Product Documentation Copyright and Terms of Use

Copyright

© 2018 Hexagon AB and/or its subsidiaries and affiliates. All rights reserved. Hexagon has registered trademarks in many countries throughout the world. Visit the Trademarks Page http://www.hexagongeospatial.com/legal/trademarks for information about the countries in which the trademarks are registered. See Product Page and Acknowledgments for more information.

Product Documentation Terms of Use

PLEASE READ THESE TERMS CAREFULLY BEFORE USING HEXAGON GEOSPATIAL’S DOCUMENTATION (“DOCUMENT”). USE OF THIS DOCUMENT INDICATES ACCEPTANCE OF THIS AGREEMENT WITHOUT MODIFICATION. IF YOU DO NOT AGREE TO THE TERMS HEREOF (“TERMS”), DO NOT USE THIS DOCUMENT.

Use Of This Document

All materials in this Document are copyrighted and any unauthorized use may violate worldwide copyright, trademark, and other laws. Subject to the terms of this Agreement, Hexagon Geospatial (a Division of Intergraph Corporation) and Intergraph’s subsidiaries (“Intergraph”) hereby authorize you to reproduce this Document solely for your personal, non-commercial use. In consideration of this authorization, you agree to retain all copyright and other proprietary notices contained therein. You may not modify the Materials in any way or reproduce or publicly display, perform, or distribute or otherwise use them for any public or commercial purpose, except as specifically authorized in a separate agreement with Hexagon Geospatial.

The foregoing authorization specifically excludes content or material bearing a copyright notice or attribution of rights of a third party. Except as expressly provided above, nothing contained herein shall be construed as conferring by implication, estoppel or otherwise any license or right under any copyright, patent or trademark of Hexagon Geospatial or Intergraph or any third party.

If you breach any of these Terms, your authorization to use this Document automatically terminates. Upon termination, you will immediately destroy any downloaded or printed Materials in your possession or control.

Disclaimers

ALL MATERIALS SUPPLIED HEREUNDER ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. Hexagon Geospatial does not warrant that the content of this Document will be error-free, that defects will be corrected, or that any Hexagon Geospatial Website or the services that make Materials available are free of viruses or other harmful components.

Hexagon Geospatial does not warrant the accuracy and completeness of this Document. Hexagon Geospatial may make changes to this Document at any time without notice.

Limitation Of Liability

IN NO EVENT SHALL HEXAGON GEOSPATIAL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, ARISING FROM YOUR ACCESS TO, OR USE OF, THIS DOCUMENT.
Indemnification
You agree to defend, indemnify, and hold harmless Hexagon Geospatial, its officers, directors, employees, and agents from and against any and all claims, liabilities, damages, losses or expense, including reasonable attorneys' fees and costs, arising out of or in any way connected with your access to or use of this Document.

Use Of Software
Use of software described in this Document is subject to the terms of the end user license agreement that accompanies the software, if any. You may not download or install any software that is accompanied by or includes an end user license agreement unless you have read and accepted the terms of such license agreement. Any such software is the copyrighted work of Hexagon Geospatial, Intergraph or its licensors. Portions of the user interface copyright 2012-2018 Telerik AD.

Links To Third Party Websites
This Document may provide links to third party websites for your convenience and information. Third party websites will be governed by their own terms and conditions. Hexagon Geospatial does not endorse companies or products to which it links.

Third party websites are owned and operated by independent parties over which Hexagon Geospatial has no control. Hexagon Geospatial shall not have any liability resulting from your use of the third party website. Any link you make to or from the third party website will be at your own risk and any information you share with the third party website will be subject to the terms of the third party website, including those relating to confidentiality, data privacy, and security.

Trademarks
The trademarks, logos and service marks ("Marks") displayed in this Document are the property of Hexagon Geospatial, Intergraph or other third parties. Users are not permitted to use Marks without the prior written consent of Hexagon Geospatial, Intergraph or the third party that owns the Mark. "Intergraph" is a registered trademark of Intergraph Corporation in the United States and in other countries. Other brands and product names are trademarks of their respective owners.


Procedure For Making Claims Of Copyright Infringement
Notifications of claimed copyright infringement should be sent to Hexagon Geospatial by mail at the following address: Intergraph Corporation, Attn: Intergraph Legal Department, P.O. Box 240000, Huntsville, Alabama 35824.

US Government Restricted Right
Materials are provided with "RESTRICTED RIGHTS." Use, duplication, or disclosure of Materials by the U.S. Government is subject to restrictions as set forth in FAR 52.227-14 and DFARS 252.227-7013 et seq. or successor provisions thereto. Use of Materials by the Government constitutes acknowledgment of Hexagon Geospatial or Intergraph's proprietary rights therein.

International Use
You may not use or export Materials in violation of U.S. export laws and regulations. Hexagon Geospatial makes no representation that Materials are appropriate or available for use in every country, and access to them from territories where their content is illegal is prohibited.

Hexagon Geospatial provides access to Hexagon Geospatial international data and, therefore, may contain references or cross references to Hexagon Geospatial products, programs and services that
are not announced in your country. These references do not imply that Hexagon Geospatial intends to announce such products, programs or services in your country.

The Materials are subject to U.S. export control and economic sanctions laws and regulations and you agree to comply strictly with all such laws and regulations. In addition, you represent and warrant that you are not a national of, or otherwise located within, a country subject to U.S. economic sanctions (including without limitation Iran, Syria, Sudan, Cuba, and North Korea) and that you are not otherwise prohibited from receiving or accessing the Materials under U.S. export control and economic sanctions laws and regulations. Hexagon Geospatial makes no representation that the Materials are appropriate or available for use in every country, and access to them from territories where their content is illegal is prohibited. All rights to use the Materials are granted on condition that such rights are forfeited if you fail to comply with the terms of this agreement.

Revisions

Hexagon Geospatial reserves the right to revise these Terms at any time. You are responsible for regularly reviewing these Terms. Your continued use of this Document after the effective date of such changes constitutes your acceptance of and agreement to such changes.

Applicable Law

This Document is created and controlled by Hexagon Geospatial in the State of Alabama. As such, the laws of the State of Alabama will govern these Terms, without giving effect to any principles of conflicts of law. You hereby irrevocably and unconditionally consent to submit to the exclusive jurisdiction of the United States District Court for the Northern District of Alabama, Northeastern Division, or the Circuit Court for Madison County, Alabama for any litigation arising out of or relating to use of this Document (and agree not to commence any litigation relating thereto except in such courts), waive any objection to the laying of venue of any such litigation in such Courts and agree not to plead or claim in any such Courts that such litigation brought therein has been brought in an inconvenient forum. Some jurisdictions do not allow the exclusions or limitations set forth in these Terms. Such exclusions or limitations shall apply in all jurisdictions to the maximum extent allowed by applicable law.

Questions

Contact us https://www.hexagongeospatial.com/about-us/our-company/contact-us with any questions regarding these Terms.
About Us

Hexagon Geospatial helps you make sense of the dynamically changing world. We enable you to envision, experience and communicate geographic information. Our technology provides you the form to design, develop and deliver solutions that solve complex, real-world challenges. Ultimately, this is realized through our creative software products and platforms.

CUSTOMERS. Globally, a wide variety of organizations rely on our products daily including local, state and national mapping agencies, transportation departments, defense organizations, engineering and utility companies, and businesses serving agriculture and natural resource needs. Our portfolio enables these organizations to holistically understand change and make clear, reliable decisions.

TECHNOLOGY. Our priority is to deliver products, platforms and solutions that make our customers successful. Hexagon Geospatial is focused on developing technology that displays and interprets information in a personalized, meaningful way. We enable you to transform location-based content into dynamic and useable business information that creatively conveys the answers you need.

PARTNERS. As an organization, we are partner-focused, working alongside our channel to ensure we succeed together. We provide the right platforms, products, and support to our business partners so that they may successfully deliver sophisticated solutions for their customers. We recognize that we greatly extend our reach and influence by cultivating channel partner relationships both inside and outside of Hexagon (http://www.hexagon.com).

TEAM. As an employer, we recognize that the success of our business is the result of our highly motivated and collaborative staff. At Hexagon Geospatial, we celebrate a diverse set of people and talents, and we respect people for who they are and the wealth of knowledge they bring to the table. We retain talent by fostering individual development and ensuring frequent opportunities to learn and grow.

HEXAGON. Hexagon’s solutions integrate sensors, software, domain knowledge and customer workflows into intelligent information ecosystems that deliver actionable information. They are used in a broad range of vital industries.

Hexagon (Nasdaq Stockholm: HEXA B) has approximately 18,000 employees in 50 countries and net sales of approximately 3.3bn USD. Learn more at hexagon.com (http://www.hexagon.com) and follow us @HexagonAB https://twitter.com/HexagonAB.
Contents

Product Documentation Copyright and Terms of Use ................................................................. 2
About Us ..................................................................................................................................... 5
Introduction ............................................................................................................................... 8
Environment ............................................................................................................................... 9
Spatial Modeler Basics ............................................................................................................... 10
Introduction to the Examples .................................................................................................... 15
Build and Run Spatial Models in C++ ......................................................................................... 16
  Build Spatial Models in C++ ..................................................................................................... 16
  Example 1: Run an NDVI ......................................................................................................... 16
  Load and Save Spatial Models ................................................................................................. 18
  Example 2: Load/Save/Edit a Spatial Model ........................................................................... 19
Extend Spatial Modeler .............................................................................................................. 20
  Create an Operator .................................................................................................................. 21
  Example 3: View Image Operator ............................................................................................ 21
  Example 4: Determine whether pixel is nodata .................................................................... 26
  Example 5: Lazy-mosaic Images ............................................................................................ 30
  Example 6: Accessing Raster and Scalar Data .................................................................... 36
Create a New Data Type ............................................................................................................. 46
  Example 7: Classification Signature ....................................................................................... 46
Create a Data Conversion .......................................................................................................... 54
  Example 8: time_t to DateTime ............................................................................................... 54
Creating a Customization Dialog ............................................................................................... 58
  Example 9: Color Picker ......................................................................................................... 58
Additional Information ............................................................................................................. 64
  Best Practices ......................................................................................................................... 64
Contents

Models and Submodels .................................................................................................................. 64
Attributes ....................................................................................................................................... 65
Creating Visual Studio Projects for Spatial Modeler Plug-ins ....................................................... 65
  Copy and Modify an Example Project ......................................................................................... 65
  Creating a Project from Scratch ................................................................................................. 68
Run a Spatial Model from the Command Line .............................................................................. 79
Test What You've Developed for Thread Safety ......................................................................... 81
What's new in Spatial Modeler 2018 ............................................................................................ 83
  Highlighted API Changes ........................................................................................................... 83
Technical Support and Information ............................................................................................... 85
Index ............................................................................................................................................... 87
SECTION 1

Introduction

The Spatial Modeler SDK is a C++ toolkit for building, modifying, and running workflows on geospatial data. It can be used to build complex algorithms or simply to run routine tasks. It is based on the Spatial Modeler tool developed for ERDAS IMAGINE 2013. It is extensible via a plugin mechanism where objects, such as operators, data types, and configuration dialogs, are discovered at runtime by demand-loading all DLLs found in a search path and identifying the Spatial Modeler objects implemented in those DLLs. The Spatial Modeler SDK can be used to build add-ons to various Hexagon Geospatial products, such as ERDAS IMAGINE and GeoMedia, or it can be used to build and run workflows within your own product.

This document assumes that the reader is familiar with C++ and Microsoft Visual Studio.
SECTION 2

**Environment**

The Spatial Modeler SDK 2018 is built on top of Visual Studio 2013 Update 2. The supported configuration and platform combinations are Win32Debug, Win32Release, x64UDebug, and x64URelease. As you might expect, the Win32 configurations build 32-bit binaries and the x64 configurations build 64-bit binaries. If you are building plug-ins for ERDAS IMAGINE, you must provide 32-bit binaries (Release only), and it is recommended that you provide 64-bit binaries as well, since IMAGINE 2018 is a hybrid system, with some executables running in 32-bit and some running in 64-bit, when possible. If you are using the SMSDK in another environment, you only need to provide binaries for whichever configurations are appropriate.
Spatial Modeler Basics

This section includes a brief overview of the basic Spatial Modeler concepts. Familiarity with the operation of the Spatial Modeler Editor may be helpful, but it is not necessary. More information about Spatial Modeler and Spatial Modeler operators and data types is located in online help. You can find a link to the online help on the Welcome page linked to in your Windows Start menu or Windows Start screen.
Spatial Modeler Basics

Spatial Modeler Objects

- Model
  - Execute
  - AddTail
  - GetOperators
  -...

- Operator
  - OnExecute
  -...

- PortConnections

- Port
  - GetData
  - GetData(request)
  -...

- Data
  - GetValueWithType
  - ToNormalizedString
  - FromNormalizedString
  - GetDataTypeName
  -...

- OperatorFactory
  - Create(namespace, name)
  -...

- ConversionFactory
  - Create(namespace, name)
  - ConversionExists(from, to)

- DataFactory
  - Create(namespace, name)
  -...

- DataRequestFactory
  - Create(namespace, name)
  -...

- URLFactory
  - Create(namespace, name)
  -...

- ModelFormatFactory
  - Open(uri)
  - Save(model, uri)
  -...

Client Application (eg Spatial Model Editor)
Model

A Spatial Model is a collection of interconnected operators. The Model object manages the list of operators which are assembled to make the model. In addition it manages the list of operators “tails”, which are the starting points for pull operations. A graphical example of a model, as it is represented in the Spatial Modeler Editor, would be the simple model below.

Operator

The main building block of Spatial Modeler is an operator. An Operator is a self-describing object that is responsible for encapsulating a computational element. It is responsible for producing zero or more outputs from zero or more inputs. These input and output points are called ports. Ports allow data to flow through an operator. They also contain information about the operator’s connections with other operators. In a connection, the operator that runs first is called the parent operator, and the operator that runs second is the child operator. In the example above the rectangular objects are operators.

Data

Data is the “unit of currency” between operators. This abstraction allows any type of data to be processed.

Conversion

A Conversion is responsible for converting between different data types. It is used to allow interconnecting operators that deal in different Data.
Port
The connection points between operators are known as ports. A port has one or more supported data types, such as raster, features (vector), string, filename, and so on. Ports are defined as either input or output. They can also contain default data. The pointed rectangles along the sides of the operator in the above diagram are ports.

Some operators will accept a variable number of the last input port. The Add operator, for example, can take any number of inputs (a minimum of two). Ports can be added to or removed from these operators.

DataRequest
A DataRequest object is responsible for framing (modularizing) a data request from an operator.

UI_Provider
UI_Provider is responsible for presenting a Dialog to show and edit the set of properties for an operator.

Other Spatial Modeler Concepts

Sub-models
Groups of operators can be selected and converted into a sub-model. You may create a sub-model to reduce the apparent complexity of a model, or you may do it so that you can reuse a piece of your model multiple times. For more information, see Models and Sub-models at the end of this document.

Flow Control vs Data Flow
The Spatial Modeler architecture is a Data Flow architecture. This means that inputs to one operation are the outputs of another operation. The order of operation is not explicitly defined, but is determined implicitly by the connections in the model. This is different from a Flow Control architecture where the sequence of steps is specified by the connectivity of the steps.

Pull Architecture
Spatial Modeler uses a pull model type of observer pattern where the end of the model chain requests data from its parent(s) and waits until the requested data is provided to it. This allows the chain to process only the data necessary to satisfy the request, so a subset of the input data may be used.
Introduction to the Examples

All referenced examples are installed as part of the Spatial Modeler SDK. They are located in the examples/smsdk subfolder of the installation directory. You can build the examples using SpatialModelerExamplesSolution.sln in the SpatialModelerExamplesSolution subfolder. For more information about specific APIs, see the User Documentation. There’s a link to the User Documentation on the Welcome page linked to in your Windows Start menu or Windows Start screen.
**Section 5**

**Build and Run Spatial Models in C++**

Using C++ you can run existing Spatial Models or build new Spatial Models by putting pieces together. The first couple of examples will show you how to do this.

**Build Spatial Models in C++**

Spatial Models can be built dynamically in C++ from the Spatial Modeler building blocks: operators, connections, and data.

**Example 1: Run an NDVI**

This example is located in smNDVIExample. This example model performs a Normalized Difference Vegetation Index (NDVI) calculation on an input raster file and saves the results in a new raster file. It shows how to create and run a model using the C++ API by creating operators, making connections, and setting data on ports.

**Dissect Example 1**

Look in NDVI.cpp at the top of the file. The first thing that must be done in any process that is going to use the Spatial Modeler API is to initialize the environment. This only needs to be done once per process. Since this particular example is distributed in SMSDK’s bin directory, the SpatialModelerInitializer with no parameter can be used. If you are initializing the environment from outside the bin directory, you need to supply the SpatialModelerInitializer with the SMSDK’s home directory (default is C:\Program Files\Hexagon\Spatial Modeler SDK 2018).

```cpp
// Initialize the Spatial Modeler environment.
SpatialModelerInitializer s_Initializer;
```

Once the environment is initialized, a Model must be created. This is the object that will store and execute the operators the NDVI model contains. This is in NDVI::RunProcess().

```cpp
ModelPtr model = boost::dynamic_pointer_cast<Model>( OperatorFactory::Create( L"IMAGINE", L"Model" ) );
```

The OperatorFactory is used to create a new operator by passing it a namespace and name. All operators must have a unique (namespace, name) pair. All the operators shipped with the SDK are located in the “IMAGINE” namespace. The operator name is always the same as the name in the Spatial Model Editor except with spaces removed (“RasterInput” instead of “Raster Input”). The newly-created operators must then be added to the model.

```cpp
OperatorPtr rasterInput = OperatorFactory::Create( L"IMAGINE", L"RasterInput" );
```
model->AddOperator( rasterInput );
OperatorPtr band1 = OperatorFactory::Create( L"IMAGINE", L"BandSelection" );
model->AddOperator( band1 );
OperatorPtr band2 = OperatorFactory::Create( L"IMAGINE", L"BandSelection" );
model->AddOperator( band2 );
OperatorPtr subtract = OperatorFactory::Create( L"IMAGINE", L"Subtract" );
model->AddOperator( subtract );
OperatorPtr add = OperatorFactory::Create( L"IMAGINE", L"Add" );
model->AddOperator( add );
OperatorPtr divide = OperatorFactory::Create( L"IMAGINE", L"Divide" );
model->AddOperator( divide );
OperatorPtr rasterOutput = OperatorFactory::Create( L"IMAGINE", L"RasterOutput" );
model->AddTail( rasterOutput );

The last operator in the chain must be added as a tail. This tells the model which operator(s) to request data from, which will pull the processing chain.

Once all the operators are added to the model, we need to set up the data connections between them.

band1->ConnectParent( rasterInput->GetPort(L"RasterOut"),
.band1->GetPort(L"RasterIn") );
band2->ConnectParent( rasterInput->GetPort(L"RasterOut"),
.band2->GetPort(L"RasterIn") );
subtract->ConnectParent( band1->GetPort(L"RasterOut"),
.subtract->GetPort(L"Input1") );
subtract->ConnectParent( band2->GetPort(L"RasterOut"),
.subtract->GetPort(L"Input2") );
add->ConnectParent( band1->GetPort(L"RasterOut"), add->GetPort(L"Input1") );
add->ConnectParent( band2->GetPort(L"RasterOut"), add->GetPort(L"Input2") );
divide->ConnectParent( subtract->GetPort(L"Output"), divide->GetPort(L"Input1") );
divide->ConnectParent( add->GetPort(L"Output"), divide->GetPort(L"Input2") );
rasterOutput->ConnectParent( divide->GetPort(L"Output"),
rasterOutput->GetPort(L"RasterIn") );
The model now contains the necessary operators and data connections. This is how it would look in the Spatial Modeler Editor.

The next step is to set the input and output filenames and the band indices for the NIR and red band.

```cpp
rasterInput->GetPort(L"Filename")->SetData(FileData)(
    L"$IMAGINE_HOME/examples/smsdk/smNDVIExample/data/lanier.img" );
rasterInput->GetPort(L"DataType")->SetDataFromString( L"Float" );
band1->GetPort(L"BandRange")->SetDataFromString( L"4" );
band2->GetPort(L"BandRange")->SetDataFromString( L"3" );
rasterOutput->GetPort(L"FilenameIn")->SetData<FileData>(
    L"$IMAGINE_HOME/examples/smsdk/smNDVIExample/data/ndvi_output.img" );
```

You can either set data on a port using a specific data type, such as FileData above, or use a string through the SetDataFromString function. This will work identical to entering a string from the properties pane in the Spatial Modeler Editor.

Now that everything is set up correctly, the model can be run.

```cpp
model->Execute();
```

Load and Save Spatial Models

Spatial Models can be saved to disk, or loaded from a saved file. These include models created in the Spatial Model Editor; in some cases it may be more convenient to create a complex model in the editor and execute it later from C++.
Example 2: Load/Save/Edit a Spatial Model
SECTION 6

Extend Spatial Modeler

Spatial Modeler can be extended by adding new operators, data types, data conversions, and dialogs for the user to set parameters for operators. All of these are plug-in pieces that are discovered at runtime by examining DLLs in specified locations. Each plug-in DLL may contain any number and combination of these plug-in objects.

By default Spatial Modeler looks for plug-in DLLs in the $IMAGINE_HOME\Configuration\Operators subfolder under the following locations

- $IMAGINE_HOME$
- $SDK_HOME$
- the Hexagon\SpatialModeler\<MajorVersion.MinorVersion> subfolder under the Common Files directory
- the directory specified in the InstallDir StringValue in the HKEY_LOCAL_MACHINE\SOFTWARE\Hexagon\SpatialModelerAddOn\<MajorVersion.MinorVersion> key in the Windows Registry

where $Configuration$ refers to one of the supported configuration and platform combinations—see the Environment section above—and $MajorVersion.MinorVersion$ refers to the Spatial Modeler SDK version number. For example, typically you would place the 32-bit release version of your plug-in DLL for Spatial Modeler 2018 into C:\Program Files\Common Files\Hexagon\SpatialModeler\16.5\usr\lib\Win32Release\Operators.

However, if you are running Spatial Models in your own environment, you may place your DLL in any location and add that directory to Spatial Modeler’s path by calling SpatialModelerEnvironment::AppendDirectory(). The examples below all place their plug-ins in the examples/smsdk/Plugins subfolder in the installation directory. See the source code for smExampleModelRunner for how this directory is added to Spatial Modeler’s path.

Another way to define the location of a plugin DLL is through an XML file located in Spatial Modeler’s path. This is useful when you don’t want Spatial Modeler to load every DLL in a directory while searching for a plugin DLL. Using the XML file lets you directly define the path of the DLL to use, and any additional search paths to find support DLLs. An example of the contents of the XML file is shown below.

```xml
<?xml version="1.0" encoding="utf-8"?>
<SpatialModelerPath xmlns="http://tempuri.org/XMLSchema.xsd">
  <DLL>$MY_APP_HOME\some_other_directory\testPlugin.dll</DLL>
  <SearchPath>$MY_APP_HOME\some_other_dll_path</SearchPath>
  <SearchPath>$MY_APP_HOME\yet_another_dll_path</SearchPath>
</SpatialModelerPath>
```

One of the projects built by the SpatialModelerExamplesSolution solution is smExampleModelRunner. This project builds an executable that allows you to easily run the example models provided to demonstrate the use of the plug-in objects. You can run this executable in the debugger to step through the example plug-in objects. Note that the example models use data from the examples/smsdk/data
directory and that the output of these models goes to this directory as well. The descriptions below will indicate which model demonstrates the example object(s).

Create an Operator

You can create new operators to implement your own algorithms or data operations. These operators can be used in your own tools or within various Hexagon Geospatial products. The following examples will show you how to create several operators, from simple to complex.

Example 3: View Image Operator

This example is located in sm_ExternalProcessExample. It creates a plug-in DLL containing a single operator that simply opens a specified file using either the Windows default application for that file type or Paint (mspaint.exe).

**Dissect Example 3**

**View Image operator class**

Let's start by looking at the operator itself. The code for the operator can be found in ViewImage.h and ViewImage.cpp. Look at definition of class ViewImage in ViewImage.h. The first thing we notice is that an operator in Spatial Modeler must be a class that derives from Operator. Operator is in the HexGeo::SpatialModeler namespace.

```cpp
class ViewImage : public HexGeo::SpatialModeler::Operator {
    ViewImage();
    virtual ~ViewImage();
}
```

There are a few Operator methods that all sub-classed operators must implement. The Spatial Modeler SDK has macros to help with some of them.

Use the SM_PLUGIN_OBJECT macro to set the namespace and name of an object—in this case our operator. L"Example" is the namespace for the operator. This is not necessarily the same as the C++ namespace. The namespace is intended, as in C++, to further define the operator name and reduce the risk of duplication. It is used when instantiating the operator using OperatorFactory::Create(). The namespace should not contain spaces. The operator(s) you write should not use "Example" as the namespace. L"ViewImage" is the name of the operator. The name is used when registering the operator—we'll discuss this below—and when instantiating the operator using OperatorFactory::Create().
The operator name does not have to match the name of the operator class; however, it should not contain spaces.

```cpp
SM_PLUGIN_OBJECT ( L"Example", L"ViewImage" );
```

Use the `OPERATOR_DISPLAY_INFO` macro to set the display information of an operator. The first parameter is the name of the category for the operator. The category name is used for grouping operators in the Spatial Modeler Editor Operator list. To include an operator in multiple categories, separate them in the string using ‘|’, e.g. setting the category to “Input|Raster” would include the operator in the “Input” category and the “Raster” category. Categories are dynamic—you do not have to restrict yourself to “existing” categories. The second parameter is the default display name of the operator. The display name is displayed on the operator in the Spatial Modeler Editor. It is also used when searching for a specific operator within a model. The convention is that the display name be the same as the operator name, plus spaces to separate words. The third parameter is the name of the icon file to be displayed on the operator in the Spatial Modeler Editor. If the file name is empty, the default icon will be used. The `ETXT_TEXT_TR` macro is used to allow these names to be localized in the ERDAS IMAGINE GUI.

```cpp
OPERATOR_DISPLAY_INFO ( ETXT_TEXT_TR ( "Example" ), ETXT_TEXT_TR ( "View Image" ), L"" );
```

Use the `OPERATOR_DESCRIPTION` macro to set the description of an operator. The description should be a short or medium-length explanation of the purpose of the operator. The description displays in the “bubble help”, when the cursor hovers over an operator on the canvas in the Spatial Model Editor.

```cpp
OPERATOR_DESCRIPTION (ETXT_TEXT_TR ( "Opens an image in the windows default viewer." ) );
```

Two other methods that must be implemented are `Init()` and `OnExecute()`. `Init()` is called when the operator is created. `OnExecute()` is called when the model is executed (run). We’ll discuss these two methods more when we examine ViewImage.cpp.

```cpp
virtual void Init();
virtual void OnExecute();
```

The `ShowHelp()` method may optionally be implemented. This method shows the online help for the operator. We’ll discuss this method more when we examine ViewImage.cpp.

```cpp
virtual bool ShowHelp();
```

Another method that is should be overridden by 3rd-party developers is `Provider()`. This method is used to specify the company or organization that developed the plug-in.

```cpp
virtual std::wstring Provider();
```

The other macro you’ll find in ViewImage.h is `OPERATOR_NOT_USABLE_IN_EXPRESSION`. This macro is used to indicate that this operator is not usable in a Spatial Modeler expression (one of the built-in Spatial Modeler operators). Expressions are textual ways of describing a chain of operators, such
Extend Spatial Modeler

as Add ($Input1, $Input2) / Divide ($Input1, $Input2). An expression must have an output, so in order for an operator to be usable in an expression, it must have an output, and since our example operator here is designed to be used at the end of a chain and does not have an output, it cannot be used in an expression.

OPERATOR_NOT_USABLE_IN_EXPRESSION;

ViewImage::Init() Implementation

Now let's look at how ViewImage is implemented in ViewImage.cpp. We create the ports for the operator in the Init() method. ViewImage has two ports: Filename (a file name) and UsePaint (a boolean). Both of them are input ports—basically parameters to our operator. For the Filename we really just need a string that is a file name; however, Spatial Modeler has a special data type for a file name: File. Having a special data type allows the Spatial Modeler Editor to present the user with a file chooser for selecting the file name. The Input attribute—set here to true—tells the file chooser that the file should be an existing file. The FileFilter and Extension attributes—set to L"raster" and L".tif" respectively—are used by the file chooser to determine what file are shown (by default). For the Filename port we also are setting the Kind attribute to L"raster" to properly identify the input file type for publishing models in ERDAS Apollo.

PortPtr filePort = CreatePortWithAttributes ( ETXT_TEXT_KEY("Filename"),
            FileData::GetDataType(), SB_PORT_INPUT,
            L"Input", DataFactory::Create<BoolData>(true),
            L"FileFilter", Data::Create<StringData> ( L"raster" ),
            L"Extension", Data::Create<StringData> ( L"*.tif" ),
            NULL );

filePort->SetDataTypeAttribute<StringData> ( FileData::GetDataType(), L"Kind",
            L"raster" );

The second port is a flag to tell ViewImage whether to open the image using the Windows default application for that file type or Paint. This input port supports Bool data. We're not going to require the user to set a value on this port—we'll have a default—and we're even going to hide this port (by default) from the user in the Spatial Modeler Editor GUI.

PortPtr paintPort = CreatePort<BoolData> ( ETXT_TEXT_KEY ( "UsePaint" ),
            SB_PORT_INPUT );
paintPort->SetIsOptional ( true );
paintPort->SetIsHidden ( true );

The ETXT_TEXT_KEY macro to allow the port name to be localized in the ERDAS IMAGINE GUI without affecting the name by which the port is referenced in the code.

ViewImage::OnExecute() Implementation

The OnExecute method is called when the model is executed (run). It does all of the real work of the operator, at least for non-raster operators—we'll get into raster operators later. Note that while OnExecute() is guaranteed to be called only once during the execution of a single model chain and
therefore does not need to be thread safe with respect to member variables, there may be multiple models running simultaneously in different threads in the same process it, so it does need to be thread safe with respect to any static data.

In this example this is where we’re going to find out what image the user wants us to display and display it in the manner requested. First we have to get the name of the image. We get the port for the Filename and request the value that is set on it. We request it as FileData, and we tell GetDataValue() that the default is an empty string. If we get an empty string, however, we set an error message on the operator—we have to have a file name to open the image. SetErrorMessage() throws an exception, so if we do not have a Filename, we’ll exit OnExecute() here.

```cpp
std::wstring filename = GetPort ( L"Filename" ) -> GetDataValue<FileData> ( L"" );
if ( filename.empty() )
    SetErrorMessage ( L"No file name specified." );
```

Now we get the value of the UsePaint port. We will default to false.

```cpp
bool usePaint = GetPort ( L"UsePaint" ) -> GetDataValue<BoolData> ( false );
```

The rest of the code in OnExecute() is just the code necessary to implement the algorithm or perform the data operation. In this case, we just want to open the file. One tricky little piece with file names, though, is that in the context of Spatial Modeler file names have UNIX slashes (forward slashes) in the path. In order to use the file name we got from the Filename port in Windows functions, we need to make sure that slashes are the backward slashes expected by Windows. That’s what efnp::FileNameInternalToExternal() does for us. If an error is encountered in OnExecute(), you should call SetErrorMessage() to report the error and abort the processing of the chain.

```cpp
filename = efnp::FileNameInternalToExternal ( filename );
HINSTANCE hInst;
if ( usePaint )
{
    hInst = ShellExecute ( NULL, L"open", L"mspaint.exe", filename.c_str(), NULL, SW_SHOWNORMAL );
}
else
{
    hInst = ShellExecute ( NULL, L"open", filename.c_str(), NULL, NULL, SW_SHOWNORMAL );
}
if ( (int)hInst <= 32 )
{
    SetErrorMessage ( L"Unable to open file." );
}
ViewImage::ShowHelp() Implementation

The ShowHelp method is called when the user selects "Help" for the operator in the Spatial Model Editor. Implementing this method allows a developer to override the default behavior of Spatial Model Editor, which displays an HTML file in the Spatial Modeler help system. If this method is not implemented or it returns false, and the appropriate HTML file cannot be found in the Spatial Modeler help system, an error will be displayed to the user. In this example we are simply displaying a Windows Message Box, but you can do whatever you want for your operator (display a webpage, display a dialog, etc.).

MessageBox(NULL, L"View Image opens a specified file using either the Windows default application for that file type or Paint.", L"Help for View Image operator", MB_OK);

If the method is able to display the help, it should return true; otherwise, it should return false.

return true;

ViewImage::Provider() Implementation

The Provider method is used to provide the name of the company or organization that developed the plug-in. This name shows up in the Operator Info pane in the Spatial Model Editor.

return L"My Company Name";

Register the plug-in

A plug-in DLL must register all of the plug-in objects that it contains. Look in SMEternalProcessExamplePlugin.cpp. In here we implement GetAvailablePluginProxiesOfType(). When the DLL is discovered at runtime, it is dynamically loaded, and the PluginManager (out of the scope of this documentation) looks for this entry point (so make sure it is exported). If it is found, it will be called for each plug-in object type: "Operator", "Data", "Conversion", and "UIProvider". Each time it is called, it should return a list of the objects of that type that are implemented in the DLL. In this example, the only object we've implemented is an operator, so we only need to return anything when the type is L"Operator". For each object we use the macro SM_PLUGIN_REGISTRATION to create an AvailablePluginProxy. The single argument to the macro is the class that implements the operator. The return value of the macro gets pushed on to the list of plug-ins.

The spatial modeler plugin DLLs also support lazy loading, which will allow a plugin DLL to only be loaded once an operator is actually created. To use lazy loading in your operator, you must support an empty type for each plugin object registered by your DLL. This is done through the type == L"" portion of the if-statement below. Also you need to run a configuration utility as a custom build step. The command can be seen in the example projects located in the Custom Build Step property.

extern "C" __declspec(dllexport) int GetAvailablePluginProxiesOfType ( const PluginType& type, AvailablePluginProxyList& pluginList )
{
//
if ( type == L"Operator" || type == L"" )
{
    pluginList.push_back (SM_PLUGIN_REGISTRATION ( sm_ExternalProcessExample::ViewImage ) );
}
return 0;
}

Example 4: Determine whether pixel is nodata

This example is located in sm_RasterExample. The sm_RasterExample project implements three new
operators. In this example, we examine the IsNoData operator. Examples later in this guide explain the
other two operators in this project. IsNoData creates a plug-in DLL that extends Spatial Modeler by
adding an operator that takes raster data as input and returns whether each pixel of the raster is
NODATA. The example model that uses the Overlay operator to perform a lazy mosaic of two or more
images is isnodata.gmdx.

IsNoData is the first of several examples that implement Raster operators. Raster operators differ from
non-raster operators in the way that data is processed. Non-raster operators process all of their data at
once in the OnExecute() method; however, raster operators process the pixels in blocks in the
ProcessBlock() method. In the OnExecute() method they only set up information about what raster data
they are able to produce and set it as RasterInfo data on the output port. In ProcessBlock() they process
the pixels for a block and put the output pixels on the output port as a TileData.

Dissect Example 4

The IsNoData operator class

Let’s start by looking at the operator itself. The code for the operator can be found in IsNoData.h and
IsNoData.cpp. The definition of the class, in IsNoData.h, is similar to that of the operator we created in
the earlier example above. But since IsNoData operates on raster data, we add the include for the header
file RasterOperator:
#include <sbrasterlib/RasterOperator.h>

Also, we derive from the class RasterOperator rather than Operator:
class IsNoData : public HexGeo::SpatialModelerRaster::RasterOperator
We use the SM_PLUGIN_OBJECT macro to set the namespace and name of the plugin, the OPERATOR_DISPLAY_INFO macro to set the display information, and the OPERATOR_DESCRIPTION macro to set the operator description as in the previous example:

```
SM_PLUGIN_OBJECT (L"Example", L"IsNoData");
OPERATOR_DISPLAY_INFO ( ETXT_TEXT_TR ("Example" ), ETXT_TEXT_TR("IsNoData"), L"" );
OPERATOR_DESCRIPTION (ETXT_TEXT_TR("Determines if a pixel is no data.") );
```

Declare the default constructor and destructor, and the required Init and OnExecute methods as in the previous example:

```
IsNoData();
~IsNoData();
```

```
void Init();
void OnExecute();
```

Since this is a raster operator, there is another method we must implement; the ProcessBlock method. Non-raster data can be processed all at once in the OnExecute method. For rasters, Spatial Modeler splits up the data into a series of blocks (rectangular arrays of pixels). Blocks are also referred to as tiles. Each block of pixels is processed using the ProcessBlock method.

```
void ProcessBlock(const HexGeo::SpatialModeler::PortPtr & port,
 const HexGeo::SpatialModelerRaster::RasterRequestPtr & rop );
```

**IsNoData::Init() Implementation**

Now let’s look at how IsNoData is implemented in IsNoData.cpp. As in the previous example, we create the ports for the operator in the Init() method. IsNoData has two ports: an input raster and an output raster. Both ports use RasterInfoData as the type supported by the port. We use SB_PORT_INPUT and SB_PORT_OUTPUT to identify the ports as input and output, respectively.

```
CreatePort<RasterInfoData> 
( 
 L"InputRaster",
 SB_PORT_INPUT
 );
```

```
CreatePort<RasterInfoData>
( 
 L"OutputRaster",
 SB_PORT_OUTPUT
 );
```
IsNoData::OnExecute() Implementation

For a raster operation, the OnExecute method performs some initialization and setup. The “real work” of the operation is performed by the ProcessBlock method.

For IsNodata::OnExecute, we first get a pointer to a RasterInfo. The RasterInfo is determined by settings in the model's Processing Properties combined with the input rasters to the model. Make sure that we’re able to get a raster.

```c++
RasterInfoPtr rinfo = GetRasterInfo();
if (!rinfo)
    SetErrorMessage(ETXT_TEXT_TR("No raster input found."));
```

IsNoData's output is binary – each output is set to True if the input pixel is NODATA, or False is the input pixel isn't NODATA. So we set the data type on the RasterInfo to binary (unsigned 1 bit):

```c++
rinfo->SetDataType(NUMBER_TYPE_U1);
```

Next we get the output port and set its data pointer to the RasterInfo:

```c++
GetPort(L"OutputRaster")->SetDataPtr<RasterInfoData>(rinfo);
```

IsNoData::ProcessBlock() Implementation

The ProcessBlock method is where we get input raster data, check each pixel to see if it is NODATA, and set the output pixels accordingly. The method is passed a pointer to the port from which the data is being requested and a pointer to a RasterRequest. The RasterRequest defines what the downstream is expecting from the operator, such as the bands and the boundary. Note that, unlike OnExecute(), ProcessBlock() will be called from multiple threads to request different tiles from the raster, so it should be completely thread safe. We recommend not modifying member variables in ProcessBlock(). If for some reason it is impossible to make the ProcessBlock() method on your Raster Operator thread-safe, you may call Operator::SetThreadSafe() in the Init() method of your operator, but performance may be affected.

```c++
void IsNoData::ProcessBlock( const PortPtr & port, const RasterRequestPtr & rop )
```

From the input port, we get a pointer to a Tile which will hold a block of pixels from the input raster. Since we’re processing just the pixels in the tile and we’re processing them band-by-band, we can pass the RasterRequest we received on to the source of our pixels. What we receive is expected to comply with the request.

```c++
ReadonlyTilePtr input = GetPort(L"InputRaster")->GetDataPtr<TileData>(rop, Port::ThrowIfEmpty);
```

Next we create the Tile for the output. The output tile will have the same number of bands, the same width, and the same height as the input tile. Since our output is binary, we set the data type to unsigned 1 bit (NUMBER_TYPE_U1).
TilePtr result = CreateTile
{
    input->GetNumBands(),
    input->GetWidth(),
    input->GetHeight(),
    NUMBER_TYPE_U1
};

Start editing the output tile.
result->StartEdit();

Loop through the pixels of the input tile, and set the value of pixels in the output tile. We use a
triple-nested loop, looping on bands, rows, and finally pixels in the row. We use GetMaskRow on the input
tile to get each row of the mask, which designates which pixels are valid data as opposed to NODATA.
The IS_MASK_VALUE_DATA macro checks that the given mask value is valid data. We use GetRow on
the output tile to get each row of output pixels. We use a Number_U1 pointer for the output row, since we
set the data type of the output tile to NUMBER_TYPE_U1.
for (long b = 0; b < input->GetNumBands(); ++b)
{
    for (long y = 0; y < input->GetHeight(); ++y)
    {
        const Byte * inputMaskRow = input->GetMaskRow(b, y);
        Number_U1 * outputDataRow = result->GetRow<Number_U1>(b, y);
        for (long x = 0; x < input->GetWidth(); ++x)
        {
            outputDataRow[x] = IS_MASK_VALUE_DATA(inputMaskRow[x]) ? 0 : 1;
        }
    }
}

Finish the editing of the output tile.
result->EndEdit();

Now we set the output port to point to the data in the output tile.
GetPort(L"OutputRaster")->SetDataPtr< TileData>(rop, result);

Register the plug-in
As in our previous example, we have to register the plug-in objects implemented in this DLL. This is in
the file SMRasterExamplePlugin.cpp.
if ( type == L"Operator" || type == L"" )
{

Example 5: Lazy-mosaic Images

This example is located in sm_RasterExample. The sm_RasterExample project implements three new operators. In this example, we examine the Overlay operator. Other examples in this guide explain the other two operators in this project. Overlay creates a plug-in DLL that extends Spatial Modeler by adding an operator that takes one or more rasters as input and returns a simple mosaic of the input rasters. The example model that uses the Overlay operator to perform a lazy mosaic of two or more images is overlay.gmdx.

Dissect Example 5

Overlay operator class

Let's start by looking at the operator itself. The code for the operator can be found in Overlay.h and Overlay.cpp. The definition of the class, in Overlay.h, is similar to that of the operator we created in the IsNoData example above. We will derive out Overlay class from RasterOperator, so that it can process raster data by tile.

class Overlay : public RasterOperator

We use the SM_PLUGIN_OBJECT macro to set the namespace and name of the plugin, the OPERATOR_DISPLAY_INFO macro to set the display information, and the OPERATOR_DESCRIPTION macro to set the operator description as in the previous examples:

SM_PLUGIN_OBJECT( L"Example", L"Overlay" );
OPERATOR_DISPLAY_INFO( ETXT_TEXT_TR("Example"), ETXT_TEXT_TR("Overlay"), L"ImageMosaic.ico" );
OPERATOR_DESCRIPTION( ETXT_TEXT_TR("Overlays multiple input raster streams."));

Declare the default constructor and destructor, and the required Init, OnExecute and ProcessBlock methods as in the previous example:

Overlay();
~Overlay();
void Init();
void OnExecute();
void ProcessBlock( const PortPtr & port,
const HexGeo::SpatialModelerRaster::RasterRequestPtr & rop );
Since we want to be able to add an arbitrary number of raster inputs, we must implement the Expand and Collapse methods. These methods will be called by Spatial Modeler when the user wishes to add or remove an input raster.

```cpp
void Expand();
void Collapse();
```

**Overlay::Init() Implementation**

Now let's look at how Overlay is implemented in Overlay.cpp. As in the previous example, we create the ports for the operator in the Init() method. Overlay initially has three ports: two input rasters and an output raster. The ports use RasterInfoData as the type supported by the port. We use SB_PORT_OUTPUT to identify the output port. Unlike the previous examples, we use our custom Expand method to add the input ports rather than using CreatePort. The Expand method will be described later.

```cpp
CreatePort<RasterInfoData> // The port data type is RasterInfoData
{
  L"OutputRaster", // The port name is "OutputRaster"
  SB_PORT_OUTPUT // The port is an output port
};
Expand();
Expand();
```

We will also need to call SetExpandible to inform Spatial Modeler that the user can add or remove input ports for this operator.

```cpp
SetExpandible(true);
```

**Overlay::Expand Implementation**

Spatial Modeler will call this method when the user wishes to add a new input port to this operator. Implementing this method allows the user to create overlays containing more than the default number (2) of raster inputs. First we will create a name for the new input port. For this operator input ports are named InputRasterN where N is the lowest number needed to avoid a duplicate port name.

```cpp
std::wstringstream name;
name << L"InputRaster" << (1 + GetInputPorts().size());
```

We will then create the new port using RasterInfoData as the supported type and SB_PORT_INPUT to identify it as an input port.

```cpp
CreatePort<RasterInfoData>
{
  name.str(),
  SB_PORT_INPUT
};
```
Overlay::Collapse Implementation

Spatial Modeler will call this method when the user wishes to remove an input port from this operator. First, we retrieve the list of input ports for this operator using GetInputPorts. We check to make sure that there are more than two input ports as it doesn’t make sense to have an overlay with only one input raster.

```cpp
PortPtrList inputs = GetInputPorts();
if (inputs.size() > 2)
{
    // Remove the port from the operator.
    RemovePort(last);
}
```

Overlay::OnExecute() Implementation

For a raster operation, the OnExecute method performs some initialization and setup. The “real work” of the operation is performed by the ProcessBlock method.

In this method, we will determine how to format our output raster and advertise that information on the output port. We will start by getting the list of input ports and initializing our output layer count to -1 to represent unset and data type to unsigned 1-bit as it is the smallest type. Then we will loop through all the input ports.

```cpp
PortPtrList inputs = GetInputPorts();
m_DataType = NUMBER_TYPE_U1;
m_NumBands = -1;
for (auto it = inputs.begin(); it != inputs.end(); ++it)
{
    RasterInfoPtr info = (*it)->GetDataPtr<RasterInfoData>
        (Port::ReturnNullIfEmpty);
    if (!info)
        continue;

    NumberType inputType = info->GetDataType();
    if (inputType > m_DataType)
m_DataType = inputType;

Then we query the raster information for its band count using GetPlaneCount. If the output band count (m_NumBands) is not set, we will set it to this raster's band count. If it is already set, we compare it with this raster's band count. If they do not match, we throw an exception since we can’t overlay images with differing band counts.

long inputBands = info->GetBandCount();
if (m_NumBands == -1)
    m_NumBands = inputBands;
else if (m_NumBands != inputBands)
    SetErrorMessage(L"All inputs must have the same number of bands");

After considering all raster inputs, we will create the output raster information to advertise on our output port. We start by getting the raster information automatically determined by settings in the model's Processing Properties combined with the input rasters to the model. We then set our calculated data type on the raster information using SetDataType. We do not need to set the band count because it will be correctly determined by Spatial Modeler. Finally, we set the raster information on the output port so it can be read by the next operator in the chain.

RasterInfoPtr rinfo = GetRasterInfo();
if (!rinfo)
    SetLastError(ETXT_TEXT_TR("No raster input found."));
else
    rinfo->SetDataType(m_DataType);
GetPort(L"OutputRaster")->SetDataPtr<RasterInfoData>(rinfo);

**Overlay::ProcessBlock() Implementation**

The ProcessBlock method is where we get input raster data and apply the mosaic operation. The method is passed a pointer to the port from which the data is being requested and a pointer to a RasterRequest.

```cpp
void Overlay::ProcessBlock( const PortPtr & port, const RasterRequestPtr & rop )
```

We begin by creating a list containing each input raster tile. We will grab all of the input ports, then iterate through them to get their raster tiles and convert them to the output data type.

```cpp
std::vector<ReadonlyTilePtr> tiles;
long numberOfBands = -1;
PortPtrList inputs = GetInputPorts();
```

For each input raster, we will get the data as a TilePtr from the port. We pass the RasterRequest object so we will get the appropriate portion of the raster. If the port is empty or does not contain raster data, we will skip the port and continue to the next one.

```cpp
ReadonlyTilePtr tile = (*it)->GetDataPtr<TileData> (rop, Port::ReturnNullIfEmpty);
if (!tile)
```
Now we once again check to make sure that the tile's band count matches the output count as we did in OnExecute.

```cpp
long inputBands = tile->GetNumBands();
if (numberOfBands == -1)
    numberOfBands = inputBands;
else if (numberOfBands != inputBands)
    EXCEPTION_THROW(__FUNCTION__, EerrException, "All inputs must have the same number of bands")
```

Once we have a valid tile, we will test its data type against the output data type (calculated in OnExecute). If it is different, we will use ConvertTile to create a new tile from it with the using the output data type. Finally, we add the tile to the tile list and end the loop.

```cpp
if (tile->GetDataType() != m_DataType)
{
    tile = ConvertTile(tile, m_DataType);
}
tiles.push_back(tile);
```

Now we will iterate through each tile to assemble the mosaic.

```cpp
TilePtr result;
for (auto it = tiles.begin(); it != tiles.end(); ++it)
{
    ReadonlyTilePtr tile = *it;
    if (!result)
    {
        result = CreateTile(tile);
        result->StartEdit();
    }
}
```

Now we will explore the actual mosaicking portion of the method. If the current tile has no valid data, we will skip it.

```cpp
if (!tile->HasData())
    continue;
```

Next we check the mosaic result tile. If it has no valid data, we will copy the current tile over it and continue to the next.

```cpp
if (!result->HasData())
{
```
result = CreateTile(tile);
result->StartEdit();
continue;
}

We get the size of each pixel in bytes to use later. Now we will iterate over each band and each row of the data.

size_t bytesPerPixel = NumberTypeUtils::GetBytesPerPixel(m_DataType);
for (long b = 0; b < result->GetNumBands(); ++b)
{
    for (long y = 0; y < result->GetHeight(); ++y)
    {
        For efficiency sake, we will get pointers to each of the data buffers for each row rather than for each pixel. We need pointers to both pixel data and mask data for both the input tile and the result tile. For the input tile we get a read only mask using GetMaskRow. For the result tile we get a writable mask using GetWritableMaskRow.
Number _U8 *resultRow = result->GetRow<Number _U8>(b, y);
Number _U8 const *tileRow = tile->GetRow<Number _U8>(b, y);
Number _U8 *resultMaskRow = result->GetWritableMaskRow(b, y);
Number _U8 const *tileMaskRow = tile->GetMaskRow(b, y);
Now we iterate through each pixel in the row.
for (long x = 0; x < result->GetWidth(); ++x)
{
    For each pixel in the row, we check to see if the result tile does not already have data (mask value is zero). If it does have data, we skip it. If it does not, we will copy the mask value from the input tile. If the input mask value indicates that the tile has data, we will copy the pixel from the input tile to the output tile.
    if (IS_MASK_VALUE_NODATA(resultMaskRow[x]))
    {
        resultMaskRow[x] = tileMaskRow[x];
        if (IS_MASK_VALUE_DATA(tileMaskRow[x]))
        {
            memcpy(&resultRow[x*bytesPerPixel], &tileRow[x*bytesPerPixel], bytesPerPixel);
        }
    }
}

Once we have processed all of the input tiles, we will end editing on the result tile using EndEdit. This will finalize its mask. Finally, we assign the result tile to the output port named “OutputRaster”.
result->EndEdit();
GetPort(L"OutputRaster")-&gt;SetDataPtr&lt;TileData&gt;(rop, result);
Register the plug-in

As in our previous example, we have to register the plug-in objects implemented in this DLL. This is in the file SMRasterExamplePlugin.cpp.

```cpp
if ( type == L"Operator" || type == L"" )
{
    pluginList.push_back ( SM_PLUGIN_REGISTRATION ( sm_RasterExample::Overlay ) );
}
```

Example 6: Accessing Raster and Scalar Data

This example is also located in sm_RasterExample. In this example, we examine the Add operator, which is the last of the three operators implemented in the sm_RasterExample project. The Add operator in this example is a somewhat-simplified version of the Add operator provided by the standard Spatial Modeler plug-ins. The example model that uses this example Add operator is add_example.gmdx.

There are a few tricky things here. First of all, our Add operator adds two or more rasters or scalars or a combination of rasters and scalars. If all inputs are scalars, the output is a scalar; however, if any of the inputs are raster, the output is a raster. That makes Add sort of a hybrid between a raster operator and a non-raster operator. Secondly, our Add operator allows the user to add color scalars and single-band rasters. This results in a three-band raster.

**Dissect Example 6**

**Add operator class**

The code for the operator can be found in Add.h and Add.cpp. Looking at the declaration of the class in Add.h, we see that, except for the private members, it is similar to the other Operator examples we've been through. In fact, it's the same as the other two raster examples. Since Add is a hybrid raster/non-raster operator, it has to also has to derive from RasterOperator and override ProcessBlock().

We will examine the private methods in a moment, but for now let's now look at the implementation of the Add operator in Add.cpp. Of interest in Add::Init() is what we do to specify that the output type may be either a raster or a scalar. First we create the output port with the RasterInfoData type, then we add the ScalarData type to the supported types.

```cpp
PortPtr port = CreatePort<RasterInfoData>(L"OutputRaster", SB_PORT_OUTPUT);
port->AddSupportedType(ScalarData::GetDataType());
```

We do the same thing in Add::Expand(), when we are creating the input ports.

```cpp
PortPtr port = CreatePort<RasterInfoData>(name.str(), SB_PORT_INPUT);
```
Add::OnExecute() Implementation

The next interesting piece of code is the implementation of Add::OnExecute(). As we've seen in the other examples, OnExecute() is the first of the Operator's methods called when the model is run. Since Add is a hybrid raster/non-raster operator, OnExecute() needs to have a dual function. If all Add is asked to do is to add scalars, then it must add the numbers together and set the result on the output port here in OnExecute(). However, if any of the inputs are rasters, Add will produce a raster, so OnExecute() just does the set-up work and leaves the pixel handling to ProcessBlock(). Let's examine OnExecute().

First we get the inputs.

```
PortPtrList inputPorts = GetInputPorts();
```

As we go through the inputs, we're going to accumulate the sum of any scalars we encounter and put any raster inputs on a list to deal with later. We're going to do this, because we have to know whether or not we have a color scalar, before we can determine whether or not the input rasters have a valid number of bands. We'll accumulate the scalar sum into scalar and put the rasters on the rasterInputs list. To determine if the input is a scalar or raster, we first try to get the data pointer as a ScalarData type. If the data on the port is a ScalarData or it can be converted to a ScalarData, it will be returned. If it is not a ScalarData and cannot be converted to a ScalarData, we tell Port::GetDataPtr() to just return a NULL. We can then do a similar check to determine if the data on the port is a RasterInfoData, except, since we only handle scalars and rasters, we tell Port::GetDataPtr() to throw and exception, if the data cannot be converted to a RasterInfoData. If we got a ScalarData and it is our first one, we will just remember it. If we already have a scalar sum, we will add the new one to it using AddScalars(), a private method on Add. We'll look at Add::AddScalars() later.

```
ScalarPtr scalar;
std::vector<RasterInfoPtr> rasterInputs;
for (auto it = inputPorts.begin(); it != inputPorts.end(); ++it)
{
    PortPtr port = *it;
    ScalarPtr input = port->GetDataPtr<ScalarData>(Port::ReturnNullIfEmpty);
    if (!input)
    {
        RasterInfoPtr input = port->GetDataPtr<RasterInfoData>(Port::ThrowIfEmpty);
        rasterInputs.push_back(input);
        continue;
    }
    if (!scalar)
        scalar = input;
    else
```

...
scalar = AddScalars(scalar, input);
}

Now we have the scalars summed and all of the rasters on a list. If there are no rasters on the list, all we had was scalars. In this case, we just set the scalar sum on the output port and return. Since we don’t set a RasterInfo on the output port, ProcessBlock() will not be called.

if (rasterInputs.size() == 0)
{
    GetPort(L"OutputRaster")->SetDataPtr<ScalarData>(m_SumOfScalars);
    return;
}

If we have rasters on the list, we have to determine the data type and the number of bands for our output raster. Like in the Overlay example above, we’re going to determine the data type of the output by “promoting” all inputs to the largest data type of the inputs. We’ll start by assuming that the output data type is the lowest possible and then “bump” it up, if we encounter a larger data type. We’ll do something similar to determine the number of bands in our output raster (assuming we output a raster). The number of output bands will depend on the number of bands in the input rasters (if any) and whether or not any of the scalar inputs (if any) are colors. Again we’ll start by assuming that our output raster will be one band, and we’ll bump the count as we go along.

m_DataType = NUMBER_TYPE_U1;
long availableBands = 1;

If we have a scalar sum, we check set the output data type to the data type of the scalar and set the output band count based on whether or not the scalar sum is a color—3 bands if it is; 1 band if it is not. We also save the scalar sum in our private class data, so that we can use it in ProcessBlock().

if (scalar)
{
    m_DataType = scalar->GetDataType();
    availableBands = scalar->IsColor() ? 3 : 1;

    m_SumOf Scalars = scalar;
}

Next we go through the rasters to determine the final data type and number of bands of the output raster. If any of the input raster has an invalid number of bands for what we’re doing, we’ll set an error message on the operator, which will exit this method.

for (auto it = rasterInputs.begin(); it != rasterInputs.end(); ++it)
{
    RasterInfoPtr input = *it;

    if (input->GetData Type() > m_DataType)
    {

if (m_SumOfScalars && m_SumOfScalars->IsColor() && _input->GetDataType() > NUMBER_TYPE_F64)
{
  SetErrorMessage(L"Cannot add a complex raster and color scalar!" );
}

m_DataType = input->GetDataType();

long bands = input->GetBandCount();
bool bandsOkay = (bands == availableBands) || (bands == 1) || (availableBands == 1);
if (!bandsOkay)
  SetErrorMessage(L"Invalid number of bands");

if (bands > availableBands)
  availableBands = bands;

Once we know the data type and number of bands of the output raster, we can create a RasterInfo to set on the output port. GetRasterInfo() is a method on RasterOperator, and it makes a best-guess at what the properties of the output raster will be, based on the input rasters and the working window. We have to adjust the data type and number of bands, though, based on what this operator is doing. We pass RasterInfo::SetBandCount() our raster output port, which will be used to create unique layer names for the model.

RasterInfoPtr rinfo = GetRasterInfo();
if (!rinfo)
  SetErrorMessage(ETXT_TEXT_TR("No raster input found."));
rinfo->SetDataType(m_DataType);
rinfo->SetBandCount(availableBands, GetPort(L"OutputRaster");

We set this RasterInfoData on the output port, so that downstream operators can know about the raster that the operator can produce from its inputs. We may not be asked to produce all available bands or all available tiles. But that will be handled in ProcessBlock().

GetPort(L"OutputRaster")->SetDataPtr<RasterInfoData>(rinfo);
Add::ProcessBlock() Implementation

The ProcessBlock method is where we get the raster tiles and add their pixel values. The method is called to process each tile of a raster, and it is passed a pointer to the port from which the data is being requested and a pointer to a RasterRequest.

```cpp
void Add::ProcessBlock( const PortPtr & port, const RasterRequestPtr & req )
```

First we get a list of the input ports.

```cpp
PortPtrList inputs = GetInputPorts();
```

Then we have to find out how many bands (planes) have been requested. The RasterRequest may not request all planes and they may not be requested in sequential order. Some operators (such as Preview, which is used to view the results of a model the 2D View) allow the user to specify which bands are requested and in which order. ProcessBlock() should honor the request and output a raster with the requested number of bands in the requested order, even if it is possible—from the input data—to generate more.

```cpp
std::vector<unsigned> planes;
req->GetPlanes(planes);
long requestedNumBands = (long)planes.size();
```

Now figure out what is the largest band requested. We’ll need this information later.

```cpp
unsigned maxPlane = 0;
for (auto it = planes.begin(); it != planes.end(); ++it)
{    maxPlane = EMSC_MAX(maxPlane, (*it));
}
```

We are now ready to process each of the input rasters. We declare a Tile for the resultant raster and loop through each of the input ports.

```cpp
TilePtr result;
for (auto it = inputs.begin(); it != inputs.end(); ++it)
```

Like in OnExecute we try to get the data off the port, but this time we want TileData—the actual pixels—instead of the RasterInfoData. In order to get the TileData, we have to tell the parent operator what data we want. We do this by passing a RasterRequest to Port::GetDataPtr(). In the previous example raster operators, we passed Port::GetDataPtr() the same RasterRequest that was passed to us; however, since Add supports input rasters with different numbers of bands, we may have to adjust the requested bands on our RasterRequest. UpdateRasterRequestForInput() is doing that—we’ll look at that later. The list of input ports includes the ports that have scalars set on them, so if we cannot get a raster request specific for this input, we’ll assume that the port has a scalar and skip it, since we processed scalars in OnExecute(). Since we do not want to modify the Tile that we get off of the input port, we will assign it to a ReadonlyTilePtr.
Extend Spatial Modeler

```
RasterRequestPtr inputSpecificReq = UpdateRasterRequestForInput(port, req, maxPlane);
if (!inputSpecificReq)
    continue;

ReadonlyTilePtr tile = port->GetDataPtr<TileData>(inputSpecificReq, Port::ThrowIfEmpty);

Once we have the tile data, we have to make sure that it’s the same data type as the tile we’re outputting. If it is not, we must convert it first. ConvertTile() is a function provided as part of the Tile implementation. Because cloning TileData is expensive, we want to avoid doing so unnecessarily, so since converting made a copy of the tile already, if it’s the first tile and it has the right number of bands, let’s go ahead and set it as our result. This avoids the Clone() later. As in the other example raster operators we have to call Tile::StartEdit() to enable editing on the tile.

if (tile->GetDataType() != m_DataType)
{
    TilePtr converted = ConvertTile(tile, m_DataType);
    tile = converted;

    if (!result && converted->GetNumBands() == requestedNumBands)
    {
        result = converted;
        result->StartEdit();
    }
}

Now we have a tile of the right output data type. If it’s the first tile, we have to copy it to our output. If it already has the right number of bands, we can just clone it. There are several versions of CreateTile() provided as part of the Tile implementation. Once we create the copy, we have to call Tile::StartEdit() to enable editing on the result tile.

result = CreateTile(tile);
result->StartEdit();

If the input tile does not have the right number of bands, we have to create a new output tile with that number of bands and copy the input tile to it. We create the new result tile with another version of CreateTile(). Before we can copy the input tile to the result tile, we have to call Tile::StartEdit() to enable editing on it. We’re taking the easy way out to do the copy by just calling AddTiles(), a method private to this class, to add the input tile to the empty result tile.

result = CreateTile(requestedNumBands, tile->GetWidth(), tile->GetHeight(), m_DataType);
result->StartEdit();
AddTiles(result, tile);
```
If we already have a result tile, we just need to add the input tile to it. We'll examine AddTiles() later.

AddTiles(result, tile);

Once we go through all of the raster inputs and add them together, we need to add to the result tile the sum of the scalars from OnExecute()—if there was one. If the scalar is not the same data type as our output raster, we need to convert it to that type before we can add it to the tile. Note that we could have done this in OnExecute(). AddSumOfScalars() is a private method on Add to add a scalar to a tile.

if (result && m_SumOfScalars)
{
    if (m_SumOfScalars->GetDataType() != m_DataType)
        m_SumOfScalars = ConvertScalar(m_SumOfScalars, m_DataType);

    AddSumOfScalars (result, m_SumOfScalars);
}

When we're finished processing all of the tiles and adding the sum of the scalars, we need to tell the result tile that we are finished editing it. Then we set it on the output port. We have to pass the RasterRequest along.

if (result)
    result->EndEdit();

GetPort(L"OutputRaster")->SetDataPtr<TileData>(req, result);

**Add::UpdateRasterRequestForInput() Implementation**

The purpose of Add::UpdateRasterRequestForInput() is to modify the RasterRequest passed to ProcessBlock() so that it requests from a specific input raster only the band(s) that are available from that raster. The deal here is that if we add a color scalar to a raster that only has one band, we get three bands. Therefore, the Add operator can produce a raster with more bands than the input raster has available. In this case the RasterRequest passed to Add::ProcessBlock() may request Add's bands 2 and/or 3 (band numbers are 1-based), but we only want band 1 from the input raster. Therefore we have to make a copy of the RasterRequest passed to Add::ProcessBlock(), modify the requested planes (bands) and pass this request to Port::GetDataPtr() to get the tile for that input raster.

To update the request, Add::UpdateRasterRequestForInput() needs to know what is available from a particular raster and what the current request is. It also needs to know the maximum band number requested. We pre-calculated this in Add::ProcessBlock() and pass it to Add::UpdateRasterRequestForInput(), so that this method does not have to calculate it each time.

RasterRequestPtr Add::UpdateRasterRequestForInput(const PortPtr & inputPort, const RasterRequestPtr & req, unsigned maxPlane)

So first we get the RasterInfoData from the input port. If we cannot get it, we'll just return an empty RasterRequestPtr, so that ProcessBlock() knows that this port does not contain a raster.
RasterInfoPtr rasterInfo =
  inputPort->GetDataPtr<RasterInfoData>(Port::ReturnNullIfEmpty);
if (!rasterInfo)
  return RasterRequestPtr();

Next we check to see if the input raster has enough bands to fulfill the RasterRequest passed to Add::ProcessBlock(). If so, we can just use it.
if (rasterInfo->GetBandCount() >= maxPlane)
  return req;

If the input raster does not have enough bands, we need to make a copy of the input RasterRequest and change the new RasterRequest to only request plane 1. This modified copy of the input RasterRequest is what we will pass to Port::GetDataPtr() to get the TileData from this input.
RasterRequestPtr rasterReqCopy = req->Clone();
std::vector<unsigned> planes;
planes.push_back(1);
rasterReqCopy->SetPlanes(planes);
return rasterReqCopy;

Add::AddScalars() / AddScalarsT() / AddScalarsComplexT() Implementation

The important take-away information in Add::AddScalars() and the templated functions that it uses is the use of the Scalar object.

- The Scalar class is declared in sbbasiclib/Scalar.h.
- A Scalar can contain a number of different types of data. The enumerated type is DataType in sbbasiclib/DataType.h.
- There are typedefs for each data type (Number_U1, Number_F64, etc.).
- There are a number of functions for creating Scalars.
- There is a function to create a new Scalar by copying an existing Scalar and converting the data type.
- Most of the data types are a single value, but a color scalar contains three values: one each for red, green and blue. Therefore, there are color-specific accessor methods. The accessor methods for the other data types are templated.
- Complex values are stored in a data type with real and imaginary parts.
Add::AddTiles()/AddTilesT()/AddTilesComplexT() Implementation

Similar to Add::AddScalars() the important information to be gleaned from Add::AddTiles() and the templated functions that it uses is the use of the Tile object.

- The Tile class is declared in sbbasiclib/Tile.h.
- A Tile can contain a number of different types of data. The enumerated type is DataTypes in sbbasiclib/DataType.h.
- There are typedefs for each data type (Number_U1, Number_F64, etc.).
- There are functions for creating Tiles.
- There is a function to create a new Tile by copying an existing tile and converting the data type.
- There are templated accessor methods.
- Complex values are stored in a data type with real and imaginary parts.

Unlike a Scalar, however, a Tile has an interesting twist: a NoData mask. Tiles have an unsigned 8-bit (Number_U8) mask that is the same size as the tile. Each of its values indicates whether or not the pixel value at the same location is “data” or “NoData”. An important concept with Spatial Modeler rasters is that anything combined with NoData is NoData. Let’s first look at Add::AddTiles().

In Add::AddTiles() we check the mask on the input results tile (t1). If it does not have any data (it is all NoData), then we can just return, since no matter what the tile we’re adding to it (t2) contains, t1 will remain all NoData.

```c
if (!t1->HasData())
    return;
```

Then we just call our templated function, AddTilesT() with the appropriate data type. So let’s now look at AddTilesT().

In AddTilesT() we go through each pixel in each row of each band and add the pixels from the read-only tile t2 to the results tile t1. That’s the simple plan. We have to account for NoData, though.

First of all, as we go through the rows, we check to see whether or not there’s any data for the band of the input tile. If there isn’t, we get the writable mask for the row of the result tile and set all of the mask values in that row to NoData. We do not need to set the pixel value, since it is masked out.

```c
if (!t2->HasData(b2))
{
    Byte *resultRowMask = t1->GetWritableMaskRow(b, y);
    for (long x = 0; x < t1->GetWidth(); ++x)
```


If the band of the input tile is not all NoData but there is some NoData in either the band of the result tile and/or the band of the input tile, then we have to actually examine each pixel. To do that, we get the mask for the row from each tile. We have to get the writable mask for the row of the result tile, since we may need to change it.

```c
if (t1->HasNoData(b) || t2->HasNoData(b2))
    Byte* resultRowMask = t1->GetWritableMaskRow(b, y);
    Number_U8 const *tileRowMask = t2->GetMaskRow(b2, y);
```

If the pixel in either tile is NoData (the mask value for the pixel is 0), the pixel in the result tile will be NoData, so if the pixel in the result tile already is NoData, we can just leave the mask and the pixel value alone.

```c
if (IS_MASK_VALUE_NODATA(resultRowMask[x]))
{
    continue;
}
```

If the pixel in the result tile is not NoData, but the pixel in the input tile is NoData, then we set the mask value for that pixel to NoData. Again, we do not need to set the pixel value, since it is masked out.

```c
else if (IS_MASK_VALUE_NODATA(tileRowMask[x]))
{
    SET_MASK_VALUE_NODATA(resultRowMask[x]);
}
```

If, however, the pixels in both tiles are data, then we will have a valid data value in the result tile, so we just add the pixel value of the input tile to the pixel value in the result tile.

```c
resultRow[x] += tileRow[x];
```

If neither the band of the result tile nor the band of the input tile has NoData, then we can be more efficient. We do not need to examine the mask values at all. We can just loop through the pixels in the row and add the pixel value from the input tile to the pixel value of the result tile.

```c
for (long x = 0; x < t1->GetWidth(); ++x)
{
    resultRow[x] += tileRow[x];
}
```
Extend Spatial Modeler

Add::AddSumOfScalars() / AddSumOfScalarsT() / AddSumOfScalarsComplexT() Implementation

There really isn’t much of note in the implementation of Add::AddSumOfScalars(), AddSumOfScalarsT or AddSumOfScalarsComplexT that we haven’t already covered in Add::AddScalars() and Add::AddTile().

Create a New Data Type

Data types are used in Spatial Modeler to pass data from one operator to another. The built-in data types can be found in the API documentation. It is recommended that you use the built-in, simple data types when possible, so that your operators can be chained with other, built-in Spatial Modeler operators; however, in creating custom operators you may find it necessary to pass complex data structures between operators. To do so you can create custom data types.

Example 7: Classification Signature

This example is located in sm_SignatureExample. It creates a plug-in DLL that extends Spatial Modeler for performing some image classification functions. For that it implements two operators and a data type. The example model that uses these components to perform a simple Mahalanobis distance classification is mahalanobis-from-signature.gmdx.

Dissect Example 7

SignatureListData class

The SignatureListData class is the class that implements a new data type. The code for the SignatureListData class can be found in SignatureList.h and SignatureList.cpp. A data class is basically a wrapper around a class containing the actual data. In that way, you can use any class you wish to actually hold your data. This may be an existing class or a new class you create just for Spatial Modeler. The only constraint is that it is must be copyable—it must implement either implicitly or explicitly a valid copy constructor. The wrapper class must derive from HexGeo::SpatialModeler::Data, but we have provided a templated class, HexGeo::SpatialModeler::DataT, that implements many of the required methods for you. Your data class should derive from DataT, not Data directly.

The SignatureListData class wraps the SignatureList class that’s declared at the top of SignatureList.h. It itself is not of particular interest to this example. The one interesting piece is the typedef of a shared pointer of this class. See the Best Practices section of this document for more information about the use of boost::shared_ptr in Spatial Modeler.

typedef boost::shared_ptr<SignatureList> SignatureListPtr;
Now let's look at the declaration of SignatureListData further down the file. As indicated above, SignatureListData derives from the templated DataT class. The template parameter is the class that actually holds the data—whatever class you want that to be.

```cpp
class SignatureListData : public DataT<SignatureList> {
    SLIB_API SignatureListData();
    SLIB_API virtual ~SignatureListData();
};
```

As with operators, there's a macro for setting the namespace and name of the data type. The first parameter to the SM_PLUGIN_DATA macro, L"Example", is the namespace for the data. This is not necessarily the same as the C++ namespace. The namespace is intended, as in C++, to further define the data type name and reduce the risk of duplication. It is used when instantiating the data type using DataFactory::Create(). The namespace should not contain spaces. The data type(s) you write should not use "Example" as the namespace. L"SignatureList" is the name of the data type. The name is used when registering the data type—we'll discuss this below—and when instantiating the operator using DataFactory::Create(). The data type name does not have to match the name of the class that the data type wraps; however, it should not contain spaces.

```cpp
SM_PLUGIN_DATA(L"Example",L"SignatureList");
```

Just as for operator plug-ins, the Provider() method should be overridden by 3rd-party developers. This method is used to specify the company or organization that developed the plug-in. Although it is not visible in the Spatial Model Editor, this attribute can be queried to who developed the plug-in.

```cpp
virtual std::wstring Provider();
```

There are a number of Data methods that your data class may need to override. These have to do with the serialization of your data and what the Spatial Modeler Editor displays—in various places—for your data. The most basic methods are ToString(), FromString(), ToJSON(), FromJSON(), and ToShortString. We'll discuss these when we go through the implementations.

```cpp
SLIB_API virtual std::wstring ToString();
SLIB_API virtual bool FromString( const std::wstring & string );
SLIB_API virtual bool ToJSON(ERDAS_JSONLIB_NAMESPACE::Value & jsonValue);
SLIB_API virtual bool FromJSON(const ERDAS_JSONLIB_NAMESPACE::Value & jsonValue);
SLIB_API virtual std::wstring ToShortString();
```

Some additional methods not shown in this example are ToSerializedString(), FromSerializedString() and ToReportString(). ToSerializedString() and FromSerializedString() are used when the information necessary to completely reconstruct the in-memory version of your data cannot be represented in a user-friendly string. The default implementations of ToSerializedString() and FromSerializedString() are to call ToString() and FromString(). ToReportString() is used if you want your data to be formatted nicely
(on multiple lines, etc.) when output in a report. See the API help for more information about the uses of each of these methods, as well as others you may override.

All data types should override the Clone() method.

```
SLIB_API Ptr Clone();
```

**SignatureListData::ToString() and SignatureListData::FromString() Implementations**

ToString() is the method called to display the data on the port in the Properties panel of the Spatial Modeler Editor. It is also called by the default implementations of ToSerializedString() and ToReportString(), so it is, by default, the main method used for representing your data to the end user. FromString() is used to set the data on your data type from textual user input, such as entering values in the Properties panel of the Spatial Modeler Editor or setting values on ports from the command line. The output of ToString() should be directly usable by FromString() to reconstruct your data. If your data is too complex for the user to enter it textually, you do not need to implement FromString(); however, it will not be possible to set the value on a port on an operator to your data type. For this reason, the output of ToString() and the expected input of FromString() should be user-friendly.

Looking at the implementation of SignatureListData::ToString() in SignatureList.cpp, we see that it uses the member variable m_ActualData. This variable is defined in the DataT template, and it is a shared pointer to our data class—the one we passed as a parameter to the DataT template when we declared our data type class. Therefore, we can access the public members of our data class. In this example, our data class, SignatureList, holds in memory data that is read from a file, and all that is needed to construct the data is a file name. Consequently, SignatureListData::ToString() just returns the name of the file.

```
std::wstring SignatureListData::ToString()
{
    if (!m_ActualData)
        return L"null";
    return m_ActualData->GetFileName();
}
```

SignatureListData::FromString() does the opposite. It takes the input string and reads the data from the file into the SignatureList (m_ActualData). FromString() returns a boolean, which indicates whether or not the method was able to convert from the string to the internal data.

```
bool SignatureListData::FromString(const std::wstring &str)
{
    if (!m_ActualData)
    {
        m_ActualData.reset (new SignatureList());
    }
    return m_ActualData->ReadSignatureFile (str);
```
SignatureListData::ToJSON() and SignatureListData::FromJSON() Implementations

ToJSON() and FromJSON() are used when serializing or de-serializing your data in a model and in other persistence workflows, including models used in Hexagon Smart M.App. If your data type is to be used in a model in the Smart M.App environment, the JSON format of your data type needs to be documented so users of the services API can create and use the data type. In our example here, we are simply persisting a JSON string; however, your data type will be more complex and may require a more complex JSON representation. See the time_tData class in Example 8 for an example of converting a data type to a more complex JSON representation. These two methods depend on jsonlib, which is part of the Spatial Modeler SDK but not covered in this document.

```cpp
bool SignatureListData::ToJSON(ERDAS_JSONLIB_NAMESPACE::Value & jsonValue)
{
    if (m_ActualData)
    {
        jsonValue = m_ActualData->GetFileName();
        return true;
    }
    return false;
}
```

```cpp
bool SignatureListData::FromJSON(const ERDAS_JSONLIB_NAMESPACE::Value & jsonValue)
{
    if (!m_ActualData)
        return false;
    m_ActualData->ReadSignatureFile(jsonValue.asString());
    return true;
}
```

SignatureListData::ToShortString() Implementation

ToShortString() is the method called to display the data on the port of the operator in the Spatial Modeler Editor canvas. This string should be rather short—probably no longer than a typical GUI menu item. In this implementation SignatureListData::ToShortString() returns the name of the signature file, but this time without the path.

```cpp
std::wstring SignatureListData::ToShortString()
{
    if (!m_ActualData)
        return L"null";
```
return m_ActualData->GetSimpleFileName();
}

**SignatureSource operator class**

The SignatureSource operator creates the SignatureListData by reading a signature list from an ERDAS IMAGINE signature file. The code for the operator can be found in SignatureSource.h and SignatureSource.cpp. The definition of the class, in SignatureSource.h, is very similar to those of operators we’ve created earlier in this Guide.

```cpp
class SignatureSource : public HexGeo::SpatialModeler::Operator {
public:
    SM_PLUGIN_OBJECT(L"Example", L"SignatureFileInput");
    OPERATOR_DISPLAY_INFO( ETXT_TEXT_TR("Example"), ETXT_TEXT_TR("Signature File Input"), L"classifier32x32.ico" );
    OPERATOR_DESCRIPTION(ETXT_TEXT_TR("Input operator for signature files."));

    SignatureSource();
    ~SignatureSource();

    void Init();
    void OnExecute();
};
```

The slight difference is that this operator specifies an icon to be used in the Spatial Modeler Editor: "classifier32x32.ico". We’ll not go into any of the rest of this declaration here.

**SignatureSource::Init() Implementation**

The SignatureSource class is implemented in SignatureSource.cpp. Let’s look at the implementation of SignatureSource::Init(). Again, it is very similar to the other operators we have created in this Guide. It creates a single input port and a single output port. The input port is a file name. Of note is the data type for the output port: SignatureListData. This is our new data type.

```cpp
CreatePort<SignatureListData>( L"SignatureList", SB_PORT_OUTPUT );
PortPtr fileNamePort = CreatePortWithAttributes(L"Filename",
    FileData::GetDataType(), SB_PORT_INPUT,
    L"FileFilter", Data::Create<StringData>(L"Signature (*.sig)"),
    L"Extension", Data::Create<StringData>(L"*.sig"),
    NULL );
```
SignatureSource::OnExecute() Implementation

As you should expect, having seen the previous examples in this Guide, the SignatureSource::OnExecute method first gets the name of the file from the input Filename port.

```cpp
std::wstring filename = GetPort(L"Filename") -> GetDataValue<FileData>({L""});
if (filename.empty())
    SetErrorMessage ( L"No file name specified." );
```

SignatureSource::OnExecute() then instantiates a SignatureListPtr from a new SignatureList, constructed by reading the data from the input file. It then sets this SignatureListPtr as the data pointer on the output port. Setting the data on the port in this manner—with the SignatureListPtr—prevents making unnecessary copies of the data.

```cpp
SignatureListPtr sigListPtr (new SignatureList(filename));
GetPort(L"SignatureList") -> SetDataPtr<SignatureListData>(sigListPtr);
```

**GetSignatureParameters operator class**

The GetSignatureParameters operator takes a SignatureListData and an index to one of the signatures in the list and converts the signature into a table of means per band and a covariance matrix of the covariances of each pair of bands, inverted. The code for the operator can be found in GetSignatureParameters.h and GetSignatureParameters.cpp. The definition of the class, in GetSignatureParameters.h, is the same as the definition of SignatureSource.

```cpp
class GetSignatureParameters : public HexGeo::SpatialModeler::Operator
{
public:
    SM_PLUGIN_OBJECT( L"Example", L"GetSignatureParameters" );
    OPERATOR_DISPLAY_INFO( ETXT_TEXT_TR("Example"),
        ETXT_TEXT_TR("Get Signature Parameters"), L"classifier32x32.ico" );
    OPERATOR_DESCRIPTION(ETXT_TEXT_TR("Returns the parameters of the input signature."));

    GetSignatureParameters();
    ~GetSignatureParameters();

    void Init();
    void OnExecute();
};
```

**GetSignatureParameters::Init() Implementation**

The GetSignatureParameters class is implemented in GetSignatureParameters.cpp. Looking at the GetSignatureParameters::Init method, we see that it creates two input ports and two output ports.

```cpp
CreatePort<SignatureListData>({ ETXT_TEXT_KEY("SignatureList"), SB_PORT_INPUT });
```
CreatePort<UnsignedData>( ETXT_TEXT_KEY("SignatureNumber"), SB_PORT_INPUT );
CreatePort<TableData>( ETXT_TEXT_KEY("MeansTable"), SB_PORT_OUTPUT );
CreatePort<MatrixData>( ETXT_TEXT_KEY("InverseCovMatrix"), SB_PORT_OUTPUT );

The input ports are the SignatureList of type SignatureListData and a SignatureNumber of type UnsignedData. We can use UnsignedData here, because this is a 0-based index into the signature list, so it must be a non-negative number. The two outputs are a table and a matrix. Spatial Modeler has built-in types for these: TableData and MatrixData.

GetSignatureParameters::OnExecute() Implementation

The GetSignatureParameters::OnExecute method, also implemented in GetSignatureParameters.cpp, starts by getting the value off of the input ports. If there is no data on the SignatureList port, we cannot continue, so we tell GetDataPtr() to throw an exception if the port is empty. We also check that the range is valid for the SignatureNumber and set an error message on the operator, if it is not.

SignatureListPtr sigListPtr =
GetPort(L"SignatureList") ->GetDataPtr<SignatureListData>(Port::ThrowIfEmpty);

unsigned long signum = (unsigned long)GetPort(L"SignatureNumber") ->GetDataValue<UnsignedData>();
if (signum >= sigListPtr ->GetCount())
    SetErrorMessage (L"Signature Number out of range");

Next we call methods that we created on the SignatureList to get the information for the requested signature. The important information shown here is how to create and populate Table and Matrix data types. Like the Tile and Scalar data types used in some of the examples above, Table and Matrix are abstract base classes—you cannot instantiate them. There are functions provided to create them: CreateTable() and CreateColorTable() (not shown) for tables and CreateMatrix() for matrices, and there are methods on the base classes for setting and getting values.

In the code below we create a Table with the number of rows that are in our means table. The data type is 64-bit floating point. CreateTable() returns a TablePtr. We then use the templated SetValue() method on Table to copy the values from the vector to the Table.

std::vector<Number_F64> means = sigListPtr ->GetMeansTable (signum);
TablePtr meansTable = CreateTable (means.size(), NUMBER_TYPE_F64);
for (int i = 0; i < means.size(); i++)
    meansTable ->SetValue<Number_F64> (i, means[i]);

Working with the Matrix data type is similar. We call CreateMatrix() and tell it the number of rows, number of columns and data type for the Matrix. We then loop through the two dimensions of the inverted covariance matrix to set the values in the Matrix.

std::vector<std::vector<Number_F64>> invertedCovariance =
sigListPtr ->GetInvertedCovariance (signum);
MatrixPtr mtx = CreateMatrix (invertedCovariance.size(),
invertedCovariance[0].size(), NUMBER_TYPE_F64);
for (int row = 0; row < invertedCovariance.size(); row++)
{
    for (int col = 0; col < invertedCovariance[row].size(); col++)
    {
        mtx->SetValue<Number_F64> (row, col, invertedCovariance[row][col]);
    }
}

Once we have the values in the Table and Matrix, we set them as the data on the output ports.
GetPort(L"MeansTable")->SetDataPtr<TableData>(meansTable);
GetPort(L"InverseCovMatrix")->SetDataPtr<MatrixData>(mtx);

Register the plug-in

As in our previous examples, we have to register the plug-in objects implemented in this DLL. You've seen the registration for operators in the previous examples.
if ( type == L"Operator" || type == L"" )
{
    pluginList.push_back( SM_PLUGIN_REGISTRATION ( sm_SignatureExample::GetSignatureParameters ));
    pluginList.push_back( SM_PLUGIN_REGISTRATION ( sm_SignatureExample::SignatureSource ));
}

Registering a data type is similar. In this case we need to return the data type when the requested plug-in type is L"Data". For each object we use the macro SM_PLUGIN_REGISTRATION to create an AvailablePluginProxy. The single argument to the macro is the class that implements the operator. The return value of the macro gets pushed on to the list of plug-ins.
if( type == L"Data" || type == L"" )
{
    pluginList.push_back( SM_PLUGIN_REGISTRATION ( sm_SignatureExample::SignatureListData ));
}
Create a Data Conversion

You can create new data conversion to convert between custom data types and standard Spatial Modeler data types. These conversions can be used in your own tools or within Spatial Modeler. The following example will show you how to create a simple data conversion.

Example 8: time_t to DateTime

This example is located in sm_ConversionExample. It creates a plug-in DLL containing a data type, a data conversion and a testing operator. The data type wraps the standard C++ time_t data type. The conversion converts a Spatial Modeler DateTime data type to a time_t data type. The test operator applies the conversion.

Dissect Example 8

The time_tConversion class

The code for the conversion can be found in time_tConversion.h and time_tConversion.cpp. Consider the definition of the time_tConversion class in time_tConversion.h. A conversion in Spatial Modeler must be a class that derives from Conversion in the HexGeo::SpatialModeler namespace.

```cpp
class time_tConversion : public HexGeo::SpatialModeler::Conversion
{
    time_tConversion();
    virtual ~time_tConversion();
}
```

There are a few Conversion methods that all sub-classed conversions must implement. We have macros to help with some of them.

Use the SM_PLUGIN_CONVERSION macro to set the name and data types of the conversion. The first two parameters, L"IMAGINE" and L"DateTime", specify the namespace and name of the source data type (the data type to convert from). The third and fourth parameters, L"Example" and L"time_t", specify the namespace and name of the destination data type (the data type to convert to).

```cpp
SM_PLUGIN_CONVERSION(L"IMAGINE",L"DateTime", L"Example",L"time_t");
```

Just as for operator plug-ins, the Provider() method should be overridden by 3rd-party developers. This method is used to specify the company or organization that developed the plug-in. Although it is not visible in the Spatial Model Editor, this attribute can be queried to who developed the plug-in.

```cpp
virtual std::wstring Provider();
```
A conversion plug-in must also implement a Convert() method. This method is called by Spatial Modeler to convert between the data types specified in SBLIB_PLUGIN_CONVERSION. We'll discuss this method more when we examine time_tConversion.cpp.

```cpp
HexGeo::SpatialModeler::DataPtr Convert(HexGeo::SpatialModeler::DataPtr from);
```

time_tConversion:Convert() Implementation

Now let's look at how Convert is implemented in time_tConversion.cpp. First, we will extract the date and time information from the source DataPtr. We cast the DataPtr to a DateTimeData shared pointer. We can assume that the DataPtr Spatial Modeler passes to the Convert function will contain a DateTime value. Once we have a DateTimeData shared pointer, we can dereference it to extract the data and time information.

```cpp
boost::shared_ptr<DateTimeData> sourceDateTime = from->CastPtr<DateTimeData>();
DateTimeData::ActualType value = sourceDateTime->GetDataValue();
```

Next, we will use the DataFactory to create a new time_tData object and store the reference to that object in a shared_pointer. We cast the result of DataFactory::CreateDirect to a time_tData since CreateDirect returns a generic DataPtr.

```cpp
boost::shared_ptr<time_tData> result = DataFactory::CreateDirect<time_tData>()->CastPtr<time_tData>();
```

Then we will extract the date and time values from the DateTime data structure into a struct tm.

```cpp
value.GetTime(tmTime.tm_year, tmTime.tm_mon, tmTime.tm_mday, tmTime.tm_hour,
              tmTime.tm_min, tmTime.tm_sec);
```

We need to alter the data from the DateTime to match the conventions used in struct tm. We will create a time_t from the struct tm.

```cpp
// struct tm::tm_year expects year from 1900.
tmTime.tm_year -= 1900;
// struct tm::tm_mon is zero based.
tmTime.tm_mon -= 1;
// DateTime doesn't store DST information, so set it to unknown.
tmTime.tm_isdst = -1;
// Create a time_t from the struct tm.
theTime = mktime(&tmTime);
```

Finally, we assign the newly created time_t value to the time_tData we created earlier and return the converted data to Spatial Modeler.

```cpp
result->SetDataValue(theTime);
return result;
```
time_tData data class

This class is a data type to wrap the standard C++ time_t data type. It can be found in time_tData.h and time_tData.cpp. It implements only the most basic data type features and will highlight the implementations of the ToJSON() and FromJSON methods here. To learn more about creating a custom data type, see Creating a new data type.

```cpp
bool time_tData::ToJSON(ERDAS_JSONLIB_NAMESPACE::Value & jsonValue)
{
    if (!m_ActualData)
    {
        return false;
    }

    time_t theTime = *(m_ActualData.get());
    struct tm tmTime;
    gmtime_s(&tmTime, &theTime);

    jsonValue[L"Month"] = tmTime.tm_mon + 1;
    jsonValue[L"Day"] = tmTime.tm_mday;
    jsonValue[L"Year"] = tmTime.tm_year + 1900;
    jsonValue[L"Hour"] = tmTime.tm_hour;
    jsonValue[L"Minute"] = tmTime.tm_min;
    jsonValue[L"Second"] = tmTime.tm_sec;
    return true;
}

bool time_tData::FromJSON(const ERDAS_JSONLIB_NAMESPACE::Value & jsonValue)
{
    struct tm tmTime;
    tmTime.tm_mon = jsonValue[L"Month"].asInt() - 1;
    tmTime.tm_mday = jsonValue[L"Day"].asInt();
    tmTime.tm_year = jsonValue[L"Year"].asInt() - 1900;
    tmTime.tm_hour = jsonValue[L"Hour"].asInt();
    tmTime.tm_min = jsonValue[L"Minute"].asInt();
    tmTime.tm_sec = jsonValue[L"Second"].asInt();

    SetDataValue(mktime(&tmTime));
    return true;
}
```
time_tTestOperator operator class

This class is an operator to test the time_tConversion data conversion. It can be found in time_tTestOperator.h and time_tTestOperator.cpp. We will only discuss the implementation of the OnExecute function where the conversion is applied. To get more details on creating an operator, see Creating an Operator.

time_tTestOperator::OnExecute implementation

In this function, we will get the input data from the port named “DateTime.” The port defines its data type as DateTimeData, but we will request the data as a time_tData. This causes Spatial Modeler to search for an appropriate data conversion to convert from the port’s data type to the requested data type. In this case, the time_tConversion data conversion will be used.

DataPtr convertedData = GetPort(L"DateTime")->GetData<time_tData>();

We will then set the newly converted data on the output port named “time_t” for use by the next operator in the chain.

GetPort(L"timeT")->SetData(convertedData);

Register the plug-in

As in our previous examples, we have to register the plug-in objects implemented in this DLL. In this case we need to return the conversion when the requested plug-in type is L"Conversion". For each object we use the macro SM_PLUGIN_REGISTRATION to create an AvailablePluginProxy. The single argument to the macro is the class that implements the operator. The return value of the macro gets pushed on to the list of plug-ins.

if ( type == L"Operator" || type == L"" )
{
    pluginList.push_back ( SM_PLUGIN_REGISTRATION ( sm_ConversionExample::time_tTestOperator ) );
}
if ( type == L"Conversion" || type == L"" )
{
    pluginList.push_back ( SM_PLUGIN_REGISTRATION ( sm_ConversionExample::time_tConversion ) );
}
if (type == L"Data" || type == L"")
{
    pluginList.push_back( SM_PLUGIN_REGISTRATION ( sm_ConversionExample::time_tData ) );
}
Creating a Customization Dialog

Spatial Modeler provides dialogs for many operators, ports, and data types. You can create custom interfaces of your own to associate with your operators, the ports of your operators, or your data types. Custom interfaces in Spatial Modeler are called UI Providers. UI Providers are implemented using a similar plug-in mechanism to the plug-ins for Operators. The next example demonstrates how to create a UI Provider for an operator.

Example 9: Color Picker

This example is located in sm_UIProviderExample. It creates a plug-in DLL that extends Spatial Modeler by adding an operator called Choose Color for Color Scalar input, and a UI Provider that provides a dialog for selecting a color. Most of the user interface for the core Spatial Modeler is written in ERDAS IMAGINE’s native user interface language EML (ERDAS Macro Language). For this example we will instead use a dialog provided by MFC (Microsoft Foundation Class library), the CColorDialog class, which implements a standard Windows color picker.

The CColorDialog class returns the color as the data type COLORREF, which is defined in the include file windows.h. The operator implemented here converts the COLORREF to a Color Scalar, which is a standard Spatial Modeler data type.

Visual Studio settings for this project

Because this project uses both Spatial Modeler libraries and MFC libraries, it requires different Visual Studio settings than the other examples in this document. Any UI Provider you create using MFC libraries will likely need these same settings. (Settings shown are for Visual Studio 2013)

- In Solution Explorer, right click on the project name (in this case, sm_UIProviderExample).
- In Configuration Properties - General, under Project Defaults
- Use of MFC should be set to “Use MFC in a Shared DLL”
- Expand Linker
- Click on Input
- Ignore All Default Libraries should be set to “No”
- Ignore Specific Default Libraries should be set to “%(IgnoreSpecificDefaultLibraries)”
Dissect Example 9

ChooseColor operator class

The ChooseColor operator takes the COLORREF data returned from the CColorDialog and converts it to Color Scalar data. ChooseColor is similar to the operators in the Input category of Spatial Modeler, in that the operator is intended to be at the beginning of a chain of operators. The code for the operator can be found in ChooseColor.h and ChooseColor.cpp. The definition of the class, in ChooseColor.h, is very similar to those of operators we’ve created earlier in this Guide.

```cpp
class CChooseColor : public HexGeo::SpatialModeler::Operator {
public:
    SLIB_PLUGIN_OBJECT ( L"Example", L"ChooseColor" );
    OPERATOR_DISPLAY_INFO ( ETXT_TEXT_TR ( "Example" ),
        ETXT_TEXT_TR ( "Choose Color" ), L"ElevationColorMinMax.ico" );
    OPERATOR_DESCRIPTION ( ETXT_TEXT_TR("Return a color selected through an UI provider." ) );

    CChooseColor();
    virtual ~CChooseColor();

    void Init();
    void OnExecute();
};
```

ChooseColor::Init() Implementation

The ChooseColor class is implemented in in ChooseColor.cpp. Let’s look at the implementation of ChooseColor::Init(). It is somewhat similar to the other operators we have created in this Guide, in that it has a single input port and a single output port. The difference is that we hide the input port, since this operator will typically be placed at the start of a chain of operators.

The input port (named Value) is used for the COLORREF returned by the CColorDialog. COLORREF is a 32 bit unsigned integer, so we can use Spatial Modeler’s UnsignedData type for this port. We hide the port by passing the bool true to the SetIsHidden method on the port.

```cpp
CreatePort<UnsignedData>(ETXT_TEXT_KEY("Value"),
    SB_PORT_INPUT)->SetIsHidden(true);
```

The output port is named Color and uses Spatial Modeler’s ColorData type.

```cpp
CreatePort<ColorData>(ETXT_TEXT_KEY("Color"), SB_PORT_OUTPUT);
```

The other task we need to do in the Init() method is to set our custom user interface for this operator. For this we use SetUIProvider, and pass it the namespace and name of the UI Provider. For this example we
use the namespace “Example” and the name “ColorDialog”. These must match the namespace and name we use in the SBLIB_PLUGIN_OBJECT macro in the definition of the UI Provider.

SetUIProvider(L"Example", L"ColorDialog");

Now when we double-click the Choose Color operator, it will look for the plug-in UI Provider with this namespace and name, and present the user interface defined in that UI Provider.

**ChooseColor::OnExecute() Implementation**

The ChooseColor::OnExecute method gets the COLORREF from the input port, converts it to ColorData, and sets the ColorData on the output port.

First we get the COLORREF as UnsignedData form the input Value port.

COLORREF colorRef = (COLORREF) GetPort(L"Value") -> GetDataValue<UnsignedData>();

Next we convert from the input type to the output type. The Spatial Modeler Color type stores red, green and blue values as doubles, typically ranging from 0.0 to 1.0.

double red = (GetRValue(colorRef)) / 255.;
double green = (GetGValue(colorRef)) / 255.;
double blue = (GetBValue(colorRef)) / 255.;

Color color (red, green, blue);

Finally we set the data on the output port.

GetPort(L" Color") -> SetDataValue<ColorData>(color);

**ColorDialogProvider class**

The ColorDialogProvider class implements the custom user interface. This class has the method that displays the CCColorDialog and gets the returned color from it. The class implements a plug-in in a somewhat similar fashion to the operator plug-ins in earlier examples, but this is a UIProvider plug-in rather than an Operator plug-in. The code for ColorDialogProvider is in the files ColorDialogProvider.h and ColorDialogProvider.cpp.

ColorDialogProvider.h has the definition of the class. It derives from the class HexGeo::SpatialModeler::UIProvider.

```cpp
class ColorDialogProvider : public UIProvider
```

Similarly to the operator plug-ins defined in earlier examples, the SM_PLUGIN_OBJECT macro is used to set the namespace, type and name of the plug-in. Notice that the namespace “Example” and the name “ColorDialog” must match the parameters for SetUIProvider in the ChooseColor operator in order for that operator to be able to use this UI Provider.

```cpp
SM_PLUGIN_OBJECT(L"Example", L"ColorDialog");
```
Just as for operator plug-ins, the Provider() method should be overridden by 3rd-party developers. This method is used to specify the company or organization that developed the plug-in. Although it is not visible in the Spatial Model Editor, this attribute can be queried to who developed the plug-in.

```cpp
virtual std::wstring Provider();
```

Each type of UIProvider implements a ShowDialog method which implements the display of the custom user interface. The ShowDialog method for an operator has a pointer to an Operator as its parameter list. The ShowDialog method returns a bool indicating whether the user OKed the information in the user interface or cancelled it.

```cpp
bool ShowDialog( const OperatorPtr & op );
```

More than one operator could use this custom interface. Any operator that will use this UI should call SetUIProvider with the namespace and name of this UIProvider. Double-clicking the operator from the Spatial Model Editor will cause this ShowDialog method to be called, passing it a pointer to the operator.

The ShowDialog method is implemented in ColorDialogProvider.cpp.

The bool variable changed indicates whether the user has clicked OK in the CColorDialog. It is initialized to false.

```cpp
bool changed = false;
```

The COLORREF variable color will used for the COLORREF returned from the CColorDialog. It is initialized to 0, which represents black (0x00000000).

```cpp
COLORREF color(0);
```

Next we get the input port from the operator which triggered this UI Provider. Any operator which uses this UI Provider should have an input port named “Value” which can take data of type UnsignedData.

```cpp
PortPtr valuePort = op->GetPort(L"Value");
```

A UI Provider should only set values for input ports of an operator. It should not set the value of any output port of an operator. It is the job of the operator’s OnExecute or ProcessBlock methods to set the values for the output ports.

Now we check to see if the Value port already has data set on it. We pass Port::ReturnNullIfEmpty to indicate that GetDataPtr should return NULL if there is no data already set on the port. If NULL is returned, the COLORREF variable color retains its initial value of black.

```cpp
if (valuePort->GetDataPtr<UnsignedData>(Port::ReturnNullIfEmpty))
```

If the Value port did have data, we get the UnsignedData value from the port and set the COLORREF variable color to that value.

```cpp
color = (COLORREF) valuePort->GetDataValue<UnsignedData>();
```

If we create a model with this operator, and use this UI Provider to choose a color, and then save the model, the color that was picked will be saved in the model as data for the Value port. When we later
reopen the model, this code allows us to retrieve the color saved with the Value port so that we can set
the initial color for the CColorDialog to that color.

Next we call the constructor for the CColorDialog, passing it the COLORREF color, which is either the
previously set color or the initial color black. The CColorDialog uses this color as its initial color.

CColorDialog dlg(color);

Now we display the CColorDialog and wait for the user to click either OK or Cancel. If they clicked Cancel,
we skip the following block of code.

if (dlg.DoModal() == IDOK)

If the user clicked OK, we get the color they chose.

color = dlg.GetColor();

Then we set that color as the data value for the Value port of the operator.

valuePort->SetDataValue<UnsignedData>(color);

The last thing we do in this block of code is set the changed value to true, indicating that the user clicked
OK, and so the data value for the Value port of the operator has changed.

changed = true;

Lastly, ShowDialog returns the changed flag, indicating to Spatial Modeler whether or not changes have
been made to the operator’s port values.

return changed;

Register the plug-ins

As in our previous examples, we have to register the plug-in objects implemented in this DLL. You’ve
seen the registration for operators in the previous examples.

if ( type == L"Operator" || type == L"")
{
    pluginList.push_back (SM_PLUGIN_REGISTRATION (sm_UIProviderExample::CChooseColor));
}

Registering a UI Provider is similar. In this case we need to return when the requested plug-in type is
L"UIProvider". For each object we use the macro SM_PLUGIN_REGISTRATION to create an
AvailablePluginProxy. The single argument to the macro is the class that implements the operator. The
return value of the macro gets pushed on to the list of plug-ins.

if( type == L" UIProvider" || type == L"")
{
    pluginList.push_back (SM_PLUGIN_REGISTRATION (sm_UIProviderExample::ColorDialogProvider));
SECTION 7

Additional Information

Best Practices

**Operator and Port Naming**

The recommended naming convention for operators and ports is that the name should not contain spaces, but display names can contain spaces. The name should be derivable from the display name by removing the spaces. So the IsNoData operator can have a display name of "Is No Data", but its name should be "IsNoData". This is requested to simplify operator and port usage in Python. It is fairly straightforward to guess how to access a particular operator or port in Python with this convention.

**Port Inputs**

If you are creating a model that is to be saved and then run with different inputs, it is recommended that you use Port Inputs to "parameterize" your model. Port Inputs allow you clearly identify the inputs to the model, to provide meaningful names and descriptions for the model’s parameters, to explicitly specify the data type(s) expected for a parameter, and to stipulate whether or not it is required for the parameter to have a value for the model to run successfully. When a model is run from the Spatial Model Editor, you will be automatically prompted to enter values for Port Inputs.

**boost::shared_ptr use**

Within Spatial Modeler, you’ll notice we use boost::shared_ptr to hold our pointers. This is a wrapper class that keeps track on how many pointers are used, and frees the pointer once the last instance is no longer held. This makes memory leaks much less frequent, so we suggest SDK users do the same. For more information about boost::shared_ptr, see http://www.boost.org.

**Data Access**

There are many different ways to access data from a port, but each has its intended use. GetData/SetData functions are used when you aren’t going to access anything from the data. This is useful when you just need to pass data through an operator. GetDataValue/SetDataValue are used for integral data types, such as integers and double. These allow you to work in the true data type without worrying about the shared pointer we use to store the data. GetDataPtr/SetDataPtr are used to access the shared pointer to the data. This is recommended when the data being accessed is a class.

**Models and Submodels**

Model is a particular class of note located in the HexGeo::SpatialModeler namespace. Models are used to store operators and execute the processing chain. Model also derives from Operator, so it can be used as an operator within the chain. When used in this context it is referred to as a Sub-model. These were originally referred to as Process and Sub-process, so you may see some references to those throughout the API documentation.
Additional operators can be added to a model through the AddOperator function. You need to define which operator is at the end of the chain (called the tail), so the model knows which operator to pull. This is done with the AddTail function.

Models are used to serialize as well. The ModelFormatFactory takes in a Model, and has functions to open the model from a file, and save a model to a file.

When using a sub-model, you need to define which ports should be exposed on the model. This is done through aliased ports.

**Aliased Ports**

Aliasing a port will create a new port on the model, which is connected to a port on an operator the model contains. The port created on the model is referred to as the “outer aliased port”, while the port on the operator the model owns is referred to as the “inner aliased port”. Ports can be aliased and unaliased through the AliasPort(s) & UnaliasPort(s) functions on Model. Aliased ports are seen in the Spatial Model Editor as Port Input and Port Output.

**Attributes**

Attributes are the preferred means of saving private information that needs to be serialized along with the operator. Attributes are not exposed to the user, but are serialized. An attribute can be used through the GetAttribute and SetAttribute functions located on Operator. Attributes are stored as Data, so it can contain any type that spatial modeler supports, and can be expanding by adding additional data types which was outlined in a previous section.

**Creating Visual Studio Projects for Spatial Modeler Plug-ins**

When creating your own Spatial Modeler plug-ins, you will need to ensure that your Visual Studio project is set up correctly with regard to compile and link settings and to include all of the Spatial Modeler libraries and include files. You can accomplish this by either copying and modifying one of the example projects or by creating a new project from scratch and applying the needed settings yourself.

**Copy and Modify an Example Project**

Start by choosing an example project to modify. For this example, we will use sm_ExternalProcessExample as our starting point. Select the sm_ExternalProcessExample directory and copy it. Rename the directory as appropriate for your plug-in. For this example, we will name our new directory sm_MyNewPlugin.
Open your new directory and rename the .props, .vcxproj and .vcxproj.filters files to match the directory name. Delete all CPP and H files aside from resource.h. Your directory should look like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource.h</td>
<td>5/23/2014 11:54 AM</td>
<td>C/C++ Header</td>
</tr>
<tr>
<td>sm_MyNewPlugin.props</td>
<td>5/23/2014 11:54 AM</td>
<td>Project Property File</td>
</tr>
<tr>
<td>sm_MyNewPlugin.vcxproj</td>
<td>5/23/2014 11:54 AM</td>
<td>VC++ Project</td>
</tr>
<tr>
<td>sm_MyNewPlugin.vcxproj.filters</td>
<td>5/23/2014 11:54 AM</td>
<td>VC++ Project Filte...</td>
</tr>
</tbody>
</table>

Open the copied vcxproj file as text in Visual Studio. Use Tools->Create GUID to create a new GUID for the new project.
Find the ProjectGuid element in the vcxproj file and replace it with your new GUID. Change the RootNamespace element to match your new project name.

```xml
<PropertyGroup Label="Globals">
  <ProjectGuid>{6ABCDEFDE-A3E-4B72-A961-50F5D6D9F02B}</ProjectGuid>
  <Keyword>Win32Proj</Keyword>
  <RootNamespace>sm_MyNewPlugin</RootNamespace>
</PropertyGroup>
```

Find the all instance of the name of the old .props file and replace them with your new .props file name.

```xml
<ItemGroup>
  <None Include="sm_MyNewPlugin.props">
    <SubType>Designer</SubType>
  </None>
</ItemGroup>
```

Close the vcxproj, then open the vcxproj.filters file and replace each of the UniqueIdentifier elements with newly created GUIDs. Close the vcxproj.filters file and then open the vcxproj file as a project.

Remove the existing CPP and H file references from the project and you are ready to begin coding your new plugin.

**Creating a Project from Scratch**

Start by opening Visual Studio. Select File->New->Project... and create a Win32 Project in the same directory as the examples. Uncheck the Create directory for solution option.
Creating a Project from Scratch

Start by opening Visual Studio. Select File->New->Project… and create a Win32 Project in the same directory as the examples. Uncheck the Create directory for solution option.
In the Application Wizard, click **Next** to reach the Application Settings page and select DLL as the application type. Also, ensure that the **Empty Project** check box is checked. Then click **Finish**.

Your new empty project should now be open in Visual Studio.
Before we apply the settings, we need to add the 64-bit Unicode configurations. Open the Configuration Manager by clicking Build->Configuration Manager…

From the popup list under Active solution configurations, choose <New…>. Set the name to UDebug and Copy settings from: to Debug. Click OK and then repeat this process for URelease and Release.

From the popup list under Active solution platform, choose <New…>. Set the platform to x64 and Copy settings from to: Win32. Click OK, and then close the Configuration Manager.

Right click on the project in the Solution Explorer and select Properties from the menu.

The following properties need to be set in all configurations (Win32Release, Win32Debug, x64URelease, x64UDebug):
**General**

- Set Output Directory to your plug-ins directory
**C/C++**

**General**

- Add the Spatial Modeler SDK include directory to the Additional Include Directories.

![Additional Include Directories](image)
Preprocessor

- Add __STDC__ to the Preprocessor Definitions
**Linker**

**General**
- Add the Spatial Modeler SDK library directory to the Additional Library Directories.
Input

- Add sblib.lib, eCommonU.lib and pluginManager.lib to Additional Dependencies. If you are creating a RasterOperator, you should include sbrasterlib.lib and sbbasiclib.lib as well.
**Custom Build Step**

**General**

- Set Command Line to
  
  `$(SpatialModelerSDKHome)\bin\$(Platform)$(Configuration)\configure_plugin.exe $(TargetPath)`
  
  replacing `$(SpatialModelerSDKHome)` with your Spatial Modeler SDK install directory

- Set Description to Configuring Spatial Modeler plugin

- Set Outputs to `$(TargetDir)$(TargetName).xml;%(Outputs)`
The purpose of this custom build step is to generate an XML file to accompany your plug-in. The XML file identifies the operators contained in your plug-in so that Spatial Modeler can learn their attributes (name, inputs and outputs) without needing to load your plug-in and its dependent libraries, which may be time consuming. This custom build step is optional but highly recommended.

Once all of the settings are correct, you are ready to begin coding your new plugin.
Run a Spatial Model from the Command Line

The command line executable smprocess.exe is delivered as part of the Spatial Modeler SDK. It is located in the bin\<Configuration> directory in your installation area. It is delivered for all four configurations: Win32Debug, Win32Release, x64UDebug, and x64URelease. The command line syntax is

```
smprocess.exe ModelPathAndName <PortNameAndValuePair> <logfile LogFilePathAndName>
```

ModelPathAndName is the full path to the saved Spatial Model, in quotation marks. PortNameAndValuePair is of the form "PortName=Value" (including the quotation marks). As it implies, PortName is the Name/Display Name of the Port Input or the name of the operator port you wish to set a value on. If referencing a port on an operator, it should be of the form OperatorDisplayName.PortName. Value is a string representation of the value you wish to set on the port. If ambiguous the string value can be prepended by a "cast" to a specific data type. The cast is represented by the data type name in parenthesis. There can be as many PortNameValuePair as you need to specify the parameters for your model. If you wish to generate a log file in a known location, you can use the –logfile option. LogFilePathAndName is the full path for the output log file, in quotation marks.
Examples

Using Input Ports

```
smprocess.exe "d:/data/smprocess_demo2.gmdx" "Raster Filename
In=d:/data/lanier.img" "Band 1=4" "Band 2=3" "Raster Filename
Out=d:/data/ndvi.img" -logfile "c:/temp/logfile.txt"
```

or

```
smprocess.exe "d:/data/smprocess_demo2.gmdx" "Raster Filename
In=d:/data/lanier.img" "Band 1=(RangeList)4" "Band 2=(RangeList)3" "Raster
Filename Out=d:/data/ndvi.img"
```
Without Input Ports

```
smprocess.exe "d:/data/smprocess_demo.gmdx" "Raster Input.Filename=d:/data/lanier.img" "Band 1.BandRange=4" "Band 2.BandRange=3" "Raster Output.FilenameIn=d:/data/ndvi.img"
```

Test What You've Developed for Thread Safety

The command line executable scourgify.exe is a test utility installed as part of the Spatial Modeler SDK. scourgify starts eight threads and runs the specified model over and over in each thread until at least 30 seconds have elapsed. The process is terminated by a watchdog timer if the run time exceeds 5 times the "test run time" (30 seconds). If the test runs without throwing an exception (or crashing), scourgify will return an exit code of 0; otherwise, a non-0 exit code will be returned. If the model creates output files, scourgify will change the output filenames for all but one run so as to avoid conflicts. Note that if the model constructs an output filename from strings (for instance, by a string to an output directory name), scourgify may be unable to determine the output filename and thus be unable to prevent conflicts when it runs multiple copies of the model simultaneously.

scourgify.exe is located in the bin\<Configuration> directory in your installation area. It is delivered for all four configurations: Win32Debug, Win32Release, x64UDebug, and x64URelease. The command line syntax is

```
scourgify -sm -model ModelPathAndName
```
ModelPathAndName is the full path to a saved Spatial Model, in quotation marks. The model must not require parameter input—all inputs to the model must be hardcoded in the model.
What's new in Spatial Modeler 2018

Highlighted API Changes

- RasterInfo::GetBlockWidth, and RasterInfo::GetBlockHeight have been added. These methods allow you to get the native block size associated with the raster. See the User Documentation for more information.

- RasterUtils::SplitUpstreamRequests has been added. This method splits a RasterRequest into smaller tiles so that each test request is less than the working block size threshold. See the User Documentation for more information.

- Methods to facilitate a pattern of operator polymorphism have been added.

  Suppose you create a single operator (the “Prototype”) that represents a particular kind of operation (addition, for instance), but may need to perform that operation on a variety of dissimilar data types. At run time, the Prototype operator can search the operator factory for "Subclass” operators that self-identify as matching the Prototype and choose one or more to handle its inputs. The Subclass operators can remain hidden if desired, and may be deployed separately from the Prototype.

  The new methods are:

  - Operator::SetPrototype(std::wstring) must be called in the constructor of Subclass operators. It is standard to set the prototype name to the Namespace.Name of the Prototype operator that is expected to invoke it.

  - OperatorFactory::GetPluginTypeInfoByPrototype(std::wstring) may be called by the Prototype operator to return a list of all Subclass operators; i.e. those operators matching the specified prototype name (normally this->GetOperatorNamespace() + L"." + this->GetOperatorName()).

The sequence of creating and connecting Subclass operators in the Prototype is critical; please reference code examples on the polymorphism pattern on the Hexagon Community site.

One example in the SMSDK is the (visible) ReadSensorMetadata operator, which invokes the (hidden) ReadGeoEye1Metadata and ReadLandSat8Metadata operators, as well as any other operators that advertise their conformance to the IMAGINE.ReadSensorMetadata prototype. To be used by ReadSensorMetadata (the Prototype), a Subclass operator must do the following:

  i. Call SetPrototype(L"IMAGINE.ReadSensorMetadata") in its constructor.

  ii. Call SetAttribute<StringData>(L"Sensor", sensorname) in its Init method, where sensorname identifies the sensor whose metadata this operator is intended to parse.

  iii. Create a required input port named "MetadataIn" of type ImagineMetadataData. This will accept a raster metadata object read by the Raster Input operator.
iv. Create an optional input port named "FilenameIn" of type FileData. This may specify a specific metadata file (usually text or XML) to read from; if it is not provided, a metadata file should be automatically located based on the DatasetReference property of the metadata.

v. Create an output port named "MetadataOut" of type ImagineMetadataData. This output will be a clone of the MetadataIn value with additional metadata added (usually placed inside the Format Specific member: under Format Specific.LandSat8, for instance).

vi. Create an output port named "FilenameOut" of type FileData. This output will indicate the (text or XML) file from which metadata was read. (If FilenameIn was provided, FilenameOut should be the same value.)

vii. The operator should set an error message (SetErrorMessage) if it fails; the Prototype operator can then continue searching for metadata using other Subclass operators.

Any operator that meets these criteria will be automatically used by the Prototype operator, ReadSensorMetadata.

- Support has been added to allow feature sinks to drive concurrent execution by default and in a uniform manner. This has been facilitated by both changing the default "Spatial Modeler" / "Feature Request Budget" from 0 (request all upstream features at once) to 8192 (request fixed-size increments from upstream) and by adding a method (FeatureInfoData::GetBudgetedDataRequest) for operator developers to access the user-preferred behavior transparently. Developers creating operators that act as feature sinks are still obligated to drive concurrent execution via the ConcurrencyHelper in sbbasiclib (or some other utility of their choosing), but the new FeatureInfoData method allows all feature sinks to conveniently implement behavior that is predictable and controllable by the user. Please see the FeatureInfoData documentation for more information.

Note that point cloud sinks do not yet have access to similar support but this will be addressed in a later release.
Technical Support and Information

Hexagon Geospatial® provides several ways to access information and to contact support, including self-help tools, Hexagon Geospatial Community, Hexagon Geospatial Developer Network, and phone support.

**Hexagon Geospatial Community**


**Blogs**

Get the latest on our technologies: from what engineering is working on, to news about the latest APIs, as well as developer tips and tricks.

**Discussions**

Discuss topics with other Hexagon Geospatial Product pioneers and experts.

**Knowledge and Support**

Learn more about our products, find answers, get the latest updates, and connect with other Hexagon Geospatial Community members, or get support from our support teams [http://www.hexagongeospatial.com/support](http://www.hexagongeospatial.com/support).

**Developer Network**

Share technical information with other developers who use Hexagon Geospatial's SDKs and M.App Portfolio. To get full access to the Developer Network you need to purchase a Hexagon Geospatial Developer Network (HGDN) Subscription. With HGDN, you get broad access to select Hexagon Geospatial development products in one place. You will also get access to powerful toolkits, including currently published APIs and SDKs. You also get access to in-depth resources such as tutorials, collaborative samples, and web-based training.


**eTraining**

Short, to-the-point videos showcasing specific workflows for many tasks and organized by product. We add new videos constantly, so check back often.

**Tutorials**

Written, step-by-step instructions for our most asked-about tasks. Includes helpful hints and introduction material to get you started with our products.
Professional Service Team

For support phone numbers or to submit sales inquiries, general questions, and comments, click the appropriate tabs at the top of the Hexagon Geospatial Support (http://www.hexagongeospatial.com/support) page.
Index

A
About Us • 5
Additional Information • 64
Attributes • 65

B
Best Practices • 64
Build and Run Spatial Models in C++ • 16
Build Spatial Models in C++ • 16

C
Copy and Modify an Example Project • 65
Create a Data Conversion • 54
Create a New Data Type • 46
Create an Operator • 21
Creating a Customization Dialog • 58
Creating a Project from Scratch • 68
Creating Visual Studio Projects for Spatial Modeler Plug-ins • 65

E
Environment • 9
Example 1
Run an NDVI • 16
Example 2
Load/Save/Edit a Spatial Model • 19
Example 3
View Image Operator • 21

Example 4
Determine whether pixel is nodata • 26
Example 5
Lazy-mosaic Images • 30
Example 6
Accessing Raster and Scalar Data • 36
Example 7
Classification Signature • 46
Example 8
time_t to DateTime • 54
Example 9
Color Picker • 58
Extend Spatial Modeler • 20

H
Highlighted API Changes • 83

I
Introduction • 8
Introduction to the Examples • 15

L
Load and Save Spatial Models • 18

M
Models and Submodels • 64

P
Product Documentation Copyright and Terms of Use • 2
R
Run a Spatial Model from the Command Line • 79

S
Spatial Modeler Basics • 10

T
Technical Support and Information • 85
Test What You’ve Developed for Thread Safety • 81

W
What’s new in Spatial Modeler 2018 • 83