SCIRun Module Generation

SCIRun 5.0 Documentation

Center for Integrative Biomedical Computing
Scientific Computing & Imaging Institute
University of Utah

SCIRun software download:

http://software.sci.utah.edu

Center for Integrative Biomedical Computing:

http://www.sci.utah.edu/cibc

This project was supported by grants from the National Center for Research Resources **(5P41RR012553-14)** and the National Institute of General Medical Sciences **(8 P41 GM103545-14)** from the National Institutes of Health.

Author(s):
Jess Tate

Contents

Chapter 1

SCIRun Overview

This tutorial demonstrates how to create new modules in SCIRun 5.0 . It will walk through all the files needed and the basic module structure used by modules. These instructions assume a basic understanding in C++ coding and other basic programing skills

1.1 Software requirements

1.1.1 SCIRun 5.0

Download SCIRun version 5.0 from the SCI software portal. Make sure to update to the most up-to-date version of the source code available, which will include the latest bug fixes. Alternatively, use git to clone the SCIRun repository (https://github.com/SCIInstitute/SCIRun.git). We suggest creating a fork of the repository so that you can track your changes and create pull requests to the SCIRun repository (Section 1.1.3).

1.1.2 Compilers, Dependencies Development Tools

SCIRun will need to be built from the source code in order to test and use any modules written. Make sure that qt 4.8, git, cmake, and the latest c++ compilers for the operating system are installed.

1.1.3 Creating Your SCIRun Fork

With your own github account, go to the SCIRun github page. Click the fork button on the upper right side of the page. It will ask you where to move the fork to, chose your own account. Once the repository is forked, clone it to your local machine with the following command.

```
$ git clone https://github.com/[yourgithubaccount]/SCIRun.git
```

After the the code is cloned, navigate to the repository directory and add the upstream path to the original SCIRun repository.

```
$ git remote add upstream https://github.com/SCIInstitute/SCIRun.git
```

You should be able to see both your and the original repositories when you use the command:

```
$ git remote -v
```

The fork is good to go, but you will need to sync the fork occationally to keep up with the changes in the main repository. To sync your fork, use the following commands:

```
$ git fetch upstream
$ git checkout master
$ git merge upstream/master
```

You should sync and merge your fork before you start a new module and before you create a pull request. It is a good practice to create a new branch in your fork for every module you will be adding. The command to create a new branch is:

```
$ git checkout -b [branch_name]
```

Please see the github help page for more information.

Chapter 2

Files Needed for a New Module

This chapter will describe the files need to create a module in SCIRun.

Scope: Overview of Files Needed for each Module - Module Configuration File - Module Source Code - Algorithm Code - Module UI Code

2.1 Overview of Files Needed for each Module

There are only three files required to create a module, though more may be needed depending on the function of the module. In addition to the required module source code and header files (*modulename.cc* and *modulename.h*), a module configuration file is needed. The module configuration file (*modulename.module*) contains a description of the module and its state and names all the files needed for the module to be included in SCIRUn.

Simple modules without user interfaces (UIs) can be created with the previously list three files alone. However, if the module function needs a UI, there are three additional files needed. SCIRun can generate a UI for a module without these, but the functionality will be very limited to nonexistent. The qt ui file (*modulenameDialog.ui*) describes the graphics and hooks of the UI can be created using the qt UI editor. Module UIs also require a code and header file (*modulenameDialog.cc* and *modulenameDialog.h*.

Most modules, especially those requiring more than minimal code, should also have algorithm code to allow for greater portability and code control. This algorithm code and header file (*modulenameAlgo.cc* and *modulenameAlgo.h*) should contain all the computation of the module. Though it is possible to build modules without these algorithm files, it is considered good practice to do so.

It is worth noting that each of the *CMakeLists.txt* files in the directories of all of the files (except the module config file). See the examples in the following chapters for details.

2.2 Module Configuration File

The module configuration file contains all the information need for the module factory to create necessary linkage and helper files for module to be properly included into SCIRun. Module configuration files should be located in *src/Modules/Factory/Config/*. It is a text file that describes fields specific to the module delimited by curly brackets. There are three fields “module”, “algorithm”, and “UI” and within each field are subfields “name” and

"header", and others depending on the field. The following is an example that reflects the template files included in the source code.

```
{
  "module": {
    "name": "@ModuleName@",
    "namespace": "Fields",
    "status": "description of status",
    "description": "description of module",
    "header": "Modules/Template/ModuleTemplate.h"
  },
  "algorithm": {
    "name": "@AlgorithmName@Algo",
    "namespace": "Fields",
    "header": "Core/Algorithms/Template/AlgorithmTemplate.h"
  },
  "UI": {
    "name": "@ModuleName@Dialog",
    "header": "Interface/Modules/Template/ModuleDialog.h"
  }
}
```

This config file example would not build. We will include specific examples that will build and work in following chapters of this tutorial (Chapters—).

As mentioned before, the UI and algorithm files are not required to generated a module, therefore the subfields for the "algorithm" or "UI" fields can changed to "N/A" to indicate that these files do not exist. Please refer to Section **??** for an example.

2.3 Module Source Code

The Module source code consist of a .cc and .h file that code the actual function of the module. However, since most modules use an algorithm file, these files can also be considered the the code that pulls all the relevant information from the algorithm, the UI, and other modules in order to achieve it's proper function. These files should be located in the proper directory with the *src/Modules/* directory. For example purposes, we will show and discuss the template files included in the *src/Modules/Template/* directory.

2.3.1 Module Header File

The module header file functions as a typical C++ header file, containing code establishing the module object and structure. The *ModuleTemplate.h* file found in *src/Modules/Template/* provides an example of the kind of coding needed for a module header. The relevant functions are included here, with annotated comments:

```
// makes sure that headers aren't loaded multiple times.
// This requires the string to be unique to this file.
// standard convention incorporates the file path and filename.
#ifndef MODULES_FIELDS_@ModuleName@_H__
```

```
#define MODULES_FIELDS_@ModuleName@_H__

#include <Dataflow/Network/Module.h>
#include <Modules/Fields/share.h>
// share.h must be the last include, or it will not build on windows systems.

namespace SCIRun {
namespace Modules {
namespace Fields {
// this final namespace needs to match the .module file
// in src/Modules/Factory/Config/

  // define module ports.
  // Can have any number of ports (including none), and dynamic ports.
  class SCISHARE @ModuleName@ : public SCIRun::Dataflow::Networks::Module,
    public Has1InputPort<FieldPortTag>,
    public Has1OutputPort<FieldPortTag>
  {
  public:
    // these functions are required for all modules
    @ModuleName@();
    virtual void execute();
    virtual void setStateDefaults();

    //name the ports and datatype.
    INPUT_PORT(0, InputField, Field);
    OUTPUT_PORT(0, OutputField, Field);

    // this is needed for the module factory
    static const Dataflow::Networks::ModuleLookupInfo staticInfo_;
  };
}}}
#endif
```

One of the key functions of this header file is the definition of the ports used by the module. This template uses one input and output, but any number can be used by changing the number and defining all the port types. To use two inputs and outputs:

```
public Has2InputPorts<FieldPortTag,FieldPortTag>,
public Has2OutputPorts<FieldPortTag,FieldPortTag>
```

If no there are no input or output ports, the commands are:

```
public HasNoInputPorts,
public HasNoOutputPorts
```

Dynamic ports are also possible for the inputs. Dynamic ports are essentially a vector of ports, and are counted as a single port in the header. For a single dynamic port, then a static port and a dynamic port:

```
public Has1InputPort<DynamicPortTag<FieldPortTag>>,
public Has2InputPorts<FieldPortTag,DynamicPortTag<FieldPortTag>>
```

List of port tags that can be used in SCIRun:

- MatrixPortTag
- ScalarPortTag
- StringPortTag
- FieldPortTag
- GeometryPortTag
- ColorMapPortTag
- BundlePortTag
- NrrdPortTag
- DatatypePortTag

2.3.2 Module Code File

```
#include <Modules/Fields/@ModuleName@.h>
#include <Core/Datatypes/Legacy/Field/Field.h>
#include <Core/Algorithms/Field/@ModuleName@Algo.h>

using namespace SCIRun::Modules::Fields;
using namespace SCIRun::Core::Datatypes;
using namespace SCIRun::Dataflow::Networks;
using namespace SCIRun::Core::Algorithms::Fields;

// this defines the location of the module in the module list.
// "NewField" is the catagory and "SCIRun" is the package.
const ModuleLookupInfo @ModuleName@::staticInfo_("@ModuleName@",
          "NewField", "SCIRun");

@ModuleName@::@ModuleName@() : Module(staticInfo_)
{
  //initialize all ports.
  INITIALIZE_PORT(InputField);
  INITIALIZE_PORT(OutputField);
}

void @ModuleName@::setStateDefaults()
{
  auto state = get_state();
  setStateBoolFromAlgo(Parameters::Knob1);
  setStateDoubleFromAlgo(Parameters::Knob2);
```

```
}

void @ModuleName@::execute()
{
  // get input from ports
  auto field = getRequiredInput(InputField);
  // get parameters from UI
  setAlgoBoolFromState(Parameters::Knob1);
  setAlgoDoubleFromState(Parameters::Knob2);
  // run algorithm code.
  auto output = algo().run(withInputData((InputField, field)));
  //send to output port
  sendOutputFromAlgorithm(OutputField, output);
}
```

2.4 Algorithm Code

2.5 Module UI Code

2.5.1 Module Design File

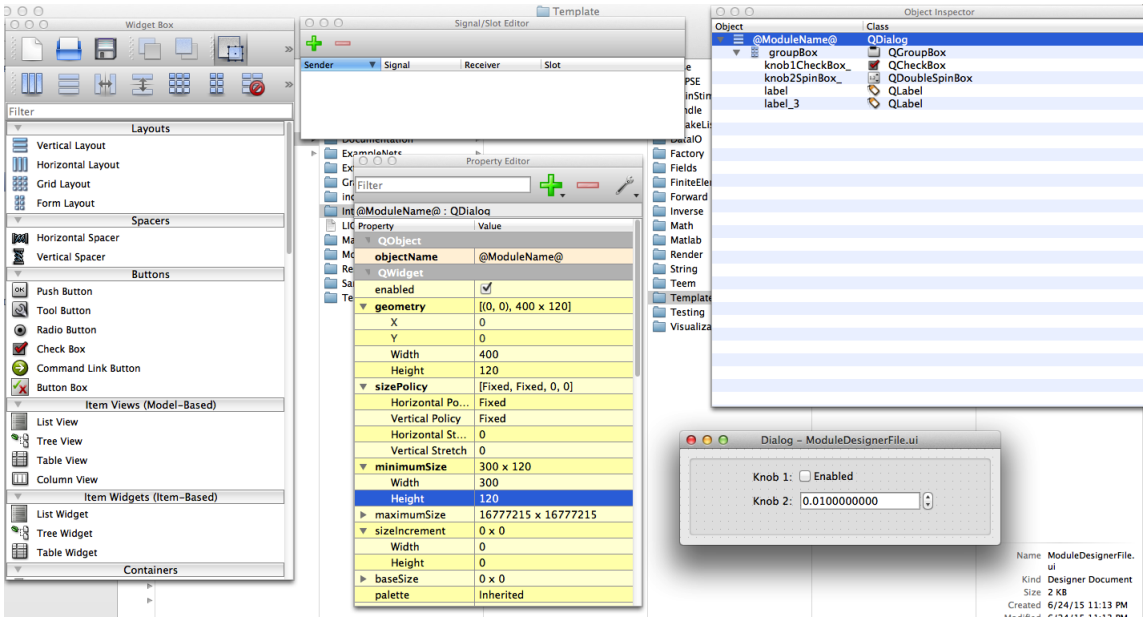

Figure 2.1. Template module interface design file as seen in the Qt editor.

Chapter 3

Example: Simple Module Without UI

This chapter describes how to create a very simple module in SCIRun. We will show how to make a simple module that outputs a simple string. This example will show the basics of the functions and code used by SCIRun to create and run modules.

If you have created a fork from the SCIRun git repository, begin by creating a new branch in your repository. Be sure to commit your changes to your repository often, as this can help you and the developers fix and make improvements to the code. It is often easiest to modify existing code to fit your purposes than create new code, so determine a module that has similar functionality or structure to the new module. If desired, there is also the template files described in Chapter 2 to use as a basis. In this example, we will provide the code needed, so it is not necessary to copy another module.

Begin with the module config file. Create a new text file in module factory configuration directory (*src/Modules/Factory/Config/*) for the new module. I should be named *TestModuleSimple.module* or something similar. The text of the file should be:

```
{
  "module": {
    "name": "TestModuleSimple",
    "namespace": "StringManip",
    "status": "new module",
    "description": "This is a simple module to show how to make new modules.",
    "header": "Modules/String/TestModuleSimple.h"
  },
  "algorithm": {
    "name": "N/A",
    "namespace": "N/A",
    "header": "N/A"
  },
  "UI": {
    "name": "N/A",
    "header": "N/A"
  }
}
```

The exact text of the status and description can be whatever the creator desires. The names of the module and filenames can also be different, but they must match the module code.

Now we move to the module code. The module will need to be placed in one of the directories in *src/Modules/*, so choose the directory that fits the modules use the best (do not place the module code in *Factory* or *Template*, and *Legacy* is generally for converted modules from earlier versions of SCIRun) Since this module will be only have a string output, we will place the module code in *src/Modules/String/*. In this directory create a file called *TestModuleSimple.h*. This file will be very similar to the *ModuleTemplate.h* file shown earlier. In addition to the SCIRun license information, the content of the header file should be:

```
#ifndef MODULES_STRING_TestModuleSimple_H
#define MODULES_STRING_TestModuleSimple_H

#include <Dataflow/Network/Module.h>
#include <Modules/Fields/share.h>

namespace SCIRun {
namespace Modules {
namespace StringManip {

class SCISHARE TestModuleSimple : public SCIRun::Dataflow::Networks::Module,
public HasNoInputPorts,
public Has1OutputPort<StringPortTag>
{
public:
  TestModuleSimple();
  virtual void execute();
  virtual void setStateDefaults() {};

  OUTPUT_PORT(0, OutputString, String);

  static const Dataflow::Networks::ModuleLookupInfo staticInfo_;
};
}}}
#endif
```

As mentioned in Section 2.2, the header files for most modules do not vary significantly. This example in particular contains only elements common to most other modules. The key to creating the header files is to ensure that the module name is correct in every place it occurs, that the namespace (StringManip) matches the module config file and that the ports are numbered and labeled correctly.

The final file needed for this module is the source code file (*ModuleTemplate.cc*). The functionality used in this module is minimal to show how the essential functions. With the license and other comments, the file should contain:

```
#include <Modules/String/TestModuleSimple.h>
#include <Core/Datatypes/String.h>

using namespace SCIRun;
```

```
using namespace SCIRun::Modules::StringManip;
using namespace SCIRun::Core::Datatypes;
using namespace SCIRun::Dataflow::Networks;

/// @class TestModuleSimple
/// @brief This module splits out a string.

const ModuleLookupInfo TestModuleSimple::staticInfo_("TestModuleSimple",
       "String", "SCIRun");

TestModuleSimple::TestModuleSimple() : Module(staticInfo_,false)
{
  INITIALIZE_PORT(OutputString);
}

void
TestModuleSimple::execute()
{
  std::string message_string;

  message_string = "[Personalize your  message here.]";

  StringHandle msH(new String(message_string));
  sendOutput(OutputString, msH);
}
```

After these files are modified correctly, the only step remaining before building is adding the module code and header to the compiler list. Open the *src/Modules/String/CMakeLists.txt* file. Add *TestModuleSimple.cc* and *TestModuleSimple.h* to the respective list. There will more to the file, but the relevant sections should look something like this:

```
SET(Modules_String_SRCS
  CreateString.cc
  NetworkNotes.cc
  TestModuleSimple.cc
)

SET(Modules_String_HEADERS
  CreateString.h
  NetworkNotes.h
  share.h
  TestModuleSimple.h
)
```

After changing the CMakeList.txt file, build SCIRun using the build script, or if you have already built SCIRun recently, go to the *SCIRun_root/bin/SCIRun* directory and run make. Take note of any build errors, if there is a problem in the with any module factory files, make sure that there are no mistakes in the the module configuration file and build again. Check out the common build errors in section [TODO].

After SCIRun builds completely, Launch SCIRun and test the module. You can use the PrintDatatype module to view the string that this module outputs. Other modules will require more testing, but due to the very simple nature of the module you can know that if the messages matches what you expect, then it is working properly.

Chapter 4

Example: Simple Module With UI

This example will build off of the previous example. In this chapter we will show how to add an input port and a UI to an existing module.

In this chapter, we will build off the module that we described in the previous chapter to show how to add a UI and an input port. This module will print a message that comes from either the input port or the UI. We will show how to add a UI incrementally to help convey the principles that the software is based upon. This incremental approach allows the user to copy this approach with more complicated module as it provides sanity checks for the user.

To begin, copy the *TestModuleSimple.module* in the *src/Modules/Factory/Config/* and name the copy *TestModuleSimpleUI.module*. Change the name and header field to reflect the new name of the module, as shown here:

```
"module": {
  "name": "TestModuleSimpleUI",
  "namespace": "StringManip",
  "status": "new module",
  "description": "This is a simple module to show how to make new modules.",
  "header": "Modules/String/TestModuleSimpleUI.h"
},
```

For now, leave the rest of the fields as 'N/A'; we will come back to those.

Next, copy the module code files *TestModuleSimple.h* and *TestModuleSimple.cc* in the *src/Modules/String/* directory and rename them appropriately (*TestModuleSimpleUI.h* and *TestModuleSimpleUI.cc*). In these new files, change all the references of the modules name to TestModuleSimpleUI. A find and replace function will manage most instances, but make sure that all of them are changed. There are 4 lines in each of the two files that need to be changed, with more than one change in some lines. The changes in *TestModuleSimpleUI.h* are these lines:

```
#ifndef MODULES_STRING_TestModuleSimpleUI_H
#define MODULES_STRING_TestModuleSimpleUI_H

...
```

```
class SCISHARE TestModuleSimpleUI : public SCIRun::Dataflow::Networks::Module,

...

public:
  TestModuleSimpleUI();

...
```

And for the *TestModuleSimpleUI.cc* file:

```
#include <Modules/String/TestModuleSimpleUI.h>



const ModuleLookupInfo TestModuleSimpleUI::staticInfo_("TestModuleSimpleUI",
       "String", "SCIRun");

TestModuleSimpleUI::TestModuleSimpleUI() : Module(staticInfo_)



void
TestModuleSimpleUI::execute()
{
```

Another change you may notice is to remove the 'false' input in the constructor line:

```
TestModuleSimpleUI::TestModuleSimpleUI() : Module(staticInfo_)
```

The 'false' options means that there is no module. Removing the options changes the input to 'true', which allows for a module UI. If no UI file is found, a default UI will be used.

With these changes, we should try to build. Make sure the files are added to the CMakeList.txt file in *src/Modules/String/* as we showed in the previous chapter. If there are build errors, check for spelling mismatches. Also, check out the common build errors in section [TODO]. Once SCIRun is built, you can try to add the new module to the workspace. SCIRun will give you a warning dialogue about not finding a UI file, so it will create a default one. This UI is not connected to anything, so it won't affect the module at all, but you should be able to open the UI and see it (a slider and two buttons). Check to make sure that the output is still the string that you expected. If everything is working properly, we can move onto the next step of adding our own module.

To create a new UI, we need to add three new files: a design file, and a cc and header file for the UI. We will need these files linked to the other module code, so we will modify the module config file again to add the name of the UI and the path to the header file. The naming convention often used is to add 'Dialog' to the end of the module name for the name of the UI and the names of the files.

```
"UI": {
    "name": "TestModuleSimpleUIDialog",
    "header": "Interface/Modules/String/TestModuleSimpleUIDialog.h"
  }
```

Next, we need to use the QT editor to design a module UI. Copy the QT module file from *src/Interface/Modules/Template/ModuleDesignerFile.ui* to *src/Interface/Modules/String/TestModuleSimp* Open the *TestModuleSimpleUIDialog.ui* file in the Qt editor, which provides a graphic method for modifying and compiling the design file. First, delete the check box and input scroll wheel widgets and delete the 'knob 2' label. Next, add a *line edit widget* by finding it in the *Widgdet Box* window (in the *Input Widget* section) and clicking and dragging it next to the remaining label. Change the text of the remaining label to 'My String:' or something similar. Finally, change the name of the of the *line edit widget* to *inputstring_* and the name of the *QDialog* object to *TestModuleSimpleUIDialog*. This can be done in the *Object Inspector* or in the *Property Editor* when the appropriate object is clicked. Figure 4.1 shows what the module should look like in the Qt editor.

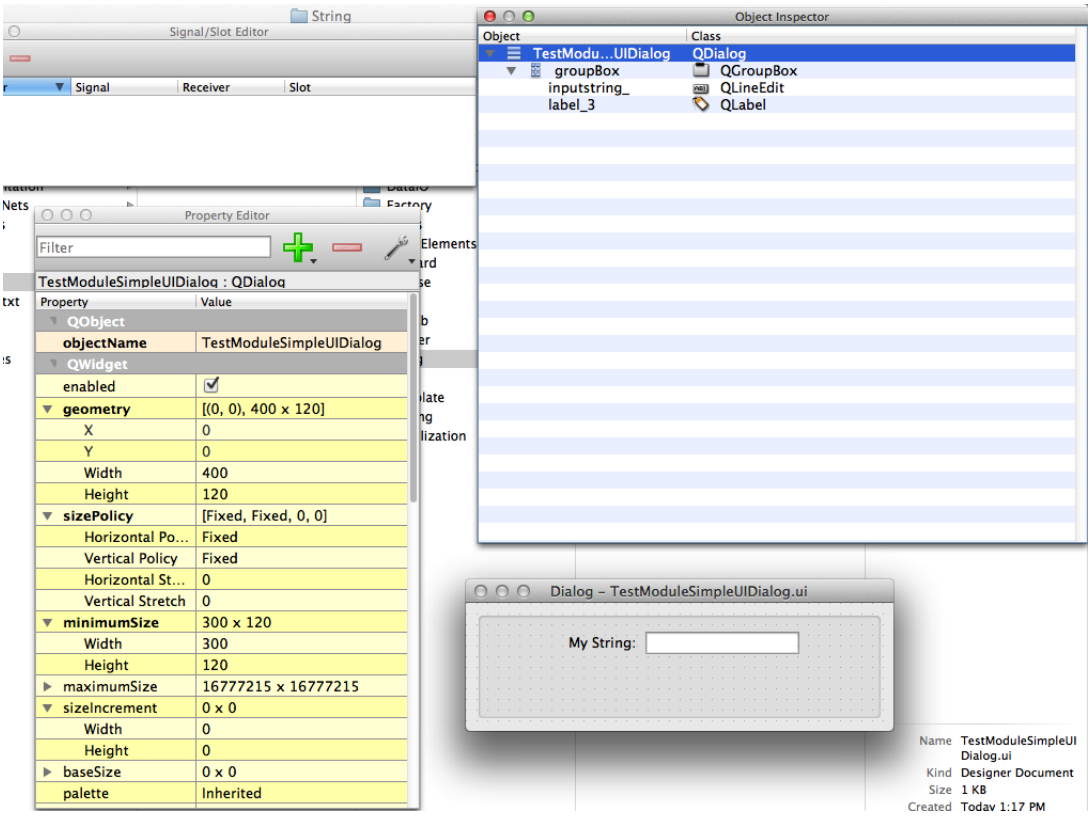

Figure 4.1. Module interface design file for the TestModuleSimpleUI module as seen in the Qt editor.

Now that the module UI is designed, we need to link it to the module with the module dialog code. Copy the *ModuleDialog.cc* and the *ModuleDialog.h* from the *src/Interface/Modules/Template/* directory to the *src/Interface/Modules/String/* directory, with the appropriate names (*TestModuleSimpleUIDialog.cc* and *TestModuleSimpleUIDialog.h*). For the *TestModuleSimpleUIDialog.h*, change the module name reference to the correct module name and delete the 'virtual

void pull()' function. The code should be very similar to the following:

```
#ifndef INTERFACE_MODULES_STRING_TestModuleSimpleUIDialog_H
#define INTERFACE_MODULES_STRING_TestModuleSimpleUIDialog_H

#include <Interface/Modules/String/ui_TestModuleSimpleUIDialog.h>
#include <Interface/Modules/Base/ModuleDialogGeneric.h>
#include <Interface/Modules/String/share.h>

namespace SCIRun {
namespace Gui {

class SCISHARE TestModuleSimpleUIDialog : public ModuleDialogGeneric,
  public Ui::TestModuleSimpleUIDialog
{
Q_OBJECT

public:
  TestModuleSimpleUIDialog(const std::string& name,
    SCIRun::Dataflow::Networks::ModuleStateHandle state,
    QWidget* parent = 0);
};
}}
#endif
```

The *TestModuleSimpleUIDialog.cc* requires similar treatment, but will require the addition of few more changes. Add and include for the module header file, change the namespace from 'Field' to 'StringManip', and delete the last two lines from the main function. The code should be:

```
#include <Interface/Modules/String/TestModuleSimpleUIDialog.h>
#include <Modules/String/TestModuleSimpleUI.h>

using namespace SCIRun::Gui;
using namespace SCIRun::Dataflow::Networks;
using namespace SCIRun::Modules::StringManip;

TestModuleSimpleUIDialog::TestModuleSimpleUIDialog(const std::string& name, ModuleStateHandl
  QWidget* parent /* = 0 */)
  : ModuleDialogGeneric(state, parent)
{
  setupUi(this);
  setWindowTitle(QString::fromStdString(name));
  fixSize();
}
```

This should be enough to create a UI for the TestModuleSimpleUI module, but it will not be able interact yet. We will need to modify this file later to connect all the required inputs.

For now, we can build SCIRun to test the UI design. Make sure that these three new files are added to the *CMakeList.txt* in the *src/Interface/Modules/String/* directory:

```
SET(Interface_Modules_String_FORMS

  ...

  TestModuleSimpleUIDialog.ui
)

SET(Interface_Modules_String_HEADERS



  TestModuleSimpleUIDialog.h
)

SET(Interface_Modules_String_SOURCES

  ...

  TestModuleSimpleUIDialog.cc
)
```

Once these files are added, SCIRun should build. Load SCIRun and place the TestModuleSimpleUI module. Open the UI for the module and make sure that looks correct.

Now we will work on connecting the input from the UI to the code in the module. Begin by modifying the *TestModuleSimpleUIDialog.cc* to include a line that reads the input field and assigns it to a variable. This line needs to go near the end in the main function of the module dialog code

```
addLineEditManager(inputstring_,TestModuleSimpleUI::FormatString);
```

This function will read the value of 'inputstring_' and set it to 'FormatString', which we have included as if it was part of the 'TestModuleSimpleUI' namespace. We will need to include it as such, by adding it as a public function in the *TestModuleSimpleUI.h* file.

```
static Core::Algorithms::AlgorithmParameterName FormatString;
```

This should be the final declaration in the public list (after the 'staticInfo_' declaration). Another change in this file is to modify the 'setStateDefault' function so that it is not empty. Remove the curly brackets from this:

```
virtual void setStateDefaults() {};
```

so that it is:

```
virtual void setStateDefaults();
```

We need a couple more additions to make the value from the UI available for use in the main function code. In the *TestModuleSimpleUI.cc* file, add the following line to the before the main execute function, i.e., right after declaring the namespaces.

```
SCIRun::Core::Algorithms::AlgorithmParameterName PrintStringIntoString::FormatString("Forma
```

Next we need to be able to set the state defaults by creating context for the 'setStateDefault' function we just exposed. Add this function just before the execute function.

```
void TestModuleSimpleUI::setStateDefaults()
{
  auto state = get_state();
  state->setValue(FormatString,std::string ("[Insert message here]"));
}
```

With these three additions, the code should build. If you load the module, you should she the default message ("[Insert message here]") in the input field. Changing this will still not affect the output because the execute function is still hard coded for a specific message.

Now we need to change the execute function to use the UI inputs, which is very simple. Simply get the state of the module (`auto state = get_state();`), then assign the output string variable to `state -> getValue(FormatString).toString();` so that the function is as follows:

```
void
TestModuleSimpleUI::execute()
{
  std::string message_string;
  auto state = get_state();
  message_string = state -> getValue(FormatString).toString();
  StringHandle msH(new String(message_string));
  sendOutput(OutputString, msH);
}
```

After building the software, you should now see that the output of module will be the same as the string that is put in the input field in the module UI.

Chapter 5

Example: Simple Module With Algorithm

Chapter 6

Converting Modules from SCIRun 4

Chapter 7

Creating Unit Tests

Chapter 8

Documenting the New Module