possible evolution through time. They are available as **CaseResults** from the simulation **Case**. You can then get all **SummaryTimeSeries** and **GridPropertyTimeSeries** or the ones corresponding to a given **SummaryResult** or **GridResult**.

Cases
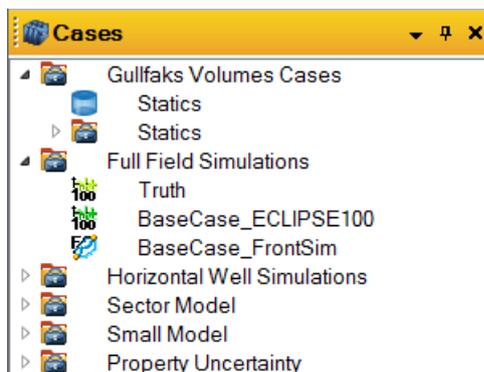
Figure 7: Cases Tree

The Cases tree exposes simulation cases based on different hypotheses but also represents various interpretation versions driving static model property variations. You can access **Case**, **AnalysisCase**, and **CalculationCase** instances from **SimulationRoot**. There is currently no hierarchy in the Ocean API.

Templates
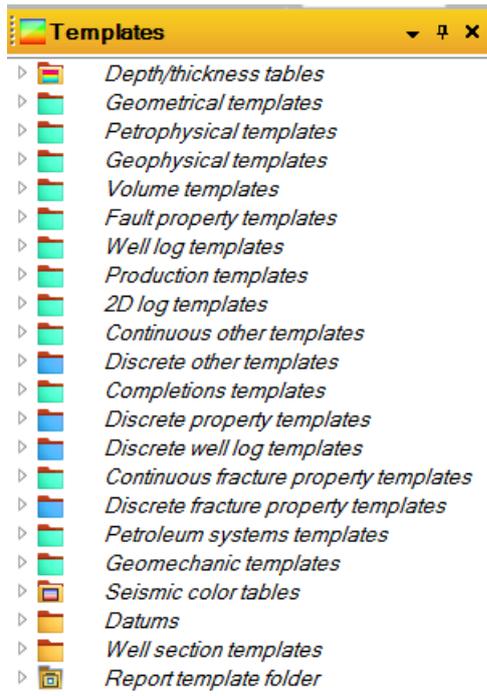
Figure 8: Templates Tree

The Templates tree exposes templates that are used by the Petrel user to define the data type for the domain objects in the project. The templates are not domain specific. This means that the Porosity template may be used to represent data for a Porosity well log as well as a Porosity property of a pillar grid.

Processes
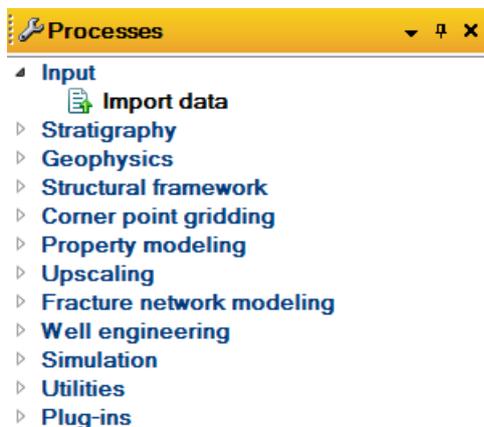
Figure 9: Processes Tree

The Processes tree exposes the Petrel processes used by the Petrel user to perform various actions on data. The processes are interactive in nature in that activating one causes the process toolbar on the right side of the Petrel UI to update to indicate the tools that are available to the process.

Data Access

The Ocean for Petrel API offers the following data access types:

Read: The API reads existing data, browsing through data folders, enumerating collection contents and following the parent-child relationships.

Update: The API updates existing objects, modifying entity characteristics (such as its geometry) and modifying property values (like correcting log values in chosen points).

Create: The API creates new instances of Petrel object classes, augmenting the project with new data the same way Petrel applications do.

Delete: The API deletes objects that are no longer needed in the project.

Common Exposed Data Types

Details are given in the compiled html manual part of the Ocean SDK release, **Ocean.chm** in the **Documentation** subdirectory.

Seismic

Ocean classes for Seismic are found in namespace **Slb.Ocean.Petrel.DomainObject.Seismic.** You can access virtual cubes and virtual 2D data via class **SeismicAttribute**.

Well

Ocean classes for well data are found in namespace **Slb.Ocean.Petrel.DomainObject.Well**.

Different types of Logs are available: property valued well logs, facies well logs, point logs, multitrace well logs, bitmap well logs, comment well logs, and borehole seismograms. Also available in the Well namespace are completions and observed data.

Geology

Ocean classes for well marker data, sometimes referred to as well tops, are found in namespace **Slb.Ocean.Petrel.DomainObject.Well**.

You can create stratigraphy columns across a number of wells. Fault and interface-type surfaces are just named references.

Shapes

Ocean classes for objects defining various shapes are found in the namespace **Slb.Ocean.Petrel.DomainObject.Shapes**.

You can add properties to point sets, surfaces, and polyline sets. Surfaces can be irregular or gridded (regular height field), though only regular height field surfaces can be created. Both can be assigned property values.

Pillar Grid

Ocean classes for pillar grids and their related objects are found in the namespace **Slb.Ocean.Petrel.DomainObject.PillarGrid**.

Pillar grid property retains some knowledge of statistical treatment completed at creation time, including which cells have been upscaled as well as statistical distributions of facies values.

Simulation

Ocean classes for simulation cases and simulation results are found in namespace **Slb.Ocean.Petrel.DomainObject.Simulation**.

A simulation case has a number of case runs, each providing time series (results vs time) data for combinations of result categories (fields, wells) and result properties (oil production, water flow rate, etc.).

Streamlines can be added to various stream line sets in a simulation case. Property values can also be read and appended and new property types added.

Read Access

Browsing Collections

There is no query service implemented on the Petrel Project as it is not implemented in a relational database. Instead, data browsing is done by following the non-taxonomic hierarchy. At the top level of each domain is a root object which gives you access to all of the data in that domain. It is possible to browse through the whole data domain of a project by starting from the different root objects.

Root objects offer access to collections, which may be nested. From each collection, the API can access the entities and the properties that are relevant to that collection type.

Seismic Data
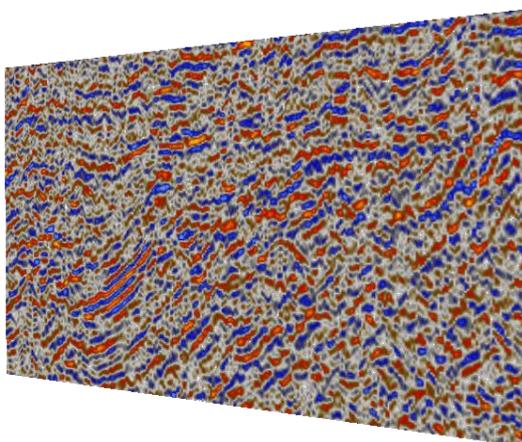
Seismic data is rooted in the project's **SeismicProject** instance (which is a singleton) once it has been created. It is accessed, if it exists, from the **SeismicRoot** of the project. It only exists if seismic data are included in the project.

Below the **SeismicProject** are two separate hierarchies in the Seismic domain, one for seismic

data with nested **SeismicCollection** objects, the other for interpretation picks with a nested hierarchy of **InterpretationCollection** instances.

```
Project proj = PetrelProject.PrimaryProject;
SeismicRoot sr = SeismicRoot.Get(proj);
SeismicProject sp = sr.GetOrCreateSeismicProject();
// navigate the Seismic data hierarchy
foreach (SeismicCollection scol in sp.SeismicCollections)
{ ... }
// navigate the Seismic interpretation hierarchy
foreach(InterpretationCollection icol in sp.InterpretationCollections)
{ ... }
```

Seismic data in a **SeismicCollection** include either seismic cubes or seismic 2D surveys (collections of lines).

Interpretation data include **HorizonInterpretation** or **FaultInterpretation** instances each including 3D and 2D time picks that can be accessed separately.

Well and Geology
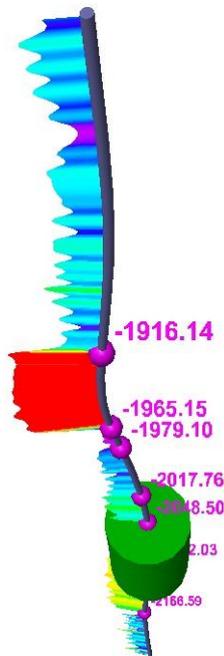
Wells belong to a nested hierarchy of **BoreholeCollection** objects that contain boreholes, trajectories, and logs. Markers are stored under stratigraphic columns or under horizon, zone, and marker collections. You can find lists of such collections at the top level, implemented in the API by the **MarkerCollection** class.

Your application can access all well and geology objects at the root of the project from the **WellRoot** class. **WellRoot** is unique for the project and retrieved from a static method in the **WellRoot** class as shown in the following.

```
Project proj = PetrelProject.PrimaryProject;
WellRoot wr = WellRoot.Get(proj);
// Get top level Borehole Collection
BoreholeCollection bhcol = wr.BoreholeCollections;
...
```

Once the **WellRoot** instance is retrieved by the system, it gives access to borehole and marker

collections.

```
public sealed class WellRoot : ...
{  ...
    public BoreholeCollection BoreholeCollection { get; }
    public int WellLogVersionCount { get; }
    public IEnumerable<PropertyVersion> WellLogVersions { get; }
    public int MarkerCollectionCount { get; }
    public IEnumerable<MarkerCollection> MarkerCollections { get; }
}
```

Boreholes are in nested collections, rooted at the top level collection retrieved from **WellRoot**.

Marker collections are not nested. Lists of collections, each corresponding to a different stratigraphic classification, are at the top level.

Pillar Grid Model
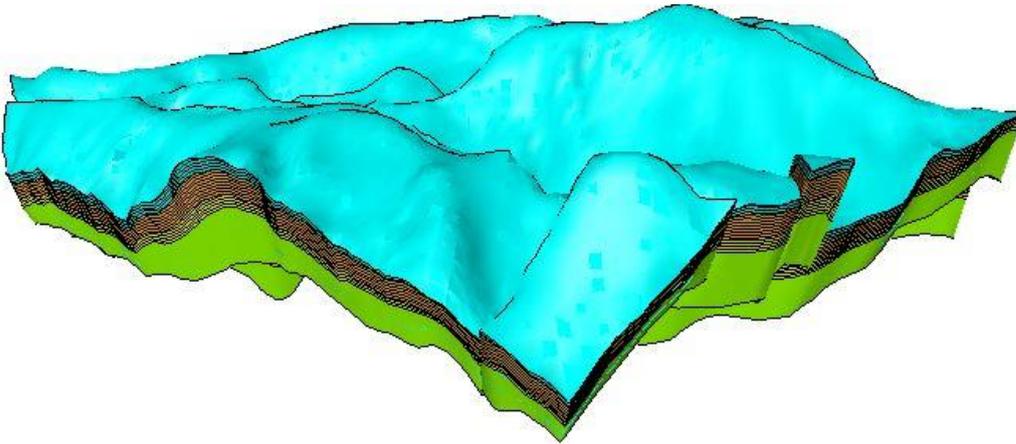
Pillar grids are in a hierarchy that contains the property hierarchies. Hierarchy retrieval starts from the **PillarGridRoot** object.

```
PillarGridRoot grid = PillarGridRoot.Get(proj);
IEnumerable<ModelCollection> mcolls;
mcolls = ModelCollection.GetRootModelCollections(proj);
foreach (ModelCollection mcol in mcolls)
{
  foreach (Grid g in gr.GetGrids(mcol)
  { ... }
}
```

Each **Grid** object interfaces a full static model with intersecting horizons and pillar-based faults.

```
Grid g = ...
foreach (Fault f in g.Faults)
{ ... }
foreach (Horizon hor in g.Horizons)
{ ... }
```

The model is populated by property values that are accessed in a nested hierarchy of **PropertyCollection** folders.

```
Grid g = ...
PropertyCollection pcol = g.PropertyCollection;
foreach (Property prop in pcol.Properties)
{
```

```
Template template = prop.Template;
// recurse on subcollections
foreach (PropertyCollection subcol in pcol.PropertyCollections)
{ ... }
}
```

Simulation and Data Analysis

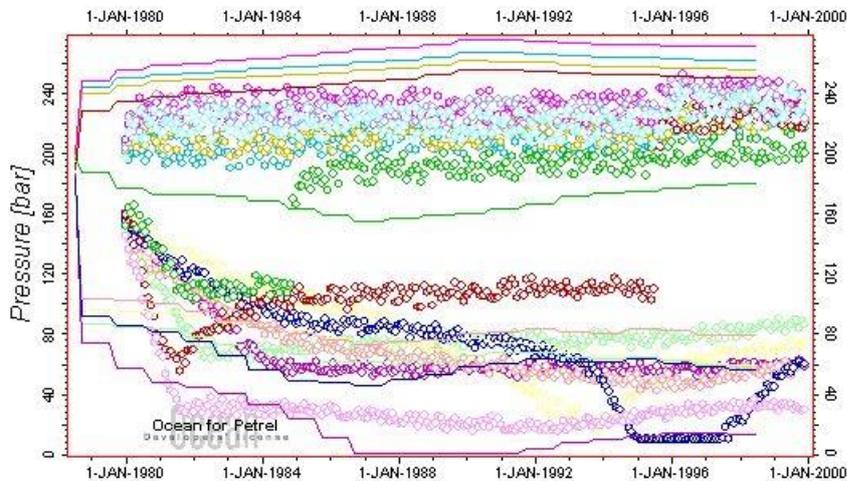Simulation cases, result properties, and result hierarchies belong to separate hierarchies. The time series are defined by combinations of these.

Simulation case runs are retrieved from **SimulationRoot**. The simulation case gives access to simulation results and streamlines. For time series, the result property and category must be supplied by the application.

```
SimulationRoot root = ...
foreach (Case case in root.Cases)
{
  StreamlineSet sls = case.StreamlineSet;
  if (sls != StreamlineSet.NullObject)
    foreach (StreamlineSubset sub in sls.StreamlineSubsets)
      foreach (Streamline sl in sub.Streamlines)
      { ... }
}
```

Updating Data

Your application can update data by modifying the object that is seen through the API. Updating an entity or a property object alters the project. After updating the project, you must decide whether to save the project changes or not. From the API standpoint, however, the project has been modified as soon as an object has been edited.

You must make any such updates inside a transaction and any modified object must be locked beforehand inside that transaction.

Using Transactions

A transaction represents a group of edits made to data (domain objects). Transactions are required for operations on native Petrel domain objects that will change the data: create,

update, and delete.

Transactions use the **Slb.Ocean.Core.ITransaction** interface. To use a transaction:

- Create a transaction.
- Lock domain objects that are going to be changed.
- Modify the domain objects.
- Commit the transaction.
- Dispose of the created transaction.

In Petrel, transactions are used to batch event notifications. Petrel does not use a data base, so changes to domain objects made within a transaction happen immediately.

Modifying Domain Objects

You can modify a domain object in the transaction after it has been locked. The online Ocean documentation will indicate which object property can be modified.

The nature of the class members is also explicitly shown in the class definitions that are accessible from the Visual Studio environment.

```
namespace Slb.Ocean.Petrel.DomainObject.Well
{
  public sealed class WellLog : ...
  {
    public Borehole Borehole { get; }
    public string Comments { get; set; }
    public int SampleCount { get; }
    public IEnumerable<WellLogSample> Samples { get; set; }
    ...
  }
}
```

The array of log values, **Samples**, is read-write but **SampleCount** is read-only (dynamically evaluated from the array size).

Accessing Domain Object Relationships

You can access object-to-object relationships via the API. For these relationships to be modifiable, the corresponding class property must allow write access, which is, as a general rule, not the case as parts-of relationships impose constraints between parent and child. For instance a pillar grid **Property** cannot be separated from the **Grid** object that defines its geometry. Even when you modify the containment hierarchy interactively, the API often makes the corresponding relationship immutable.

However, there are cases where it makes sense to allow write access to parts-of relations. This is the case when object creation does not require such a relationship to be established. For instance, it is possible to update the reference from a marker to a given surface (fault or horizon).

The example below clears the stratigraphic references of a set of markers in a given borehole.
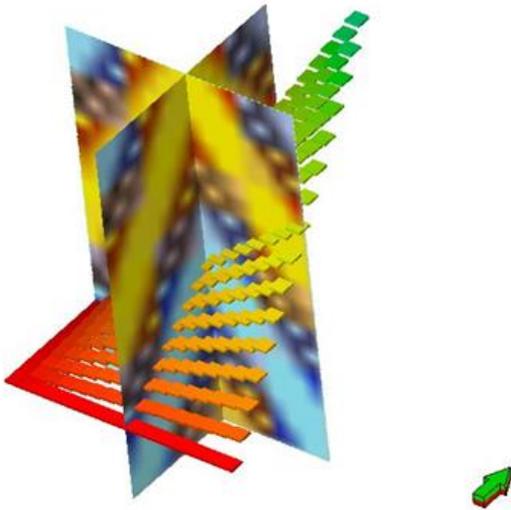
```
MarkerCollection mcol = ...
Borehole bh = ...
Horizon hor = Horizon.NullObject;
```

```
foreach (Marker m in mcol.GetMarkers(bh))
{ m.Surface = hor; }
```

Refer to the online help to determine whether a relationship is read-only.

Domain Object Creation

Creating New Instances

Most objects accessible via the API can be instantiated. However, the API never allows class instantiation in the classical manner. API classes are sealed, and some objects are even exposed simply as interfaces. To create a new class instance, the API has specialized Create methods.

Using Transactions

The plug-in code must create domain objects inside transactions, just like object updates. However, an object creation modifies its parent entity (in the non-taxonomic hierarchy) or the collection of similar entities that will harbor it. That parent or collection has to be locked in the transaction.

You can modify the newly created object until the transaction commits. It is locked by default, only its parent had to be locked explicitly in the transaction.

Locating the Correct Create Method

The Create method is always located in the class of the containing object. **Borehole** contains **WellLog** collections where the **Create** method for logs can be found.

```
using (ITransaction t = DataManager.NewTransaction())
{
  Template template = ...
  WellRoot wr = WellRoot.Get(PetrelProject.PrimaryProject);
  LogVersionCollection rootLVCol = wr.LogVersionCollection;
  t.Lock(rootLVCol);
  WellLogVersion version = rootLVCol.CreateWellLogVersion("Gamma", template);
  Borehole bh = ...
  t.Lock(bh);
```

```
  WellLog wl = bh.Logs.CreateWellLog(version);
   ...
}
```

Creating New Collections

When it does not make sense to create an entity in a predefined collection, you must create a new collection. Collections are often nested, so adding a collection is adding a branch to a tree. Classes that model nested collections allow creation of sub-collections. For example a **PropertyCollection** can contain sub-collections and would carry a **Create** method for branching out at that point.

```
ITransaction t = ...
PropertyCollection pcol = ...
t.Lock(pcol);
string colName = "New Sub-Collection";
PropertyCollection subcol = pcol.CreatePropertyCollection(colName);
t.Commit();
t.Dispose();
```

New Top-Level Collections

For top-level objects (like collections) the Create methods are in static classes. For example, the top level collection for Seismic data is created in the **SeismicProject** instance.

There are few collections that can appear at the top level of Petrel data trees. Nested collections are rooted under a top level collection that is only created once. Un-nested collections that can be added at the root of Input or Models data trees include the following:

- MarkerCollection

- ModelCollection

- Collection

These collections are created directly from the Petrel primary **Project** or the **WellRoot** object.

```
ITransaction t = ...
Project proj = PetrelProject.PrimaryProject;
WellRoot root = WellRoot.Get(proj);
t.Lock(proj);
proj.CreateModelCollection("New Pillar Model");
t.Lock(root);
root.CreateMarkerCollection("New Stratigraphic Model");
t.Commit();
t.Dispose();
```

Nested Collections

Collections that are always grouped under a root collection are created using a specific container, **WellRoot** for boreholes and **SeismicProject** for seismic data and interpretation. These are used for the following nested collections:

- SeismicCollection

- InterpretationCollection

- BoreholeCollection

There is no specific object at the root of all borehole collections, instead there is a generic **BoreholeCollection** called "Wells" that you can only create once.

```
ITransaction t = ...
Project proj = PetrelProject.PrimaryProject;
SeismicRoot root = SeismicRoot.Get(proj);
SeismicProject sproj = root.SeismicProject;
WellRoot wroot = WellRoot.Get(proj);
t.Lock(sproj);
string colName = "New Collection";
SeismicCollection scol = sproj.CreateSeismicCollection(colName);
String iName = "New Interpretation");
InterpretationCollection icol;
icol = sproj.CreateInterpretationCollection(iName);
...
t.Lock(wroot);
BoreholeCollection bcol = wroot. GetOrCreateBoreholeCollection();
...
```

You can also create collections that are placed under an object container, for example, under the **PillarGrid**.

```
ITransaction t = ...
Grid g = ...
t.Lock(g);
PropertyCollection root = g.PropertyCollection;
String colName = "New Properties";
PropertyCollection sub = root.CreatePropertyCollection(colName);
...
```

Deleting Objects

Deletion of an existing instance is done by calling the Delete method on the object itself. Deletion is always done inside a transaction and the object to be deleted must be locked first.

Accessing Data: Examples

In the following example you will browse through wells in the project, retrieve a porosity log, browse through static models, and create a new **Property** for the latest modified model in the project.

The created property will be filled with values extracted from the well log in cells where actual log measurements exist.
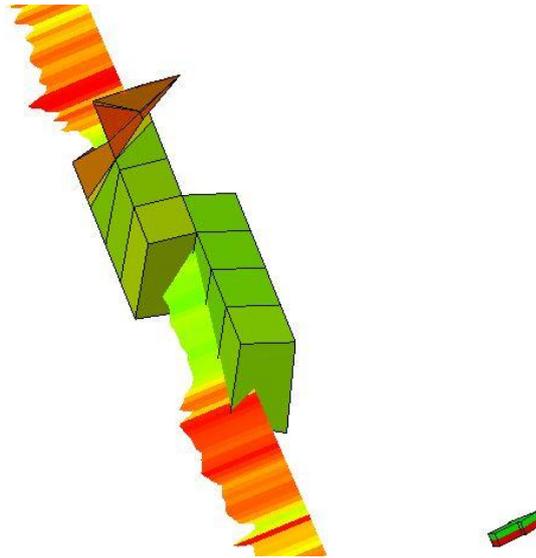
Browsing Well Logs

Use the following code to browse well logs to find a porosity measurement.

```
Project proj = PetrelProject.PrimaryProject;
WellRoot root = WellRoot.Get(proj);
BoreholeCollection wells = root.BoreholeCollection;


WellLog poro = WellLog.NullObject;
ITemplate template = PetrelProject.WellKnownTemplates.PetrophysicalGroup.Porosity;
IUnitMeasurement um = template.UnitMeasurement;
foreach (BoreholeCollection bhc in wells.BoreholeCollections)
{
  foreach (Borehole bh in bhc)
  {
    foreach (WellLog l in bh.Logs.WellLogs)
    {
      if (l.WellLogVersion.UnitMeasurement.Equals(um))
      {
        poro = l;
        break;
      }
    }
    if (!poro.IsGood) break;
  }
  if (!poro.IsGood) break;
}
if (poro == WellLog.NullObject) return;
PetrelLogger.InfoOutputWindow("Found log " + poro.Name);
```

Retrieving Models

To retrieve the latest model:

```
Grid latest = Grid.NullObject;
DateTime last = new DateTime(1980, 1, 1);
PillarGridRoot proot = PillarGridRoot.Get(proj);
```

```
foreach (Grid g in proot.Grids)
{
  if (g.LastModified.Time.CompareTo(last) >= 0)
  {
    latest = g;
    last = latest.LastModified.Time;
  }
}
if (latest == Grid.NullObject) return;
PetrelLogger.InfoOutputWindow("Found grid " + latest.Name);
```

Creating New Property Collections

To create a new property collection in the model:

```
ITransaction t = DataManager.NewTransaction();
PropertyCollection pcol = latest.PropertyCollection;
t.Lock(pcol);
string Name = "New Collection";
PropertyCollection newcol = pcol.CreatePropertyCollection(Name);
```

Creating the Pillar Grid Property

To create the pillar grid property:

```
t.Lock(newcol);
Property prop = newcol.CreateProperty(template);
```

Filling Values

To fill values:

```
IPillarGridIntersectionService ipgs;
ipgs = CoreSystem.GetService<IPillarGridIntersectionService>();

IPolyline3 traj = poro.Borehole.Trajectory.Polyline;
IEnumerable<SegmentCellIntersection> cells;
cells = ipgs.GetPillarGridPolylineIntersections(latest, traj);
foreach (SegmentCellIntersection cell in cells)
{
  Index3 ijk = cell.EnteringCell;
  Point3 xyz = cell.IntersectionPoint;
  double index = poro.Borehole.Transform(Domain.ELEVATION_DEPTH,
    xyz.Z, Domain.INDEX);
  WellLogSample[] sam = new WellLogSample[](poro.Samples);
  latest[ijk] = sam[(int)index];
}
t.Commit();
t.Dispose();
```

EXTENDING THE PETREL UI

Ocean allows application developers to customize and extend the Petrel application UI. In this chapter, you will explore a very simple customization: adding your own menu item to an existing Petrel menu.

However, this document is only intended to help you get started using the Ocean for Petrel API, and not every feature is covered here. For a complete discussion of the Ocean for Petrel features, functionality and usage, refer to the *Ocean Developer's Guide*.

Adding a New ribbon tab and command button

To add your own button to Petrel ribbon UI, chose **Ocean Command** from the **Add/New Item** option from the Visual Studio context menu, or if you checked the option to add a new command in Step 1 of the Wizard dialog, then you will see the following wizard screen:
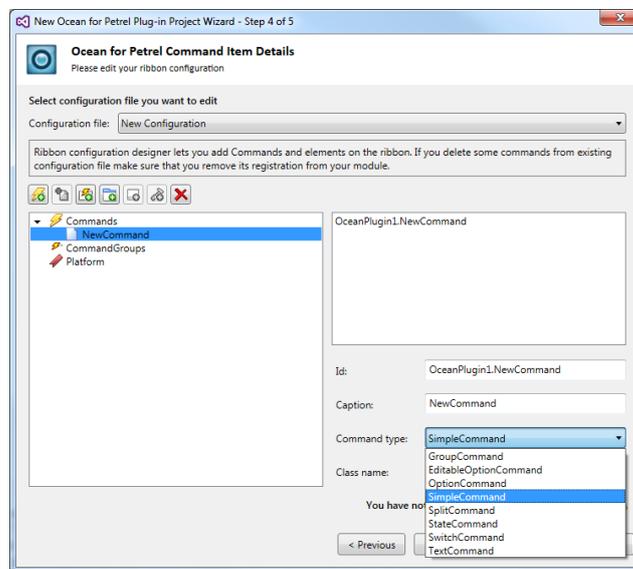

Figure 10: Preview of Solution Explorer with New Command resources

From this screen, you can add new command, create new ribbon or ribbon tab, or extend existing Petrel command group with commands. You must always select the parent item to which the new item will be added.

The types of the menu items that can be added include the following, this wizard will create basic container and register these command to make them visible in the ribbon. For detail guide and sample on how to implement your logic for these command refer to Ocean for Petrel CHM under Slb.Ocean.Petrel.Commands namespace:

- SimpleCommand: Simple command to perform a specific action. When added to Petrel UI it will be rendered as a basic button.

- StateCommand: Command that indicates one of two states (e.g. "on" or "off"). When added to the Petrel UI, this will be represented by a button with an on/off state.

- TextCommand: Command that defines an editable text option. When added to the Petrel UI, this will be represented by a Textbox.

- OptionCommand: Command that defines several selectable options. When added to the Petrel UI, this will be represented by a drop down list.

- EditableOptionCommand: Command that defines several selectable options with the possibility to type other texts as well. When added to the Petrel UI, this will be represented by a Dropdown list with an editable text area.

- GroupCommand: Button with the appearance of the GroupCommand. When clicked popup menu list of sub-commands is displayed.

- SwitchCommand: Button with the appearance of the GroupCommand. When clicked popup menu list of sub-commands is displayed.

- SplitCommand: Split button. The primary part displays and executes the selected sub-command while the secondary part display a popup menu list of sub-commands.

-

To create a new command, select Commands parent item in the tree and click "Add New Command". Command id, caption and type can be change as shown in below:
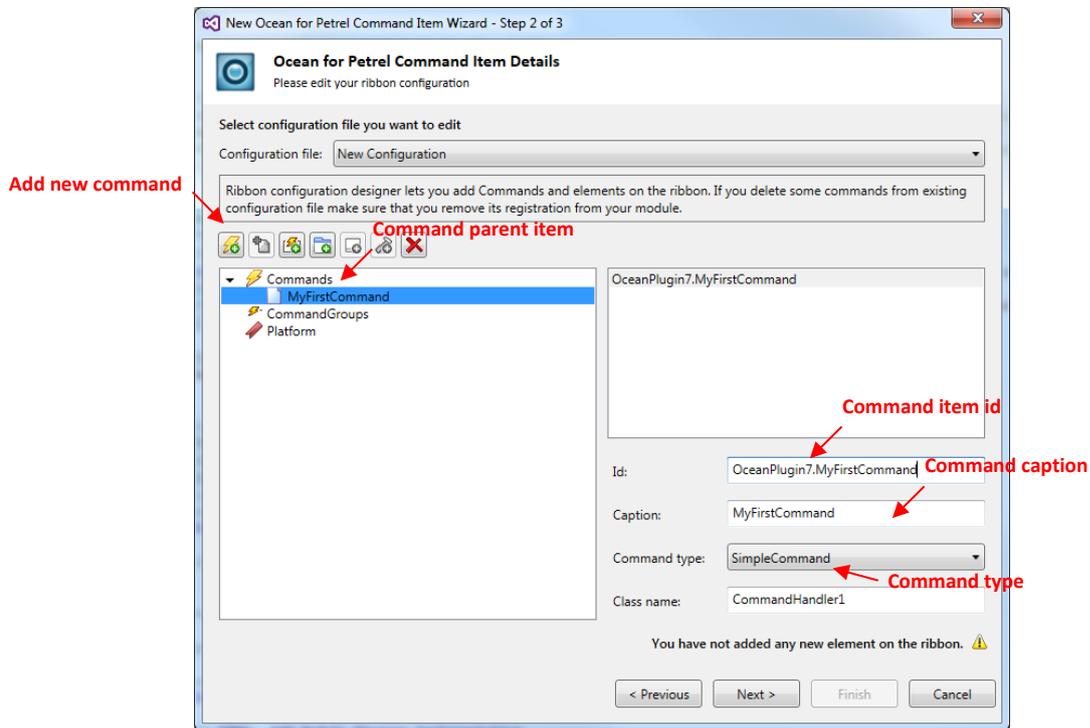

Figure 11: Wizard to add command

To include the new command you created above, select ribbon parent item and click "Add new ribbon to tab". Then click "Add new ribbon tabgroup". Finally to add your command to tabgroup, click "Add command to tabgroup" and your ribbon tree will apear as below:
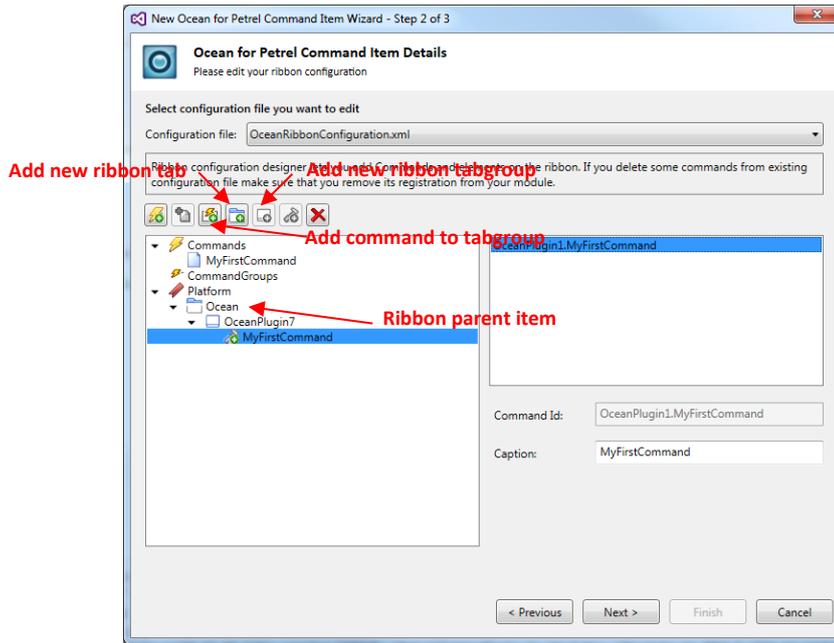
Figure 12: Wizard to add command to ribbon

Once plugin is installed, developer can also manage more advance command appearance properties and ribbon settings using configuration designer in Petrel. Launch configuration designer use CTRL + F2. Configuration designer is available with Ocean Marketing License. Refer to Quick Start for Ocean UX white paper in Ocean for Petrel CHM for detail on how to use configuration designer.
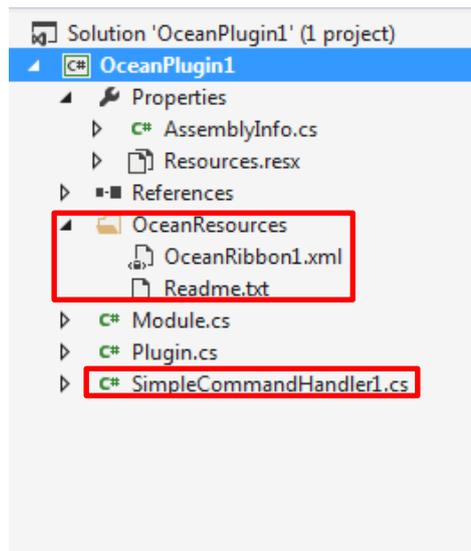
Wizard generated files and resources

Figure 13: Preview of Solution Explorer with New Command resources

OceanResources folder which will contain OceanRibbon.xml and Readme.txt. OceanRibbon.xml is also registered as resources of the plugin-in project (Figure 14)
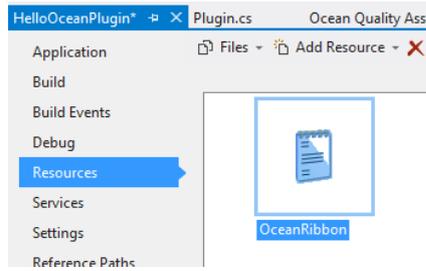
Figure 14: OceanRibbon.xml registed as plug-in project resources

o <CommandName>handler.cs implements SimpleCommandHandler. The default implementation will output to Petrel logger window when command button is clicked.

Figure 15: Command Handler implementation.

Viewing the Results

Output in Petrel will be ribbon labelled Ocean, ribbon tab and a command. Note: In this example the command created is a simple command
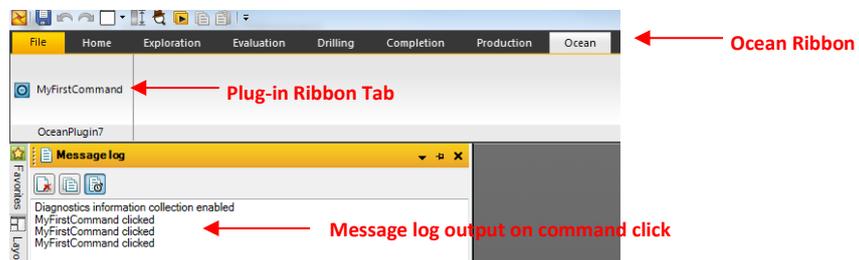
EXTENDING THE DATA DOMAIN

Ocean for Petrel allows you to customize and extend the Petrel application by creating new data objects, called custom domain objects. Custom domain objects can contribute implementations for, and participate in, standard Petrel behavior. They can be added to the Input and Model trees, or they can be displayed in a standard Petrel window.

You can decide what functionality to implement for a given custom domain object by deciding what interfaces (services) to implement and register with Ocean for Petrel. All are optional in terms of Ocean requirements.

This chapter briefly describes how to write a simple custom domain object and have it participate in standard Petrel behavior.

However, this document is only intended to help you get started using the Ocean for Petrel API, and not every feature is covered here. For a complete discussion of the Ocean for Petrel features, functionality and usage, refer to the *Ocean Developer's Guide*.

Basic Custom Domain Object

A custom domain object in Petrel is just a new class. The most basic custom domain object is a class that inherits from the .NET **object**, even when it does not implement any Ocean interfaces. Consider a simple object that has a position in space and a radius that defines its size.

```csharp
public class XYZObj
{
  private float m_x = 100.0f;
  private float m_y = 100.0f;
  private float m_z = 100.0f;
  private float m_radius = 100.0f;

  public float X
  {
    get { return m_x; }
    set { m_x = value; }
  }

  public float Y
  {
    get { return m_y; }
    set { m_y = value; }
  }

  public float Z
  {
    get { return m_z; }
    set { m_z = value; }
  }
```

```
public float Radius
{
  get { return m_radius; }
  set { m_radius = value; }
}
}
```

This basic object is a valid custom domain object that can participate in Petrel behavior. It can be added to the Petrel Input and Model trees.

Adding to Input and Models Trees

The Input tree and the Models tree support the addition of new nodes (domain objects) via the Project domain object.

Adding to Native Petrel Domain Objects

Custom domain objects may be added as children to some native Petrel domain objects using the **Slb.Ocean.Petrel.Basics.IExtensions** interface:

```
public interface IExtensions : IEnumerable<object>, IEnumerable
{ ...
  void Add(      object item );
  bool Remove(   object item );
  bool Contains( object item );
}
```

Use the Project.Extensions property to add to the Input tree, and the Project.ModelsExtensions property to add to the Models tree. You must use a transaction when adding the children. The transaction is created by the **DataManager** class. Lock the object that is going to have new children, then commit the transaction to complete the operation.

```
using (ITransaction txn = DataManager.NewTransaction())
{
  Project proj = PetrelProject.PrimaryProject;
  txn.Lock(proj);
  proj.Extensions.Add(new XYZObj( ));
  proj.ModelsExtensions.Add(new XYZObj( ));
  txn.Commit();
}
```


Figure 16 – XYZ Object in Input and Models Trees

This example adds a **XYZObj** custom domain object to a **Borehole**, a native Petrel domain object. Begin with a **Borehole** object. Lock the **Borehole** since you are going to change its children. Add a new **XYZObj** using **IExtensions**. Putting it all together, you have the following:

```
Slb.Ocean.Petrel.DomainObject.Well.Borehole bore = ...;
using (ITransaction txn = DataManager.NewTransaction())
{
  txn.Lock(bore);
  bore.Extensions.Add(new XYZObj());
```

```
    txn.Commit();
}
```
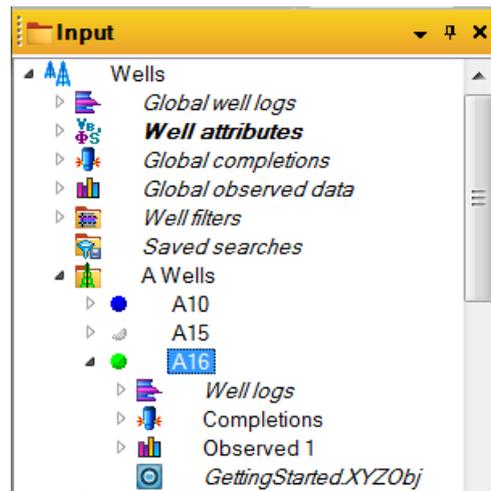
The following figure shows the result.

Figure 17: XYZ Object added to a Borehole object (A10)

Adding an Object from a Context Menu

All native Petrel objects have context menus that appear when the user right clicks on the object in the tree.

In the next example, you will add a context menu item to the **Borehole** context menu, which will add a **XYZObj** as a child.

To do so, complete the following procedure:

Define a context menu item using the **SimpleContextMenuHandler<TDomainObject>** class:

```
public class SimpleContextMenuHandler<TDomainObject> :
  ContextMenuHandler<TDomainObject> where TDomainObject : class
{ ...
  public SimpleContextMenuHandler ( string, System.Drawing.Bitmap,
        bool, ContextMenuClickedEventHandler<TDomainObject> );
}
```

Add the new context menu item in the module's **IntegratePresentation** method. Create a **Borehole** context menu item. Add it to the Petrel UI using the **PetrelSystem.ToolService** service.

```
SimpleContextMenuHandler<Borehole> cItem =
    new SimpleContextMenuHandler<Borehole>("Insert XYZ object",
        PetrelImages.Ball, false, AddXYZClick );
PetrelSystem.ToolService.AddContextMenuHandler( cItem );
```

Provide an event handler to respond to the click of the context menu item. The **AddXYZClick** event handler will add the **XYZObj** object to the **Borehole** from the context menu.

```
public void AddXYZClick( object sender, ContextMenuClickedEventArgs<Borehole> e )
{
  Borehole bore = e.ContextObject as Borehole;
  using (ITransaction txn = DataManager.NewTransaction())
  {
    txn.Lock(bore);
    bore.Extensions.Add(new XYZObj());
    txn.Commit();
```

```
  }
}
```

The new "Insert XYZ object" context menu item is added to the bottom of the **Borehole** context menu as shown in the following figure.
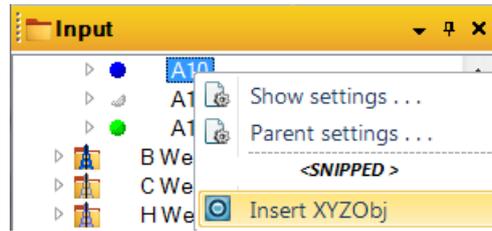

Figure 18 – Context Menu to Insert XYZ Object

Customizing Tree Presentation

A new custom domain object has a default presentation style when displayed in the tree. The icon is the Ocean "O" icon, and the text is **object.ToString**.
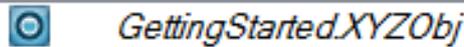

Figure 19 - Default Presentation in Tree

You can control an object's appearance in the Petrel trees by implementing the **Slb.Ocean.Petrel.UI.INameInfoSource** and **Slb.Ocean.Petrel.UI.IImageInfoSource** interfaces and providing implementations of the abstract **NameInfo** and **ImageInfo** classes.

INameInfoSource and **IImageInfoSource** allow the custom domain object to provide a text string and a bitmap to be displayed in the user interface. Implementing **INameInfoSource** and **IImageInfoSource** for the **XYZObj** object is as follows:

```csharp
using Slb.Ocean.Petrel.UI;
public class XYZObj : INameInfoSource, IImageInfoSource
{ ...
  private Bitmap myRedCircle = MyResources.XYZObjImage;
  private NameInfo nameInfo;
  public NameInfo NameInfo
  {
    get
    {
        if (nameInfo == null) nameInfo = new DefaultNameInfo("XYZ Object",
                            "XYZ Object", "GettingStarted.XYZObj");
        return nameInfo;
    }
  }
  private DefaultImageInfo m_imgInfo;
  public ImageInfo ImageInfo
  {
    get
    {
        if (m_imgInfo == null) m_imgInfo = new DefaultImageInfo(redCircle);
        return m_imgInfo;
    }
  }
}
```

MyResources contains a resource for the custom bitmap.

Figure 20 - Custom Presentation of XYZ Object

Rendering a Custom Domain Object

A custom domain object may be rendered in standard Petrel windows. The 3D and 2D windows render data using OpenInventor (Oiv). The Map, Intersection, and Interpretation windows use .NET GDI+ to render data, and the WellSection window uses INT GeoToolkit.

3D Window Display

You must perform the following steps to render a custom object in the 3D or 2D window.

- Implement a renderer for the 3D window for the object type.

- Add the renderer service to Ocean.

Implement the **IWindow3DRenderer** interface to provide a renderer for the 3D window for displaying your object. Create a class to implement **IWindow3DRenderer** and implement the following **IWindow3DRenderer** methods. **CanCreate** determines if the domain object can be rendered in the window. It can use properties of the object or the context to decide. **Create** is called the first time an OpenInventor scenegraph is needed to display an object. It creates the OpenInventor node structure. The position of the **XYZObj** object is translated into the window coordinates. A sphere is defined to represent it, and is added to the node hierarchy. The **Update** method is called when the scenegraph may be stale. (Refer to OpenInventor documentation for details.) **Dispose** should clean up any resources associated with the scenegraph and the node you are sent.

The complete OpenInventor factory class for rendering the **XYZObj** object is shown in the following example：

```
using OIV.Inventor.Nodes;
using Slb.Ocean.Petrel.DomainObject;
using Slb.Ocean.Petrel.UI;

public class MyXYZ3DRenderer : IWindow3DRenderer
{
  public bool CanCreate( object o, Window3DContext ctx )
  {
    return true;
  }

  public SoNode Create( object o, Window3DContext ctx )
  {
    XYZObj xyz = o as XYZObj;
    SoSeparator root = new SoSeparator( );
    OIV.Inventor.SbVec3f vec3 = window.WorldToWindow(new Point3(xyz.X, xyz.Y, xyz.Z));
    // Create a translation
    trans = new SoTranslation();
```

```
      trans.translation.Value = vec3;
      root.AddChild(trans);
      // create a sphere
      mySphere = new SoSphere();
      mySphere.radius.Value = xyz.Radius;
      root.AddChild(mySphere);
      return root;
   }


   public void Update(SoNode n, object o, Window3DContext ctx)
   {
      return; // nothing in this example
   }


   public void Dispose(SoNode n, object o, Window3DContext ctx )
   {
      n.Dispose();
   }
}
```

Add the OpenInventor factory as a service in the module's **Initialize** method to display the
XYZObj object.

```
public void Initialize ( )
{
  Type objType    = typeof( XYZObj );
  Type factoryType = typeof(IWindow3DRenderer);
  XYZObjOivFactory factory = new MyXYZ3DRenderer( );
  CoreSystem.Services.AddService( objType, factoryType, factory);
}
```

When a 3D window is active, the **XYZObject** object will have a checkbox next to it indicating that
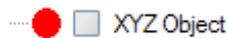it may be displayed.

Figure 21 – XYZ Object Ready to Display in 3D Window

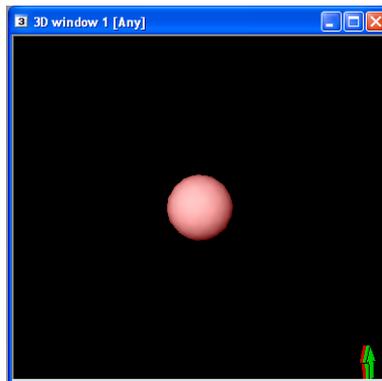When the end user clicks the checkbox the object will be displayed.

Figure 22 – XYZObj Object in 3D Window

Map Window Display

Complete the following steps to render the custom domain object in the Map window:

- Implement the **IMapWindow** factory interface.

- Add the factory to Ocean as a service.

Implement the **IMapRenderer** interface to display the **XYZObj.** Define the class that implements the **IMapRenderer** interface. In this example, you will render the object as a filled circle. Implement the **Initialize** method and set the rendering layer in which the object should be drawn. The **CanDraw** method reports whether or not the renderer knows how to draw the object. In this example, it will always return true. **Draw** renders a representation of the object into the window using the appropriate GDI+ drawing surfaces. It obtains the world coordinates from the context for drawing. It defines the brush style and color and uses the properties of the **XYZObj** instance to draw a filled ellipse. **GetBounds** computes the bounding box that the instance occupies and returns a **Box2**. **Dispose** has nothing to do for this simple **XYZObj** object.

The complete **IMapRenderer** implementation for **XYZObj** is as follows:

```
using System.Drawing;
using System.Drawing.Drawing2D;
using Slb.Ocean.Geometry;
using Slb.Ocean.Petrel.UI;

public class XYZObjMapDisplay : IMapRenderer
{
  public void Initialize ( object o, MapRendererContext ctx )
  {
    ctx.RenderingLayers = RenderingLayers.Solid;
  }

  public bool CanDraw ( object o, MapRendererContext ctx )
  {
    return true;
  }

  public void Draw ( object o, MapRendererContext ctx )
  {
    // Get the World coordinates
    Graphics gworld = ctx.World;
    using (Brush br = new SolidBrush(Color.Blue))
    {
      XYZObj xyz = (XYZObj)o;
      int x = (int)xyz.X;
      int y = (int)xyz.Y;
      int size = (int)xyz.Radius;

      // Draw the circle
      gworld.FillEllipse(br, x-size, y-size, size*2, size*2);
    }
  }

  public Box2 GetBounds ( object o, MapRendererContext ctx )
```

```
  {
    XYZObj xyz = (XYZObj)o;

    double x = xyz.X;

    double y = xyz.Y;

    double size = xyz.Radius;

    Point2 begin = new Point2( x - size, y - size );

    Point2 end = new Point2( x + size, y + size );


    // Return a box that describes the area the object occupies

    return new Box2( begin, end );

  }


  public void Dispose ( object o, MapRendererContext ctx )

  {

    return;

  }

}
```

Register the drawing service with Ocean in the module's **Initialize** method.

```
Type    xyzType     = typeof(XYZObj);

Type    factoryType = typeof(IMapRenderer);

XYZObjMapDisplay mapFactory = new XYZObjMapDisplay();


CoreSystem.Services.AddService(xyzType, factoryType, mapFactory);
```

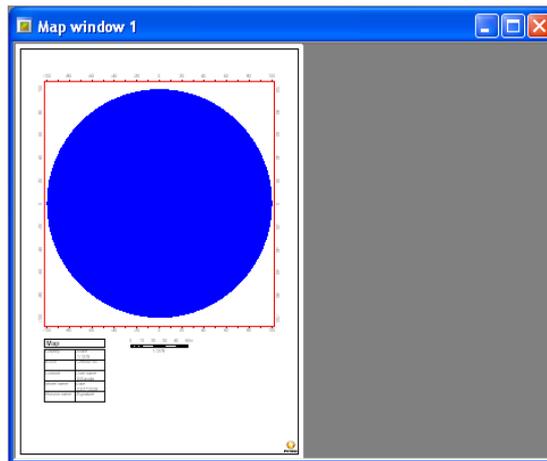When the end user checks the checkbox, the **XYZObject** is rendered in the Map window.

Figure 23 – XYZ Object in Map Window

Saving Custom Domain Objects

You may save a custom domain object into the Petrel project using a structured archived data source provided or a custom data source. If the structured archive data source does not meet your needs, then you should use the custom data source approach; information for it may be found in the **IDataSource** interface documentation for the Ocean API. For the simple example here we will illustrate saving the custom XYZ object using the structured archive data source.

Structured Archive Data Source

To use the structured archive data source:

- Implement **DataSourceFactory** and create the data source.

- Make the class **IIdentifiable**.

- Tag the class and its member to serialize with attributes.

- Add the data source factory to register the data source.

- Add your custom domain object to a Petrel tree.

You have a new **DataSourceFactory** class to register the data source. Here you will create the structured data source with a unique identifier, and you will provide a list of the type of object(s) to serialize using the structured archive data source.

```
class XYZObjDataSourceFactory : DataSourceFactory
{
  // singleton instance of data source
  private static string DataSourceId = "UniqueXYZObjDataSourceId";
  public static StructuredArchiveDataSource Get(IDataSourceManager dsm)
  {
    return dsm.GetSource(DataSourceId) as StructuredArchiveDataSource;
  }
  public override Slb.Ocean.Core.IDataSource GetDataSource()
  { // create data source with unique id and type of object to be saved
    return new StructuredArchiveDataSource( DataSourceId, new[] {typeof(XYZObj)});
  }
}
```

Have the **XYZObj** class implement **IIdentifiable**. You will use the structured data source to generate the Droid and add a custom object instance to the data source. Make the following additions to your **XYZObj** class.

```
public class XYZObj : ..., IIdentifiable
{ ...
  Droid m_droid;
  StructuredArchiveDataSource m_dataSource;

  // add Constructor
  public XYZObj()
  {
    IDataSourceManager mgr = DataManager.DataSourceManager;
    this.m_dataSource = XyzObjDataSourceFactory.Get(mgr);
    this.m_droid = datasource.GenerateDroid();
    datasource.AddItem(m_droid, this);
    // set initial values:
    m_x = 100.0f; m_y = 0.0f; m_z = 0.0f; m_radius = 100.0f;
  }
  public void Droid Droid
  {
    get { return m_droid; }
  }
}
```

You must also tag your class and any members you want to serialize into the structured archive

data source with one of the following attributes:

- **ArchivableAttribute** on the class

- **ArchivedAttribute** on the member

Add the attributes to your **XYZObject**.

```
[Archivable]
public class XYZObj : IIdentifiable, ...
{ ...
  [Archived]
  float m_x;
  [Archived]
  float m_y;
  [Archived]
  float m_z;
  [Archived]
  float m_radius;
  // can provide name:
  [Archived(Name = "Droid")]
  Droid m_droid;
  StructuredArchiveDataSource m_dataSource;
}
```

You must register your data source factory with the system in your **Module.Integrate** using the **PetrelSystem.AddDataSourceFactory** method:

```
public class Module : IModule, ...
{ ...
  public void Integrate()
  { ...
    PetrelSystem.AddDataSourceFactory(new XYZObjDataSourceFactory);
  }
}
```

Next you will create your object and add it to the tree, providing the data source instance in the object's constructor. You will retrieve your data source factory from the **Get** method you added to it above.

```
public void CreateXYZObj()
{
  // create XyzObj instance and provide data source
  XyzObj xyz = new XyzObj();
  Project project = PetrelProject.PrimaryProject;
  using (ITransaction t = DataManager.NewTransaction())
  {
    t.Lock(project);
    // Add XyzObj to project extensions.
    project.Extensions.Add(xyz);
    t.Commit();
  }
}
```

When you save the project, the **XYZObject** is saved with it. Reopening the project will load this data back into Petrel.