# Server Manager

The ParaView server manager is responsible for handling communication between the client and the server(s) as well as maintaining a copy of the state of the server(s). It was created to simplify the development of distributed visualization applications. The important components of the server manager are as follows.

**Proxies**: `vtkSMProxy` and its subclasses provide placeholders for VTK objects (`vtkObject` and its subclasses) that reside on the server(s). Proxies are responsible for the following tasks.

- Creating the server-side VTK objects
- Maintaining references to server-side VTK objects using `vtkClientServerID`'s
- Maintaining a copy of the state of the server-side objects using properties (see below)

**Properties**: `vtkSMProperty` and its subclasses provide the client access to the interface of server-side objects represented by proxies. Each property has one or more commands and optionally values that are passed as arguments to the command(s).

**Domains**: `vtkSMDomain` and its subclasses represent the possible values properties can have. For example, the resolution of a sphere may be limited to a range of values between 2 and positive infinity.

## 1.1 General Information

Most server manager objects are subclasses of `vtkSMObject`. `vtkSMObject` provides access to the following two singletons.

```
static vtkSMProxyManager* GetProxyManager();

static vtkSMApplication* GetApplication();
```

The `vtkSMProxyManager` and `vtkSMApplication` singletons must be created and assigned at the beginning of the application.

```
void vtkSMApplication::Initialize()
{
  // ...
  vtkSMProxyManager* proxyM = vtkSMProxyManager::New();
  this->SetProxyManager(proxyM);
  this->SetApplication(this);
  // ...
}
```

## 1.2 Proxy Creation and Management

`vtkSMProxyManager` is a singleton that creates and manages proxies. It maintains a map of XML elements (populated by the XML parser) from which it can create and initialize proxies and properties. Once a proxy is created, it can either be managed by the user code or the proxy manager. In the latter case, pass the control of the proxy to the manager with `RegisterProxy()` and call `UnRegister()` on the proxy. At destruction, the proxy manager deletes all managed proxies.

The server manager configuration is stored in XML files (described in detail in section 1.6). These XML files, and possibly additional user- or developer-specified ones, are parsed by `vtkSMXMLParser` in ParaView's initialization stage using the following methods. (`vtkPVXMLParser` is the superclass of `vtkSMXMLParser`, and `vtkXMLParser` is its superclass.)

```
void vtkSMXMLParser::ProcessConfiguration(
                              vtkSMProxyManager* manager);

void vtkPVXMLParser::SetFileName(const char* fileName);

int vtkSMXMLParser::Parse(const char* xml_string);
```

Shown below is an example of using these methods in `vtkSMApplication`'s `ParseConfigurationFile()` method.

```
int vtkSMApplication::ParseConfigurationFile(
  const char* fname, const char* dir)
{
  // ...
  vtkSMProxyManager* proxyM = this->GetProxyManager();
  vtksys_ios::ostringstream tmppath;
  tmppath << dir << "/" << fname << ends;
  vtkSMXMLParser* parser = vtkSMXMLParser::New();
  parser->SetFileName(tmppath.str().c_str());
  int res = parser->Parse();
  parser->ProcessConfiguration(proxyM);

  parser->Delete()
  return res;
}
```

Once an XML file or string is parsed, the configuration is stored in the proxy manager as a hierarchy of `vtkPVXMLElement`'s. Internally, this is a stored as

```
map<vtkStdString,
    map<vtkStdString, vtkSmartPointer<vtkPVXMLElement> >
```

where the first `vtkStdString` is a unique group name, and the second vtkStdS`tring` is a unique proxy name. An example XML file demonstrating this is shown below.

```
<ServerManagerConfiguration>
  <ProxyGroup name="filters">
   <SourceProxy name="AllToN">
   </SourceProxy>
  </ProxyGroup>
</ServerManagerConfiguration>
```

Proxy groups (see the `ProxyGroup` XML element above) are only used for organizational purposes; it is perfectly valid to store all proxy descriptions in one group. Note that these are not actual proxies but proxy descriptions stored in `vtkPVXMLElement` objects. To actually create proxies, the following method is used.

```
vtkSMProxy* vtkSMProxyManager::NewProxy(
            const char* groupName, const char* proxyName);
```

This method creates and initializes a proxy from the given group and with the given proxy name. The `vtkSMProxy*` returned has a reference count of 1. At this point, the developer can choose to maintain this reference to the proxy by passing the proxy back to the proxy manager with the `RegisterProxy()` method.

```
void vtkSMProxyManager::RegisterProxy(
                  const char* groupName, const char* name,
                  vtkSMProxy* proxy);
```

Once `RegisterProxy()` is called, the proxy manager will maintain a reference to the proxy so the developer can safely call `UnRegister()` on it. Note that the group name and name passed to `RegisterProxy()` are not necessarily the same as the ones passed to `NewProxy()`. For `RegisterProxy()`, the group name points to a map in which proxy *instances* are stored (as opposed to XML representations), and the name is the key used for retrieving the proxy from this map using the first method shown below. The second method looks for the proxy in all groups and returns the first match.

```
vtkSMProxy* vtkSMProxyManager::GetProxy(
                const char* groupName, const char* name);

vtkSMProxy* vtkSMProxyManager::GetProxy(const char* name);
```

The groups in this case (used by `RegisterProxy()` and `GetProxy()`) are used for more than simply organizational purposes. Grouping instances allows a domain (see section 1.4) to list only a sub-group of all instances. For examples, all sources that can be used as glyphs by the **Glyph** filter (i.e., cone, sphere, arrow, etc.) can be stored in a group called `glyph_sources`.

To cause the proxy manager to release its reference to a previously registered proxy, one of the following is called. The first method releases its reference to the proxy pointed to by `proxy` with the given `groupName` and `name`. The second method releases its reference to the proxy with the given `groupName` and `name`. The third one releases its reference to the first proxy in any group with the specified name. The last method releases its references to all proxies that have been registered with it.

```
void vtkSMProxyManager::UnRegisterProxy(
    const char* groupName, const char* name, vtkSMProxy* proxy);

void vtkSMProxyManager::UnRegisterProxy(
   const char* groupName, const char* name);

void vtkSMProxyManager::UnRegisterProxy(const char* name);
```

```
void vtkSMProxyManager::UnRegisterProxies();
```

Finally, the proxy manager provides the following utility method.

```
void vtkSMProxyManager::InstantiateGroupPrototypes(
                                    const char* groupName);
```

Given the specified group name, this function creates one proxy instance per proxy representation in the group. It stores all these instances in an instance group called `groupName_prototypes`. This function is commonly used for automatically creating prototypes for an entire group. The prototypes can then be used for accessing information about the proxies not easily available from the proxy descriptions.

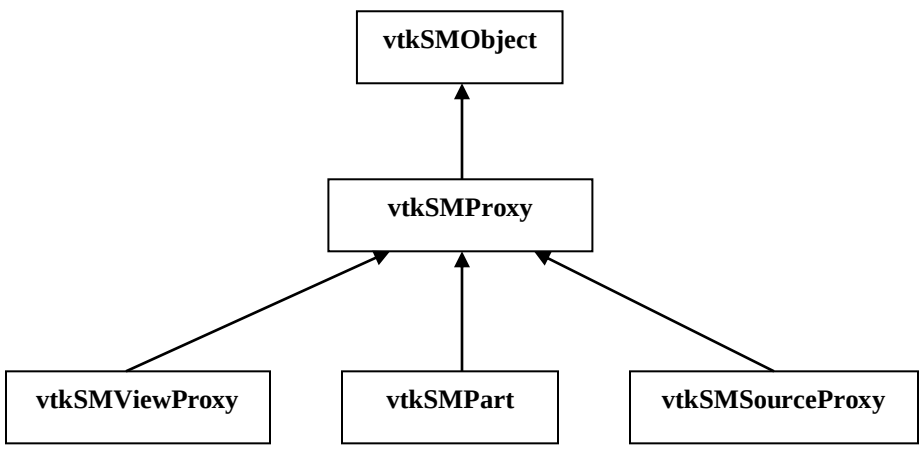


**Figure 1. Partial class hierarchy for proxies**

## 1.3 Properties

Each instance of `vtkSMProperty` (or of one of its subclasses) represents a method and any associated arguments of a VTK object stored on one or more client, server manager, data server, or render server nodes. It may have a state and can push this state to the VTK object to which it refers. `vtkSMProperty` only supports methods with no arguments. Use one of its subclasses for a method with arguments. Different types of arguments are supported by different subclasses. A list of the property class names and the type(s) of arguments they support follows. ParaView's on-line documentation, created with Doxygen, provides information about the interfaces to these classes; it can be found at http://www.paraview.org/ParaView3/Doc/Nightly/html/classes.html.

- `vtkSMProperty`: no arguments
- `vtkSMProxyProperty`: proxy argument
- `vtkSMInputProperty` (a subclass of `vtkSMProxyProperty`): proxy argument
- `vtkSMVectorProperty`: abstract superclass for vector properties, contains one or more arguments of the type specified by the subclass
- `vtkSMDoubleVectorProperty`: vector of `double`'s
- `vtkSMIntVectorProperty`: vector of `int`'s
- `vtkSMStringVectorProperty`: vector of strings or some combination of strings, `int`'s, and `double`'s
- `vtkSMIdTypeVectorProperty`: vector of `vtkIdType`'s (a type in VTK used for point and cell ids)

Usually, the developer does not directly create properties. The properties are created and assigned when the XML configuration file(s) and/or string(s) are parsed. Example XML code to be parsed by ParaView for a single proxy and its associated properties is shown below. A thorough description of the XML used in the server manager is described in section 1.6.

```
<SourceProxy name="CubeSource" class="vtkCubeSource">

  <DoubleVectorProperty
    name="XLength"
    command="SetXLength"
    number_of_elements="1"
    default_values="1.0" >
    <DoubleRangeDomain name="range" min="0" />
  </DoubleVectorProperty>

  <DoubleVectorProperty
    name="YLength"
    command="SetYLength"
    number_of_elements="1"
    default_values="1.0" >
    <DoubleRangeDomain name="range" min="0" />
  </DoubleVectorProperty>
```

```
  <DoubleVectorProperty
    name="ZLength"
    command="SetZLength"
    number_of_elements="1"
    default_values="1.0" >
    <DoubleRangeDomain name="range" min="0" />
  </DoubleVectorProperty>

  <DoubleVectorProperty
    name="Center"
    command="SetCenter"
    number_of_elements="3"
    default_values="0.0 0.0 0.0" >
    <DoubleRangeDomain name="range"/>
  </DoubleVectorProperty>

  <!-- End Box -->

</SourceProxy>
```

If you need to create and add your own properties, the following method can be used.

```
void vtkSMProxy::AddProperty(const char* name,
                             vtkSMProperty* prop);
```

The property can then be retrieved with this method.

```
vtkSMProperty* vtkSMProxy::GetProperty(const char* name);
```

This is true for properties created through XML parsing as well.

`vtkSMProxy` has two important methods related to properties.

```
// Update the VTK object on the server by pushing the
// values of all modified properties (unmodified
// properties are ignored). If the object has not been
// created, it will be created first.
virtual void UpdateVTKObjects();

// Updates all property information by calling
// UpdateInformation() and populating the values. This
// method also calls UpdateDependentDomains() on all
```

```
// properties to make sure that domains that depend on the
// information are updated.
virtual void UpdatePropertyInformation();
```

Below is a C++ example of how properties are used.

```
someProxy->UpdatePropertyInformation();

vtkSMDoubleVectorProperty* info =
  vtkSMDoubleVectorProperty::SafeDownCast(
    someProxy->GetProperty("some information property"));
// do something with the values of the double vector
// property obtained from the server

vtkSMDoubleVectorProperty* prop =
  vtkSMDoubleVectorProperty::SafeDownCast(
    someProxy->GetProperty("some property"));
prop->SetElement(0, 5.17);

// Note that the property value is not pushed to the
// server until the following is called
someProxy->UpdateVTKObjects();
```

### vtkSMProperty

`vtkSMProperty` is the superclass of all server manager properties. All subclasses of `vtkSMProperty` overwrite at least two virtual methods defined in this class.

```
// Append a command to update the VTK object with the
// property values(s). The proxy objects create a stream
// by calling this method on all the modified properties.
virtual void AppendCommandToStream(
               vtkSMProxy*, vtkClientServerStream* stream,
               vtkClientServerID objectId );

// Set the appropriate instance variables from the XML
// element. This should be overwritten by any subclass if
// adding instance variables.
virtual int ReadXMLAttributes(vtkSMProxy* parent,
                              vtkPVXMLElement* element);
```

The other methods and instance variables of this class are documented on-line at http://www.paraview.org/ParaView3/Doc/Nightly/html/classvtkSMProperty.html. The documentation was generated from the source code by Doxygen.

### vtkSMProxyProperty

`vtkSMProxyProperty` is a concrete subclass of `vtkSMProperty` representing pointer(s) to `vtkObject`(s) (through `vtkSMProxy`). Note that if the proxy has multiple IDs, they are all appended to the command stream. If `UpdateSelf` (an instance variable of `vtkSMProperty`) is true, the proxy ids (as opposed to the server object ids) are passed to the stream. This instance variable is set through ParaView's server manager XML. (See section 1.6.)

### vtkSMInputProperty

`vtkSMInputProperty` is a concrete subclass of `vtkSMProperty` representing inputs to a filter (through `vtkSMProxy`). It is a special property that always calls `AddInput` on a `vtkSMSourceProxy`. This property is further discussed in section 1.5.

### Information properties

If the `InformationOnly` instance variable (in `vtkSMProperty`) of a property is set (`information_only="1"` in XML), the property behaves as an information property. Information properties are used to obtain information from the server (i.e., pull) as opposed to sending information (i.e., push). Since there can be more than one way of filling a property with values, the property delegates this functionality of retrieving information to a helper that is configured in the XML. The C++ code to do this and the corresponding XML are shown below.

```
void vtkSMProperty::UpdateInformation(
   vtkIdType cid, int serverIds, vtkClientServerID objectId)
{
  if (!this->InformationOnly)
    {
    return;
    }

  if (this->InformationHelper)
    {
    this->InformationHelper->UpdateProperty(cid,
                                            serverIds,
                                            objectId,
                                            this);
    }
}
```

```
<IntVectorProperty
  name="TimeStepRangeInfo"
  command="GetTimeStepRange"
  information_only="1">
  <SimpleIntInformationHelper/>
</IntVectorProperty>
```

In the simplest case (i.e., `vtkSMSimpleIntInformationHelper`), an information helper calls the command and assigns the values returned by the server to the individual elements of the `vtkSMIntVectorProperty`. There are also more sophisticated information helpers. For example, `vtkSMArraySelectionInformationHelper` instantiates a helper class called `vtkPVServerArraySelection` on the server and uses it to collect information across nodes about the arrays available in a data set.

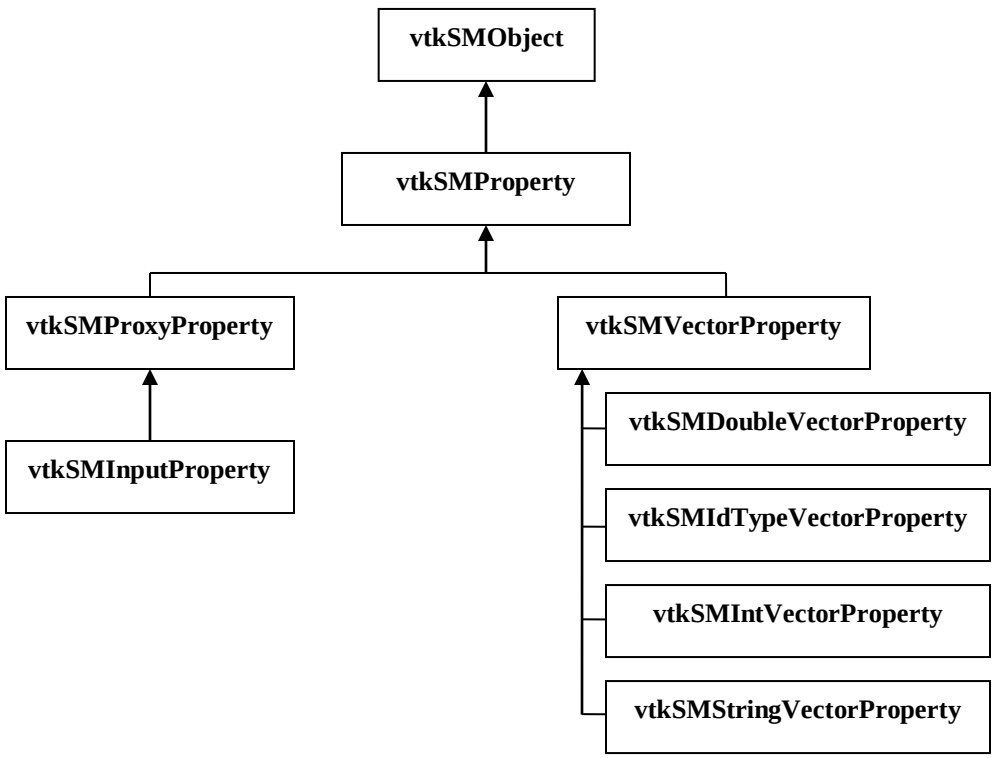


**Figure 2. Class hierarchy for properties**

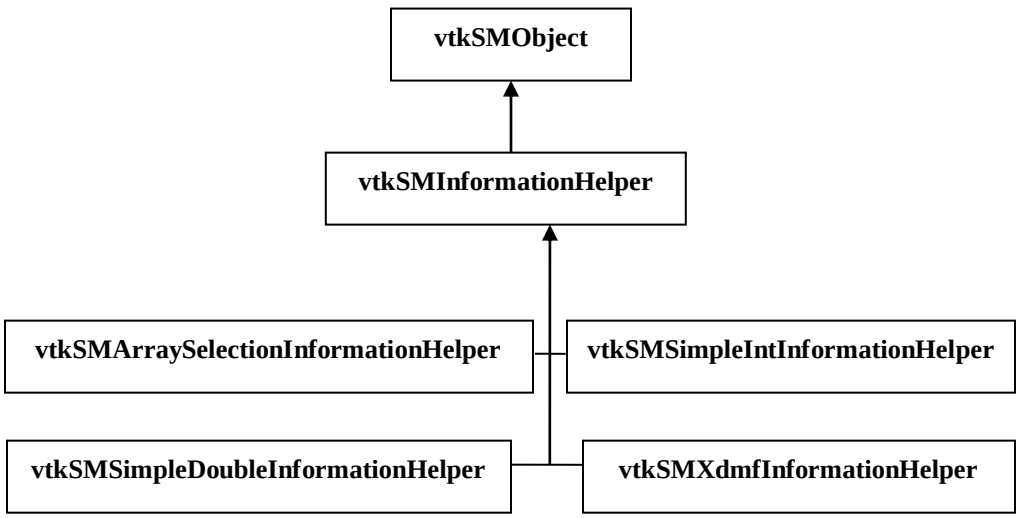


**Figure 3. Partial class hierarchy for information helpers**

## 1.4 Domains

`vtkSMDomain` is an abstract class that describes the domain of a property. A domain is a collection of possible values a property can have. Each domain can depend on one or more properties to compute its values. These are called *required* properties and can be set in the XML configuration file.

Properties are generic; there are only a few types of them. Alone, they are not sufficient to provide all the information related to a VTK object property. For example, a `vtkSMIntVectorProperty` can represent a single value from a list of enumeration values, or it can represent a vector of values, each of which is constrained between a minimum and a maximum value. The property itself has no way to differentiate between the two cases. This ambiguity is the reason for domains. Each property can have one or more domains described in its XML.

```
<InputProperty name="Input" command="AddInputConnection"
               clean_command="RemoveAllInputs"
               multiple_input="1">
  <ProxyGroupDomain name="groups">
    <Group name="sources"/>
    <Group name="filters"/>
  </ProxyGroupDomain>
  <DataTypeDomain name="input_type">
    <DataType value="vtkDataSet"/>
```

```
  </DataTypeDomain>
</InputProperty>
```

There are a large number of domain types: more than one corresponding to each property type. Domains have multiple uses as described below.

1. They provide more information about a property (e.g., enumeration values, range, etc.).

2. They restrict the values to which the property may be set.

The second use of domains is accomplished as follows. In `SetElement()` (or a similar method), the argument passed is copied into an "unchecked" element. Then `vtkSMProperty::IsInDomains()` is called. This method in turn calls `vtkSMDomain::IsInDomain()` on all domains passing the property (`this` pointer in C++) as an argument. The domains then each look at the unchecked element and return `1` if the value is acceptable according to this domain and `0` if not. If any of the domains do not contain the value of the unchecked element, the value is not copied to the actual element, and `0` is returned (from `SetElement()` or its equivalent). Otherwise, the value is copied to the actual element, and `1` is returned. The unchecked elements are never pushed to the server, and changing them does not modify the property.

Some domains are specialized to update their values based on properties. This happens as follows. Each domain may have one or more properties on which it depends (`RequiredProperties` in XML). This is demonstrated below.

```
<StringVectorProperty
  name="SelectInputScalars"
  command="SetInputArrayToProcess"
  number_of_elements="5"
  element_types="0 0 0 0 2">
  <ArrayListDomain name="array_list"
                   attribute_type="Scalars">
    <RequiredProperties>
      <Property name="Input" function="Input"/>
    </RequiredProperties>
  </ArrayListDomain>
</StringVectorProperty>
```

When a required property is added to domain, a two-way connection is made between the property and the domain.

```
void vtkSMDomain::AddRequiredProperty(vtkSMProperty *prop,
```

```
                                        const char *function)
{
  if (!prop)
    {
    return;
    }

  if (!function)
    {
    vtkErrorMacro("Missing name of function for new required
property.");
    return;
    }

  prop->AddDependent(this);
  this->Internals->RequiredProperties[function] = prop;
}
```

Then `vtkSMProperty::UpdateDependentDomains()` is called when the property value changes. An example follows.

```
void void vtkSMXYPlotRepresentationProxy::Update(
  vtkXMViewProxy* view)
{
  if (!this->ObjectsCreated)
    {
    vtkErrorMacro("Objects not created yet!");
    return;
    }
  this->Superclass::Update(view);
  vtkSMProxy *subProxy =
    this->GetSubProxy("DummyConsumer");
  vtkSMProxyProperty *pp =
    vtkSMProxyProperty::SafeDownCast(
      subProxy->GetProperty("Input");
  pp->UpdateDependentDomains();
}
```

**Note**: Do not confuse a property on which the domain depends with the property that actually contains that domain.

```
<InputProperty name="Input" command="AddInputConnection"
               clean_command="RemoveAllInputs"
```

```
                multiple_input="1">
  <ProxyGroupDomain name="groups">
    <Group name="sources"/>
    <Group name="filters"/>
  </ProxyGroupDomain>
  <DataTypeDomain name="input_type">
    <DataType value="vtkDataSet"/>
  </DataTypeDomain>
</InputProperty>

<StringVectorProperty
  name="SelectInputScalars"
  command="SetInputArrayToProcess"
  number_of_elements="5"
  element_types="0 0 0 0 2">
  <ArrayListDomain name="array_list"
                   attribute_type="Scalars">
    <RequiredProperties>
      <Property name="Input" function="Input"/>
    </RequiredProperties>
  </ArrayListDomain>
</StringVectorProperty>
```

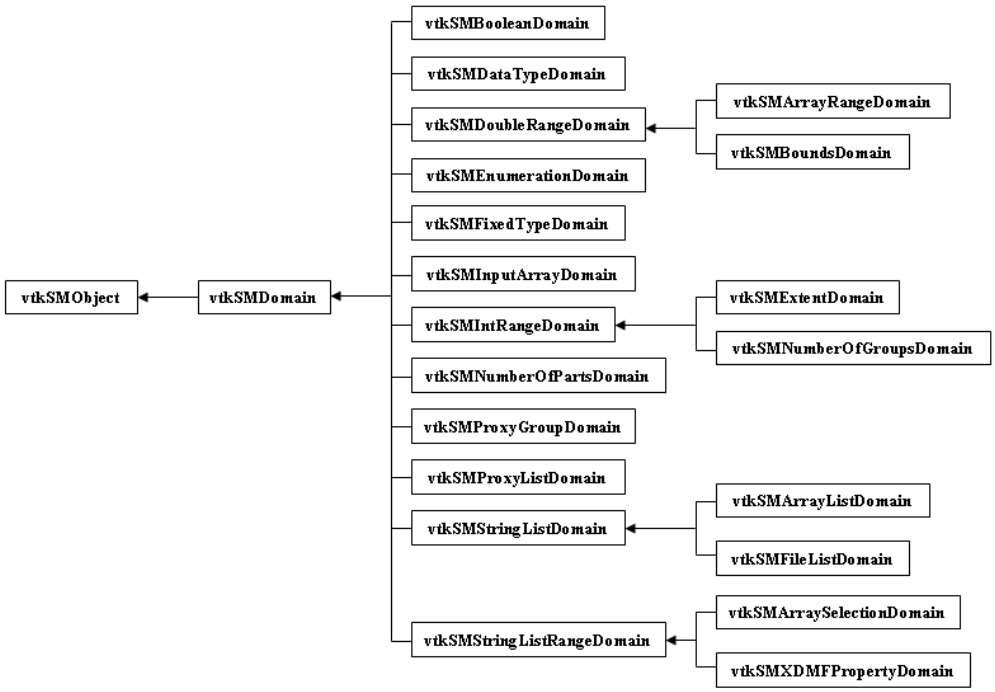


**Figure 4. Class hierarchy for domains**

## 1.5 Advanced Features

### vtkSMSourceProxy

`vtkSMSourceProxy` manages VTK source(s) that are created on a server using the proxy pattern. In addition to functionality provided by `vtkSMProxy`, `vtkSMSourceProxy` provides methods to connect and update sources. Each source proxy has one or more parts (`vtkSMPart`). Each part represents one output of one filter. These are created automatically when `vtkSMSourceProxy::CreateParts()` is called by the source. Each `vtkSMSourceProxy` creates a property called `DataInformation`. This property is a composite property that provides information about the output(s) of the VTK sources (obtained from the server).

`vtkSMInputProperty` is different from other properties in that it always calls methods on the source proxy that contains it, never on the server-side objects.

**Note**: By default, all input properties immediately push their values to the server. This is safest because much internal functionality depends on the input being set. (See the section

later in this chapter on multiple parts.) However, this causes a side-effect for multiple input filters; every time a new input is added to an input property, the following occurs.

It is possible to avoid this by setting `ImmediateUpdate` to `0` for each input for a multiple input filter and manually updating the filter using the `UpdateVTKObjects()` method of `vtkSMProxy` after all inputs have been added.

## 1.6 Server Manager XML

ParaView parses XML files to make the appropriate readers, writers, sources, and filters available within ParaView. The XML for ParaView's server manager determines how the values changed by various user interface controls affect the underlying sources, filters, etc. The server manager XML files are located in the `ParaView/Servers/ServerManager/Resources` directory. To obtain the class name of the server manager properties, domains, and information helpers discussed in this section, prepend `vtkSM` to the name listed in the text. These classes are located in the `ParaView/Servers/ServerManager` directory.

**`<ServerManagerConfiguration> </ServerManagerConfiguration>`**: These are the tags that begin and end each server manager XML file.

The type of the only direct sub-element of `ServerManagerConfiguration` is `ProxyGroup`. The `ProxyGroup` element has one attribute, `name`, which is used by the proxy manager to refer to a group of proxies. The valid value of this attribute for the `readers.xml` and `sources.xml` files is `"sources"`. The `filters.xml` file uses `"filters"`, the `rendering.xml` file uses `"rendering"`, and the `writers.xml` file uses `"writers"`. There are several proxy groups in `utilities.xml`; their names include `"lookup_tables"`, `"implicit_functions"`, `"transforms"`, `"matrices"`, and `"data_arrays"`.

### Readers, Sources, and Filters

Each reader, source, and filter is represented by a `SourceProxy` element (a sub-element of `ProxyGroup`). The `SourceProxy` element has the following three attributes.

- **`name`**: This is the unique identifier of this source proxy. It must match that attribute of the corresponding `Module` element in the user interface XML files. (See the previous section.)

- **`class`**: This is the name of the VTK class to be created.

- **`label`**: This is the text label to display in ParaView's user interface (e.g., the **Sources** and **Filters** menus).

Each user interface element for a source, filter, or reader typically corresponds to a single server manager property. These properties are sub-elements of `SourceProxy`. The following attributes are valid for all server manager properties. (The first two are required.)

- **`name`**: This is the name of this server manager property specified by the `property` attribute in the user interface XML. It is used to access the property from the proxy containing it. The corresponding instance variable in `vtkSMProperty` is `XMLName`.

- **`command`**: This attribute lists the name of the method to invoke on the reader, source, or filter containing this property. The corresponding instance variable in `vtkSMProperty` is `Command`.

- **`information_only`**: Use this attribute to specify that this property is for retrieving information instead of setting it (i.e., `information_only="1"`). It defaults to `"0"`. The corresponding instance variable in `vtkSMProperty` is `InformationOnly`; it defaults to `0`.

- **`information_property`**: List the name of a property (with `information_only` set to `"1"`) that is being used to obtain information for this property. The associated instance variable in `vtkSMProperty` is `InformationProperty`.

- **`immediate_update`**: If this attribute is set to `"1"`, the property value is passed to the server as soon as the property is modified. (This is determined by the value of `vtkCommand::ModifiedEvent`. **Warning**: If this attribute is set to `"1"`, *any* `ModifiedEvent` – regardless of whether it was caused by this reader, source, or filter – will cause the value of the property to be passed to the server.) This attribute's default value is `"0"`. The associated instance variable in `vtkSMProperty` is `ImmediateUpdate`; its default value is `"0"`.

- **`update_self`**: If this attribute is set to `"1"`, then the command is called on the proxy using this property rather than on the VTK object residing on the server. This can be used for exposing a method of the proxy through the property interface. The corresponding instance variable in `vtkSMProperty` is `UpdateSelf`; it defaults to `"0"`.

- **`animateable`**: The value of this attribute determines whether the property value can be animated. (See chapter **Error! Reference source not found.** for a discussion of animation in ParaView.) A value of `"0"` means that the property is not animateable; a value of `"1"` means that the property is animateable. The corresponding instance variable in `vtkSMProperty` is `Animateable`.

The four vector properties, `DoubleVectorProperty`, `IntVectorProperty`, `IdTypeVectorProperty`, and `StringVectorProperty`, each represent a list of one or more values of the type indicated by the property name. The common superclass of the four types of vector properties is `vtkSMVectorProperty`. These property types have several additional XML attributes in common.

- **`clean_command`**: This attribute specifies a command to remove all the values of a property. This attribute is typically used when `repeat_command` is set to `"1"`. (See below for a description of `repeat_command`.) If the `clean_command` attribute is set, this command is called before the main command (set using the `command` attribute).

- **`default_values`**: This attribute allows the user to specify default values for each element in this property. For example, for a `DoubleVectorProperty` containing three elements, setting this attribute to `"0.0 0.0 0.0"` would initialize all three of these elements to `0.0`.

- **`number_of_elements`**: The value of this attribute is the length of the list of values contained in this property.

- **`number_of_elements_per_command`**: This attribute determines the number of elements that should be passed as parameters of `command` each time command is called. For example, if this property with command `SetFoo` has four elements, `repeat_command="1"`, and this attribute has a value of `"2"`, then `command` would be called twice: once with the first two elements (e.g., `0` and `1`) and once with the second two (e.g., `5` and `9`) as shown below the description of `repeat_command`. The instance variable in `vtkSMVectorProperty` associated with this attribute is `NumberOfElementsPerCommand`; it defaults to `1`.

- **`repeat_command`**: If this attribute is set to `"1"`, then the command for this property should be called once per `number_of_elements_per_command` elements. This attribute's default value is `"0"`. The instance variable in `vtkSMVectorProperty` associated with this attribute is `RepeatCommand`; it defaults to `0`.

```
objID SetFoo 0 1
objID SetFoo 5 9
```

- **`use_index`**: If this attribute and `repeat_command` have values of `"1"`, then an integer index will be included as the first parameter to this command. For example, if the `command` is `SetValue`, there are four elements (`0.0`, `1.0`, `2.0`, and `3.0`), two per command, and `use_index` is set to `"1"`, then the `command` will be called

twice as shown below. The associated instance variable in `vtkSMVectorProperty` is `UseIndex`; it defaults to `0`.

```
objID SetValue 0 0.0 1.0
objID SetValue 1 2.0 3.0
```

Following is a description of the XML for each type of server manager property, including a description of the domain and information helper types used with each property. Both domains and information helpers are sub-elements of their respective properties.

Each domain type is associated with only one type of property. Domains represent the possible values a property may have. A property may have more than one domain. Some domains also have required properties on which they depend. (For example, a domain which lists the possible scalar arrays on which a filter operates may also depend on the `InputProperty` of that filter.) Any domain may use the attribute `optional` with a value of `"1"` to indicate that this domain provides information (e.g., to initialize the user interface) rather than to restrict the value of the property.

An information helper is only used with server manager properties whose `information_only` attribute is set to `"1"`. It gathers the required information from the server for the property with which it is associated.

### *DoubleVectorProperty*

The `DoubleVectorProperty` represents a list of one or more values of type `double`. It supports two additional XML attributes.

- **`argument_is_array`**: If set to `"1"`, the list of values is passed to the command as a single `double*`, not each value as an individual parameter. The associated instance variable in `vtkSMDoubleVectorProperty` is `ArgumentIsArray`; it defaults to `0`.

- **`set_number_command`**: If set, this attribute contains the name of a command to call before the one indicated by the `command` attribute. The value of the parameter passed to this additional command is `number_of_elements / number_of_elements_per_command`. An example `DoubleVectorProperty` that uses this attribute is shown below.

```
<DoubleVectorProperty
  name="ContourValues"
  command="SetValue"
  set number command="SetNumberOfContours"
  number of elements="0"
  repeat_command="1"
```

```
  number_of_elements_per_command="1"
  use_index="1">
</DoubleVectorProperty>
```

If the vector values for the above example are 5, 7, and 9.5, then the following methods are called.

```
contourId SetNumberOfContours 3
contourId SetValue 0 5
contourId SetValue 0 7
contourId SetValue 0 9.5
```

The `DoubleVectorProperty` supports the following types of domains and information helpers.

- **`ArrayRangeDomain`**: If the range of values for this variable depends on the range of values in a particular data array (i.e., a data set attribute), use the `ArrayRangeDomain`. The only attribute for this domain is `name`. It has two required properties: input (as described below for the `BoundsDomain`) and a `StringVectorProperty` associated with the array selection menu determining which array's range to use. The `function` for the `StringVectorProperty` must be `"ArraySelection"`.

```
<ArrayRangeDomain name="scalar_range">
  <RequiredProperties>
     <Property name="Input" function="Input"/>
     <Property name="SelectInputScalars"
               function="ArraySelection"/>
  </RequiredProperties>
</ArrayRangeDomain>
```

- **`BoundsDomain`**: To appropriately set some variables, it is necessary to know information about the bounds (an axis-aligned bounding box) of the data set being manipulated. To provide this information, we use a `BoundsDomain`. This domain has three attributes: `name`, `mode`, and `scale_factor`. The possible values of the mode attribute are `"normal"` (i.e., the values of the domain are the bounds of the data set) and `"magnitude"` (i.e., the values are the diagonal of the bounding box and its negative). The default value is `"normal"`.

  To use this domain, an `InputProperty` must also be specified. Because it is a required property for this domain, it must be enclosed in `<RequiredProperties>` `</RequiredProperties>` tags as shown below.

The value of the `name` attribute must match that of the `InputProperty` of the filter being described. The `function` attribute is used in looking up a particular required property. For the `BoundsDomain`, its value must be `"Input"`.

```
<BoundsDomain name="bounds">
  <RequiredProperties>
     <Property name="Input" function="Input"/>
  </RequiredProperties>
</BoundsDomain>
```

- **`DoubleRangeDomain`**: This domain provides the minimum and/or the maximum value of this property. Its attributes are `name`, `min`, and `max`.

- **`SimpleDoubleInformationHelper`**: This information helper calls its property's command on the appropriate server and uses the values returned to populate the property. It does not have any XML attributes.

- **`TimeStepsInformationHelper`**: This information helper retrieves the time step values (using `vtkPVServerTimeSteps`) from the server and uses the returned values to populate the `DoubleVectorProperty`. Server helper objects such as vtkPVServerTimeSteps are discussed in section **Error! Reference source not found.**.

### *IntVectorProperty and IdTypeVectorProperty*

The `IntVectorProperty` represents a list of one or more values of type `int`. (`IdTypeVectorProperty` should be used instead when the number of bytes in the integer value is unknown.) It supports the following additional XML attribute.

- **`argument_is_array`**: If set to `"1"`, the list of values is passed to the command as a single `int*`, not each value as an individual parameter. The associated instance variable in `vtkSMIntVectorProperty` (or `vtkSMIdTypeVectorProperty`) is `ArgumentIsArray`; it defaults to `0`.

It supports the following domains and information helpers.

- **`BooleanDomain`**: Use this domain if this `IntVectorProperty` can only have the values `"0"` or `"1"` (e.g., if this property is associated with a toggle button). The only XML attribute for this domain is `name`.

- **`EnumerationDomain`**: An `EnumerationDomain` contains a list of acceptable integer values for this property and an associated text string for each value. An `Entry` sub-element for each value/text pair must be provided. An example of the XML for doing this is shown below.

```
<EnumerationDomain>
  <Entry value="0" text="X Min"/>
  <Entry value="1" text="Y Min"/>
  <Entry value="2" text="Z Min"/>
  <Entry value="3" text="X Max"/>
  <Entry value="4" text="Y Max"/>
  <Entry value="5" text="Z Max"/>
</EnumerationDomain>
```

- **`ExtentDomain`**: This domain can be used with structured data sets to provide information about the extents of the data set. This domain only has a `name` attribute. It is required to have an `InputProperty`. The XML specification for this is described in the section about the `BoundsDomain` of the `DoubleVectorProperty`.

- **`IntRangeDomain`**: This domain provides the minimum and/or the maximum value of this property. Its attributes are `name`, `min`, and `max`.

- **`SimpleIntInformationHelper`**: This information helper calls its property's command on the appropriate server and uses the values returned to populate the property. It does not have any XML attributes.

### *StringVectorProperty*

`StringVectorProperty` represents a list of text strings. This property is also used when the parameters to a function do not all have the same data type. It supports the following attributes, domains, and information helpers.

- **`element_types`**: If an instance of `StringVectorProperty` will have non-string elements, this attribute allows you to specify which type(s) to use. To specify multiple elements, provide a space-separated list. A `StringVectorProperty` can handle elements of type int (`="0"`), double (`="1"`), and string (`="2"`). For example, if the parameters to your `command` are two strings followed by an int, this attribute would be set to `"2 2 0"`. If the element type is not string, it is converted to a string before it is sent to the server(s). This is most commonly used for commands that have a hybrid list of arguments. For example,

```
void vtkArrayCalculator::AddScalarVariable(
     const char* variableName,
     const char* arrayName,
     int component);
can be accessed through XML as follows.
<StringVectorProperty
  name="AddScalarVariable"
```

```
  command="AddScalarVariable"
  number of elements="3"
  repeat command="1"
  number_of_elements_per_command="3"
  element_types="2 2 0" />
```

Note that converting a floating-point number to a string will cause a loss of precision; try to avoid using `StringVectorProperty` in this case.

- **`ArrayListDomain`**: The `ArrayListDomain` contains a list of arrays obtained from the input to the filter using this property. The attributes supported by this domain are `name`, `attribute_type` (one of `"Scalars"`, `"Vectors"`, `"Normals"`, or `"TCoords"`, indicating the attribute type of the array names to include), and `input_domain_name` (lists the name of the appropriate `InputArrayDomain` if the input has more than one of them).

  The `ArrayListDomain` also has a required property: an input property.

```
<ArrayListDomain name="array_list"
                 attribute_type="Scalars">
  <RequiredProperties>
     <Property name="Input" function="Input"/>
  </RequiredProperties>
</ArrayListDomain>
```

- **`ArraySelectionDomain`**: An `ArraySelectionDomain` lists array names that are valid text strings for this property. Its only attribute is `name`. Its required property is a `StringVectorProperty` (an information property) from which it retrieves the list of array names. This domain is used when selecting which data arrays to load from a file.

- **`FieldDataDomain`**: This domain provides information about whether any valid arrays in the data set are point- (`"Point Data"`) or cell-centered (`"Cell Data"`). This domain only has a `name` attribute. It is required to have an `InputProperty`. The XML specification for this is described in the section about the `BoundsDomain` of the `DoubleVectorProperty`.

- **`StringListDomain`**: As the name implies, this type of domain contains a list of text strings that are valid values for this property. It may include a required `StringVectorProperty` from which to obtain the list of strings. It is required to have a `name` attribute.

- **`XDMFPropertyDomain`**: This domain contains a list of strings for use with the XDMF reader. It supports the `name` attribute. It has a required `StringVectorProperty` from which it obtains this information. That property must have an `XDMFInformationHelper` (described below).

- **`FileListDomain`**: The `FileListDomain` provides a list of file names that are valid values for the associated `StringVectorProperty`. Its only attribute is `name`.

- **`ArraySelectionInformationHelper`**: This information helper retrieves information from the server about the names of the attribute arrays contained in the data set being loaded by the reader using this property. (This information helper is only used with readers, not sources or filters.) The only attribute supported by this information helper is `attribute_name`. The acceptable values of `attribute_name` are `"Cell"` and `"Point"`, indicating cell-centered or point-centered data arrays.

- **`SimpleStringInformationHelper`**: This information helper calls its property's command on the appropriate server and uses the values returned to populate the property. It does not have any XML attributes.

- **`XDMFInformationHelper`**: The XDMFInformationHelper obtains XDMF parameters from the data server. This information is propagated to the `XDMFPropertyDomain`. This information helper does not require any XML attributes.

### *ProxyProperty*

The `ProxyProperty` represents pointer(s) to object(s). This property supports the `ProxyGroupDomain` and the `ProxyListDomain`.

- **`ProxyGroupDomain`**: This domain contains a list of proxy groups on which this property operates. Its only attribute is `name`. It contains one sub-element called `Group` per appropriate proxy group. The only attribute of the `Group` element is its `name`, which must match the name of the proxy group containing proxies appropriate for the task performed by this property.

```
<ProxyGroupDomain name="groups">
  <Group name="sources"/>
  <Group name="filters"/>
</ProxyGroupDomain>
```

- **ProxyListDomain**: A `ProxyListDomain` provides a list pf proxy types that are valid values for the associated `ProxyProperty`. Its only attribute is `name`. Each of its sub-elements lists the `group` and `name` of a valid proxy type for the `ProxyProperty`.

### *InputProperty*

The `InputProperty` represents the input to a filter. This property supports the following attributes and domains in addition to the `ProxyGroupDomain` (described above). This domain can be used by two properties because it is defined for `vtkSMProxyProperty`, the superclass of `vtkSMInputProperty`.

- **clean_command**: The value of this attribute is a command to remove the inputs to the filter using this `InputProperty`. (The name of the command is passed as the parameter to the `vtkSMSourceProxy::CleanInputs` method.) If this attribute is set, the `clean_command` will be called before the `command` is called.

- **multiple_input**: Setting the value of this attribute is set to `"1"` indicates that the filter using this `InputProperty` accepts multiple inputs (e.g., the **Append** filter or the **Stream Tracer** filter). The instance variable in `vtkSMInputProperty` associated with this attribute is `MultipleInput`; it defaults to `0`.

- **DataTypeDomain**: This domain is used to specify which data set types are appropriate for this `InputProperty`. Its only attribute is `name`. It has one sub-element called `DataType` per data set type; its only attribute is `value`, which indicates the name of the VTK class for that data set type. If a filter can accept any data set type (except `vtkMultiGroupDataSet` and its subclasses) as input, value should be set to `"vtkDataSet"`.

```
<DataTypeDomain name="input_type">
  <DataType value="vtkImageData"/>
  <DataType value="vtkRectilinearGrid"/>
  <DataType value="vtkStructuredPoints"/>
  <DataType value="vtkStructuredGrid"/>
</DataTypeDomain>
```

- **FixedTypeDomain**: This domain is necessary when a filter accepts multiple types of data sets as input, but the type cannot change once it has been initially set. Its only attribute is its `name`.

- **InputArrayDomain**: The `InputArrayDomain` requires that an input to this filter must have one or more attribute arrays. Besides the `name` attribute, this domain

also supports attribute_type ("point" or "cell" to indicate whether the arrays must be point- or cell-centered), number_of_components, and optional. Neither attribute_type nor number_of_components are required in all cases. Additionally, this domain may have a required IntVectorProperty (like that used for the ArrayListDomain) to determine whether the arrays are point- or cell-centered.

- **NumberOfGroupsDomain**: The NumberOfGroupsDomain restricts whether the filter using this property operates on multi-group data sets containing only on a single data set or containing only a group of data sets. In addition to the name attribute, this domain also uses an attribute called multiplicity. The possible values for multiplicity are "single" and "multiple".

- **NumberOfPartsDomain**: The NumberOfPartsDomain restricts whether the filter using this property operates only on a single input or only on a group of inputs (a multi-part data set). In addition to the name attribute, this domain also uses an attribute called multiplicity. The possible values for multiplicity are "single" and "multiple".

In addition to various Property elements, Proxy elements can also have SubProxy elements. The exposed properties of the subproxy can be accessed by calling the GetProperty method on the parent proxy. The SubProxy element must contain a Proxy sub-element, as demonstrated in the following excerpt from rendering.xml.

```
<PVRepresentationProxy
   name=”UnstructuredGridRepresentation”
   base_proxygroup=”representation”
   base_proxyname=”PVRepresentationBase”>
...
   <SubProxy>
      <Proxy name=”VolumeRepresentation”
        proxygroup=”representations”
        proxyname=”UnstructuredGridVolumeRepresentation”/>
      <ShareProperties subproxy=”SurfaceRepresentation”>
         <Exception name=”Input”/>
         <Exception name=”Visibility”/>
      </ShareProperties>
      <ExposedProperties>
         <Property name=”ScalarOpacityFunction”/>
         <Property name=”ScalarOpacityUnitDistance”/>
      </ExposedProperties>
   </SubProxy>
</PVRepresentationproxy>
```

If the subproxy has been defined elsewhere (in this or another server manager XML file), then you can reference that `Proxy` definition rather than duplicating it, as demonstrated in the above example. There are three required attributes to the `Proxy` sub-element of the `SubProxy` element.

- **`name`**: This attribute identifies the subproxy and must be unique within the parent `Proxy` element.

- **`proxygroup`**: This attribute must match the `name` attribute of the `ProxyGroup` element within which the `Proxy` element we are referencing is contained.

- **`proxyname`**: This attribute lists the name attribute of the `Proxy` element being referenced.

If a particular parent Proxy element has more than one subproxy, and the values of particular properties should always be the same for two subproxies, then the `ShareProperties` sub-element of the `SubProxy` element should be used. Any properties between these two proxies whose values should not necessarily match should be listed using an `Exception` element, as shown above. In our example, all the properties of `VolumeRepresentation` and `SurfaceRepresentation` should match except the `Input` and `Visibility` properties.

Only the properties listed as `ExposedProperties` will be accessible by calling `GetProperty` on the parent proxy. In the above example, the `ScalarOpacityFunction` and `ScalarOpacityUnitDistance` properties of the `VolumeRepresentation` subproxy are exposed.

### Writers, Rendering, and Utilities

The writers.xml, rendering.xml, and utilities.xml files provide XML descriptions of proxies, properties, etc., used in ParaView's file writing, rendering, and various other capacities. Several new proxies (new subclasses of `vtkSMProxy`) are introduced as needed, but the same properties, domains, and information helpers are used as in the previous section. The `Proxy` elements corresponding to the various proxies used are sub-elements of several different `ProxyGroup` elements.