Contributing Code

The previous chapters (those found in Part II) offered an introduction to VTK via example. By now it should be apparent that VTK offers the functionality to create powerful graphics, imaging, and visualization applications. In addition, because you have access to the source code, you can extend VTK by adding your own classes. In Part III of the *User's Guide*, we show you how to extend VTK to suit the needs of your application. We begin in this chapter by introducing coding conventions that you may consider adopting—especially if you wish to contribute code to the VTK community. We also describe standard conventions and methods that your objects must implement to be incorporated into VTK. Later in Part III we discuss the implementation details of the process and data objects that make up VTK's visualization pipeline, as well as controlling the execution of the visualization pipeline, and describe how to interface VTK to various windowing systems.

9.1 Coding Considerations

If you develop your own filter or other addition to the *Visualization Toolkit*, we encourage you to contribute the source code. You will have to consider what it means to contribute code from a legal point of view, what coding styles and conventions to use, and how to go about contributing code.

Conditions on Contributing Code To VTK

When you contribute code to VTK, two things are bound to happen. First, many people will see the code—dissecting, improving, and modifying it; and second, you will in some sense "lose control" of the code due to the modifications that will inevitably occur to it. You will not want to release proprietary code or code you cannot relinquish control over (also, patented code is discouraged), and you'll want to carefully craft the code so that others can understand and improve it.

VTK's copyright is an open-source copyright (refer to any `.cxx` or `.h` file to see the copyright in its entirety). The copyright is stated as follows:

Conditions on the copyright are derived from the BSD license (see `www.opensource.org`) and places no constraints on modifying, copying, and redistributing source or binary code, with the exception of the four bulleted clauses, warranty, and indemnification clauses shown above. Other than respecting these four clauses, observing the usual indemnification clause, and commercial patent restriction on the handful of classes in the `VTK/Patented/` directory, you can use VTK in any way whatsoever, including in commercial application.

If these restrictions are acceptable, you can consider contributing code. However, you will need to meet other criteria before your code is accepted. These criteria are not formalized, but have to do with the usefulness, simplicity, and compatibility with the rest of the system. Important questions to ask are:

- Does the code meet the VTK coding standards (see next section)?
- Is the code documented and commented?
- Is the code general? Or is it specific to a narrow application?
- Does it require extensive modification to the system? (For example, modifications to widely-used object APIs.)
- Does the code duplicate existing functionality?
- Is the code robust?
- Does the code belong in a visualization toolkit?

If you can answer these questions favorably, chances are that the code is a good candidate for inclusion in VTK.

Coding Style

There are many useful coding styles, but we insist that you follow just one. We know this is a contentious issue, but we have found that it is very important to maintain a consistent style. Consistent style means that the code is easier to read, debug, maintain, test, and extend. It also means that the automated documentation facilities operate correctly. And these benefits are available to all users of VTK.

Here's a summary of the coding style. You may wish to examine VTK source code to see what it looks like.

- Variables, methods, and class names use changing capitalization to indicate separate words. Instance variables and methods always begin with a capital letter. Static variables are discouraged, but should also begin with a capital letter. Local variables begin with a lower-case letter. SetNumberOfPoints() or PickList are examples of a method name and instance variable.
- Class names are prefixed with `vtk` followed by the class name starting with a capital letter. For example, vtkActor or vtkPolyData are class names. The `vtk` prefix allows the VTK class library to be mixed with other libraries.
- Explicit `this->` pointers are used in methods. Examples include `this->Visibility` and `this->Property` and `this->Execute()`. We have found that the use of explicit `this->` pointers improves code understanding and readability.

- Variable, method, and class names should be spelled out. Abbreviations can be used, but the abbreviation should be entirely in capital letters. For example, vtkPolyDataConnectivityFilter or vtkLODActor.
- Preprocessor variables are written in capital letters. These variables are the only one to use the underscore "_" to separate words. Preprocessor variables should also begin with `VTK_` as in `VTK_LARGE_FLOAT`.
- Instance variables are typically `protected` or `private` class members. Access to instance variables is through `Set/Get` methods. Note that VTK provides `Set/Get` macros which should be used whenever possible (look in `VTK/Common/vtkSetGet.h` for the implementation, and `.h` header files for example usage).
- The indentation style can be characterized as the "indented brace" style. Indentations are two spaces, and the curly brace (scope delimiter) is placed on the following line and indented along with the code (i.e., the curly brace lines up with the code).
- Use `//` to comment code. Methods are commented by adding `// Description:` followed by lines each beginning with `//`.

How To Contribute Code

Contributing code is fairly easy once you've created your class or classes following the coding convention described above. First, include the copyright notice in both the `.cxx` and `.h` source files. You may wish to place your name, organization, and or other identifying information in the "`Thanks:`" field of the copyright notice (see `Vtk/Hybrid/vtkEarthSource.h` for an example). Next, send e-mail to `kitware@kitware.com` with an explanation of what the code does, sample data (if needed), test code (in either C++, Tcl, or Python), and the source code (a single `.h` and `.cxx` file per class). Another method is to send the same information to the `vtkusers` mailing list (see "Additional Resources" on page 4 for information about joining the list). The advantage of sending contributed code to the mailing list is that users can take advantage of the code immediately.

There is no guarantee that contributed code will be incorporated into official VTK releases. This depends upon the quality, usefulness, and generality of the code as outlined in "Conditions on Contributing Code To VTK" on page 201.

(Note: skilled developers can obtain CVS write access and add their own code to the VTK code repository. Contact `kitware@kitware.com` for more information.)

9.2 Standard Methods: Creating and Deleting Objects

Almost every object in VTK responds to a set of standard methods. Many of these methods are implemented in vtkObject, from which most VTK classes are derived. However, subclasses typically require that you extend or implement the inherited methods for proper behavior. For example, the New() method should be implemented by every concrete (i.e., non-abstract, instantiable) class, while the Delete() is generally inherited from its superclass' vtkObject. Before you develop any code you should become familiar with these standard methods, and make sure your own classes support them.

`New()`

This static class method is used to instantiate objects. We refer to this method as an "object factory" since it is used to create instances of a class. In VTK, every New() method should be paired with the Delete() method. (See also "Object Factories" on page 211.)

`instance = MakeObject()`

This method is a virtual constructor. That is, invoking this method causes an object to create an instance of the same type as itself and then return a pointer to the new object.

`Delete()`

Use this method to delete a VTK object created with the New() or MakeObject() method. Depending upon the nature of the object being deleted, this may or may not actually delete the object. For example, reference-counted objects will only be deleted if their reference count goes to zero.

`DebugOn()/DebugOff()`

Turn debugging information on or off. These methods are inherited from vtkObject.

`Print()`

Print out the object including superclass information. The Print() method, which is defined in vtkObject, requires implementation of a PrintSelf() method for each class. The PrintSelf() method is invoked in a chain, each subclass calling its superclass's PrintSelf() and then printing its own instance variables.

`PrintSelf(ostream, indent)`

Each class should implement this method. The method invokes its parents PrintSelf(), followed by methods to print itself.

`name = GetClassName()`

Return the name of the class as a character string. This method is used for debugging

information. (In vtk3.1, the macro vtkTypeMacro found in `vtkSetGet.h` defines this method.)

`flag = IsA(className)`

Return non-zero if the named class is a superclass of, or the same type of, this class. (In vtk3.1 and beyond, the macro vtkTypeMacro found in `vtkSetGet.h` defines this method.)

`<class> *ptr = <class>::SafeDownCast(vtkObject *o)`

This static class method is available in C++ for performing safe down casts (i.e., casting a general class to a more specialized class). If `ptr` is returned `NULL`, then the down cast failed, otherwise `ptr` points to an instance of the class `<class>`. For example, `ptr=vtkActor::SafeDownCast(prop)` will return non-`NULL` if `prop` is a vtkActor or a subclass of vtkActor. (In vtk3.1, the macro vtkTypeMacro found in `vtkSetGet.h` defines this method.)

`void Modified()`

This updates the internal modification time stamp for the object. The value is guaranteed to be unique and monotonically increasing.

`mtime = GetMTime()`

Return the last modification time of an object. Normally this method is inherited from vtkObject, however in some cases you'll want to overload it. See "Modified Time" on page 277 for more information.

The most important information you can take from this section is this: instances should be created with the New() method and destroyed with the Delete() method; and that for every New() there should be a Delete() method. For example, to create an instance of an actor:

```
vtkActor *anActor = vtkActor::New();
... (more stuff)
anActor->Delete();
```

The New() method is called an object factory: it's a class method used to create instances of the class. Typically, the New() method first asks the vtkObjectFactory to create an instance of the object, and if that fails, it simply invokes the C++ `new` operator as is shown in this excerpted code from `vtkSphereSource.cxx`:

```
vtkSphereSource* vtkSphereSource::New()
{
  // First try to create the object from the vtkObjectFactory
  vtkObject* ret = vtkObjectFactory::CreateInstance(
```

```
                                                    "vtkSphereSource");
  if(ret)
    {
    return (vtkSphereSource*)ret;
    }
  // If the factory was unable to create the object, create it here.
  return new vtkSphereSource;
  }
```

However, the New() method can be more complex, for example, to instantiate device-independent classes. For example, in `vtkGraphicsFactory.cxx`, the code used to instantiate a vtkActor looks like this:

```
if(strcmp(vtkclassname, "vtkActor") == 0)
  {
  return vtkOpenGLActor::New();
  }
```

Here the New() method is used to create a device-*dependent* subclass of vtkActor (i.e., OpenGL), which is then returned as a pointer to a device-*independent* vtkActor. For example, depending on the compile-time flags (e.g., `VTK_USE_OGLR`) and possibly other information such as environment variables, different actor types corresponding to the rendering libraries (e.g., OpenGL, Mesa, and so on) are created transparently to the user. Using this mechanism we can create device-independent applications, or at run-time select different classes to use in the application. (See "Object Factories" on page 211 for more information.)

9.3 Copying Objects & Protected Methods

The constructor, destructor, operator=, and copy constructor methods are either protected or private members of most every VTK class. This means that for VTK class `vtkX`, the methods

- `vtkX()` — constructor
- `~vtkX()` — destructor

are protected, and the methods

- `operator=(const vtkX &)` — equivalence operator, and
- `vtkX(const vtkX &)` — copy constructor

are private. In addition, the assignment operator and copy constructors should be declared only, and not implemented. This prevents the compiler from creating one automatically, and does not generate code that cannot be covered in the testing process. This means that you cannot use these methods in your application. (The reason for this is to prevent potentially dangerous misuse of these methods. For example, reference counting can be broken by using the constructor and destructor rather than the standard `New()` and `Delete()` methods described in the previous section.)

Since the copy constructor and operator= methods are private, other methods must be used to copy instances of an object. The methods to use are `DeepCopy()` and/or `ShallowCopy()`. For example:

```
vtkActor *a1 = vtkActor::New();
vtkActor *a2 = vtkActor::New();
a2->ShallowCopy(a1);
```

A shallow copy is a copy of an object that copies references to objects (via reference counting) rather than the object themselves. For example, instances of the class `vtkActor` refer to a `vtkMapper` object (the mapper represents the geometry for the actor). In a shallow copy, replication of data is avoided, and just the reference to the `vtkMapper` is copied. However, any changes to the shared mapper indirectly affects all instances of `vtkActor` that refer to it. Alternatively, a `DeepCopy()` can be used to copy an instance, including any data it represents, without any references to other objects. Here's an examples:

```
vtkIntArray *ia1 = vtkIntArray::New();
vtkIntArray *ia2 = vtkIntArray::New();
ia1->DeepCopy(ia2);
```

In this example, the data in `ia2` is copied into `ia1`. From this point on, `ia1` and `ia2` can be modified without affecting each other.

At the current time, VTK does not support `DeepCopy()` and `ShallowCopy()` in all classes. This will be added in the future.

9.4 Writing A VTK Class: An Overview

In this section we give a broad overview of how to write a VTK class. If you are writing a new filter, you'll also want to refer to Chapter 12 "How To Write A Process Object" on

page 275. If you want to extend the graphics subsystem, see Chapter 12 "Extending The Graphics System" on page 291. And of course, you should read the previous portion of this chapter.

Probably the hardest part about writing a VTK class is figuring out if you need it, and if so, where it fits into the system. These decisions come easier with experience. In the mean time, you probably want to start by working with other VTK developers, or posting to the `vtkusers` mailing list (see "Additional Resources" on page 4). If you determine that a class is needed, you'll want to look at the following issues.

Find A Similar Class

The best place to start is to find a class that does something similar to what you want to do. This will often guide the creation of the object API, and/or the selection of a superclass.

Identify A Superclass

Most classes should derive from vtkObject or one of vtkObject's descendants. Exceptions to this rule are few, since vtkObject implements important functionality such as reference counting, command/observer user methods, print methods, and debugging flags. All VTK classes use single inheritance. While this is a contentious issue, there are good reasons for this policy including Java support (Java allows only single inheritance) and simplification of the wrapping process as well as the code. You may wish to refer to the object diagrams found in "Object Diagrams" on page 321. These provide a succinct overview of many inheritance relationships found in VTK.

Single Class Per .h File

Classes in VTK are implemented one class per `.h` header file, along with any associated `.cxx` implementation file. There are some exceptions to this rule—for example, when you have to define internal helper classes. However, in these exceptions the helper class is not visible outside of the principle class. If it is, it should placed into its own `.h`/`.cxx` files.

Required Methods

Several methods and macros must be defined by every VTK class as follows. See "Standard Methods: Creating and Deleting Objects" on page 205 for a description of these methods.

- `New()` — Every non-abstract class (i.e., a class that does not implement a pure virtual function) must define the `New()` method.
- `vtkTypeMacro(className,superclassName)` — This macro is a convenient way to define methods used at run-time to determine the type of an instance, or to perform safe down casting. `vtkTypeMacro` is defined in the file `vtkSetGet.h`. The macro defines the methods GetClassName(), IsA(), and SafeDownCast().
- `PrintSelf()` — Print out instance variables in an intelligent manner.
- Constructor (must be `protected`)
- Destructor (must be `protected`)
- Copy Constructor (must be `private` and not implemented)
- `operator=` (must be `private` and not implemented)

The constructor, copy constructor, destructor, and `operator=` must not be public. The `New()` method should use the procedure outlined in "Object Factories" on page 211. Of course, depending upon the superclass(es) of your class, there may be additional methods to implement to satisfy the class API (i.e., fill in abstract virtual functions).

Document Code

The documentation of VTK classes depends upon proper use of documentation directives. Each `.h` class file should have a `// .NAME` description, which is a single-line description of what the class does. In addition, the header file should have a `// .SECTION Description` section, which is a multi-paragraph description (delimited with the C++ comment indicator `//`) giving detailed information about each class and how it works. Other possible information is contained in the `// .SECTION Caveats` section, which describes quirks and limitations of the class, and the `// .SECTION See Also` section, which refers to other, related classes.

Methods should also be documented. If a method satisfies a superclass API, or overloads a superclass method, you may not need to document the method. Those methods that you do document use the following construct:

```
// Description:
// This is a description...
```

Of course, you may want to embed other documentation, comments, etc. into the code to help other developers and users understand what you've done.

Use SetGet Macros

Whenever possible, use the `SetGet` macros found in `VTK/Common/SetGet.h` to define access methods to instance variables. Also, you'll want to use the debugging macros and additional `#defines` (e.g., `VTK_LARGE_FLOAT`) in your code, as necessary.

Add Class To VTK

Once you've created your class, you'll want to decide whether you want to incorporate it into the VTK build, or separate, in your own application. If you add your class to VTK, modify the `CMakeLists.txt` file in the appropriate subdirectory. You'll then have to re-run CMake (as described in "Running CMake" on page 15), and then recompile. You can add an entire new library by creating a file called `LocalUser.cmake` in the top level of your VTK source tree. This file is used only if it exits. You can put a CMake `SUBDIR` command into the file, telling CMake to go into the new library's directory. Inside the directory, you will need to create a new `CMakeLists.txt` file that contains CMake commands for building your library. If you are using one of the wrapped languages like Tcl, you can either add your library into the VTK executable with another `LocalUser.cmake` file in the `Wrapping/Tcl` directory, or you will have to build shared libraries, and load the new library at run time into the Tcl environment. To add classes that simply use VTK, see "C++" on page 30 for example `CMakeLists.txt` files.

9.5 Object Factories

VTK Version 3.0 and later has a potent capability that allows you to extend VTK at run time. Using *object factories*, you can replace a VTK object with one of your own creation.

For example, if you have special hardware, you create your own special high-performance filter at *run-time* by replacing object(s) in VTK with your own objects. So, if you wanted to replace the vtkImageFFT filter with a filter that performed FFT in hardware, or replace the cell vtkTetra with a high-performance, assembly code implementation, you could do this. Here are the benefits of using object factories.

- Allow sub-classes to be used in place of parent classes in existing code.
- Your application can dynamically load new object implementations.
- You can extend VTK at run-time.
- Proprietary extensions can be isolated from the public VTK builds.
- Removes the need for many `#ifdefs` in C++ code, for example, an OpenGL factory could replace all of the `#ifdefs` in vtkRenderer and vtkRenderWindow.
- An object factory can be used as a debugging aid. For example, a factory can be created that does nothing except track the invocation of `New()` for each class.
- Easier implementation of accelerated or alternative VTK objects on different hardware similar to the plug-in model of Netscape and Photoshop.

Overview

The key class when implementing an object factory is vtkObjectFactory. vtkObjectFactory maintains a list of "registered" factories. It also defines a static class method used to create VTK objects by string name—the method CreateInstance()—which takes as an argument a `const char*`. The create method iterates over the registered factories asking each one in turn to create the object. If the factory returns an object, that object is returned as the created instance. Thus, the first factory returning an object is the one used to create the object.

An example will help illustrate how the object factory is used. The New() method from the class vtkVertex is shown below.

```
vtkVertex* vtkVertex::New()
{
  // First try to create the object from the vtkObjectFactory
  vtkObject* ret = vtkObjectFactory::CreateInstance("vtkVertex");
  if(ret)
    {
    return (vtkVertex*)ret;
```

```
    }

  // If the factory was unable to create the object, then create it
  return new vtkVertex;
}
```

The implementation of this New() method is similar to most all other New() methods found in VTK. If the object factory does not return an instance of class vtkVertex, then the constructor for vtkVertex is used. Note that the factory must return an instance of a class that is a subclass of the invoking class (i.e., vtkVertex).

How To Write A Factory

The first thing you need to do is to create a subclass of vtkObjectFactory. You must implement two virtual functions in your factory: GetVTKSourceVersion() and GetDescription().

```
virtual const char* GetVTKSourceVersion();
virtual const char* GetDescription();
```

GetDescription() returns a string defining the functionality of the object factory. The method GetVTKSourceVersion() should return `VTK_SOURCE_VERSION` and NOT call vtkVersion::GetVTKSourceVersion(). You cannot call vtkVersion functions, because the version must be compiled into your factory, if you did call vtkVersion functions, it would just use the VTK version that the factory was loaded into and not the one it was built with. This method is used to check the version numbers of the factory and the objects it creates against the installed VTK. If the software versions are different, a warning will be produced and there's a good chance that a serious program error will follow.

There are two ways for a factory to create objects. The most convenient method is to use the protected method of vtkObjectFactory called RegisterOverride().

```
void RegisterOverride(const char* classOverride,
                      const char* overrideClassName,
                      const char* description,
                      int enableFlag,
                      CreateFunction createFunction);
```

This method should be called in your constructor once for each object your factory provides. The following are descriptions of the arguments:

- classOverride — This is the name of the class you are replacing.
- overrideClassName — This is the name of the class that will replace classOverride
- description — This is a text based description of what your replacement class does. This can be useful from a GUI if you want to select which object to use at run-time.
- enableFlag — This is a Boolean flag that should be 0 or 1. If it is 0, this override will not be used. Note, it is possible to change these flags at run-time from vtkObjectFactory class interface.
- createFunction — This is a pointer to a function that will create your class. The function must look like this:

```
vtkObject* createFunction();
```

  You can write your own function or use the `VTK_CREATE_FUNCTION` macro provided in vtkObjectFactory.h.

The second way in which an object factory can create objects is the virtual function CreateObject():

```
virtual vtkObject* CreateObject(const char* vtkclassname);
```

The function should return NULL if your factory does not want to handle the class name it is being asked to create. It should return a sub-class of the named VTK class if it wants to override the class. Since the CreateObject() method returns a vtkObject* there is not much type safety other than the object must be a vtkObject, so be careful to only return sub-classes of the object to avoid run-time errors. A factory can handle as many objects as it wants. If many objects are to be created, it would be best to use a hash table to map from the string names to the object creation. The method should be as fast as possible since it may be invoked frequently. Also note that this method will not allow a GUI to selectively enable and disable individual objects like the RegisterOverride() method can.

How To Install A Factory

How factories are installed depends on whether they are compiled into your VTK library or application; or whether they are dynamically loaded DLLs or shared libraries. Compiled in factories need only call

```
vtkObjectFactory::RegisterFactory ( MyFactory::New() );
```

For dynamically loaded factories, a shared library or DLL must be created that contains the object factory subclass. The library should include the macro `VTK_FACTORY_INTERFACE_IMPLEMENT(factoryName)`. This macro defines three external "C" linkage functions named vtkGetFactoryCompilerUsed(), vtkGetFactoryVersion(), and vtkLoad() that returns an instance of the factory provided by the library.

```
#define VTK_FACTORY_INTERFACE_IMPLEMENT(factoryName)   \
extern "C"                                             \
VTK_FACTORY_INTERFACE_EXPORT                           \
const char* vtkGetFactoryCompilerUsed()                \
{                                                      \
  return VTK_CXX_COMPILER;                             \
}                                                      \
extern "C"                                             \
VTK_FACTORY_INTERFACE_EXPORT                           \
const char* vtkGetFactoryVersion()                     \
{                                                      \
  return VTK_SOURCE_VERSION;                           \
}                                                      \
extern "C"                                             \
VTK_FACTORY_INTERFACE_EXPORT                           \
vtkObjectFactory* vtkLoad()                            \
{                                                      \
  return factoryName ::New();                          \
}
```

The library must then be put in the `VTK_AUTOLOAD_PATH`. This variable follows the convention of `PATH` on your machine using the separation delimiters ";" on Windows, and ":" on Unix. The first time the vtkObjectFactory is asked to create an object, it loads all shared libraries or DLLs in the `VTK_AUTOLOAD_PATH`. For each library in the path, vtkLoad() is called to create an instance of vtkObjectFactory. This is only done the first time to avoid performance problems. However, it is possible to re-check the path for new factories at run time by calling

```
vtkObjectFactory::ReHash();
```

(Note that the VTK class vtkDynamicLoader handles operating system independent loading of shared libraries or DLLs.)

Example Factory

Here is a simple factory that uses OLE automation on the Windows operating system to redirect all VTK debug output to a Microsoft Word document. To use this factory, just compile the code into a DLL, and put it in your `VTK_AUTOLOAD_PATH`.

```
#include "vtkOutputWindow.h"
#include "vtkObjectFactory.h"
#pragma warning (disable:4146)
#import "mso9.dll"
#pragma warning (default:4146)
#import "vbe6ext.olb"
#import "msword9.olb" rename("ExitWindows", "WordExitWindows")
// This class is exported from the vtkWordOutputWindow.dll
class vtkWordOutputWindow : public vtkOutputWindow {
public:
  vtkWordOutputWindow();
  virtual void DisplayText(const char*);
  virtual void PrintSelf(vtkOstream& os, vtkIndent indent);
  static vtkWordOutputWindow* New() { return new vtkWordOutputWindow;}
protected:
  Word::_ApplicationPtr m_pWord;
  Word::_DocumentPtr m_pDoc;
};

class vtkWordOutputWindowFactory : public vtkObjectFactory
{
public:
  vtkWordOutputWindowFactory();
  virtual const char* GetVTKSourceVersion();
  virtual const char* GetDescription();
};

// vtkWordOutputWindow.cpp : the entry point for the DLL application.
//
#include "vtkWordOutputWindow.h"
#include "vtkVersion.h"
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved )
{
  if(ul_reason_for_call == DLL_PROCESS_ATTACH)
    {
    CoInitialize(NULL);
    }
```

```
    return TRUE;
}

void vtkWordOutputWindow::PrintSelf(vtkOstream& os, vtkIndent indent)
{
  vtkOutputWindow::PrintSelf(os, indent);
  os << indent <<  "vtkWordOutputWindow " << endl;
}

// This is the constructor of a class that has been exported.
// see vtkWordOutputWindow.h for the class definition
vtkWordOutputWindow::vtkWordOutputWindow()
{
   try
     {
     HRESULT hr = m_pWord.CreateInstance(__uuidof(Word::Application));
     if(hr != 0) throw _com_error(hr);

     m_pWord->Visible = VARIANT_TRUE;
     m_pDoc = m_pWord->Documents->Add();
     }
   catch (_com_error& ComError)
     {
     cerr << ComError.ErrorMessage() << endl;
     }
}

void vtkWordOutputWindow::DisplayText(const char* text)
{
  m_pDoc->Content->InsertAfter(text);
}

// Use the macro to create a function to return a vtkWordOutputWindow
VTK_CREATE_CREATE_FUNCTION(vtkWordOutputWindow);

// Register the one override in the constructor of the factory
vtkWordOutputWindowFactory::vtkWordOutputWindowFactory()
{
   this->RegisterOverride("vtkOutputWindow",
  "vtkWordOutputWindow",
  "OLE Word Window",
  1,
  vtkObjectFactoryCreatevtkWordOutputWindow);
}
```

```
// Methods to load and insure factory compatibility.
VTK_FACTORY_INTERFACE_IMPLEMENT(vtkWordOutputWindowFactory);

// return the version of VTK that the factory was built with
const char* vtkWordOutputWindowFactory::GetVTKSourceVersion()
{
  return VTK_SOURCE_VERSION;
}

// return a text description of the factory
const char* vtkWordOutputWindowFactory::GetDescription()
{
  return "vtk debug output to Word via OLE factory";
}
```