Limit Point Systems, Inc.

# Introduction

This document shows how to use key features of the SheafSystem. The C++ examples in the document are available as source code in the examples subdirectory of the SheafSystemProgrammersGuide module and the reader is encouraged to build and execute the examples along with reading the text. The examples are numbered and the source for example N is in exampleN.cc. There are a few compilation and execution examples in the text, these are given in Linux using the csh shell, Gnu C++, and Gnu make.

# What you'll need

To take full advantage of this document, you'll need a few things in addition to the document itself, namely:

* an installed copy of the SheafSystemProgrammersGuide module, which includes the examples,
* a C++ compiler,
* a web browser, and
* an installed copy of the SheafSystem.

# The SheafSystem installation

The SheafSystem installer installs all the files of the SheafSystem in a directory tree. We will have to refer to the root of this directory tree repeatedly, so to simplify the notation, we'll let <sheaf\_dir> refer to the full path to the root directory of the installation, for instance:

<sheaf\_dir> = /usr/local/SheafSystem

Wherever you see <sheaf\_dir> in this document, mentally replace it with the full path to your SheafSystem installation.

The installation includes 4 configurations of the libraries: Debug-contracts, Debug-no-contracts, Release-contracts, and Release-no-contracts. The "Debug" configurations are unoptimized and contain symbol information for use by interactive debuggers such as gdb. The "Release" configurations are optimized and contain no debugging information. We'll describe "Contracts" below. Generally speaking, the Release configurations are higher performance that the Debug configurations and the no-contracts configurations are much faster than the contract configurations.

The examples will compile and execute with any configuration, but we will always use the Debug-contracts configuration in the text below.

# Getting started

## PartSpace metaphor

The PartSpace document describes the fundamental concepts of the SheafSystem in non-mathematical terms using the common notion of basic and composite parts, tables, and table schema. This document assumes the reader is familiar with the PartSpace metaphor.

## Namespaces

As described in the PartSpace document, a SheafSystem database is a collection of tables. A namespace table is a special table in each database that serves as a container and table of contents for all the other tables. The SheafSystem includes 3 predefined namespace types: the sheaves\_namespace, the fiber\_bundles\_namespace, and the geometry\_namespace. Each of these predefines the sheaf schema for the C++ types defined in the sheaf, fiber\_bundle, and geometry components, respectively. (The fields component doesn't have its own schema).

Creating an instance of a namespace is typically the first thing a client must do to use the SheafSystem, so we start with an example of how to do that using the most basic namespace, sheaves\_namespace. This example will also cover the basic mechanics of compiling and linking with the SheafSystem.

### Example 1: Hello, Sheaf

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

// Disable the concurrency access control mechanism;

// will explain access control in example 3.

//

read\_write\_monitor::disable\_access\_control();

// Create a standard sheaves namespace.

sheaves\_namespace lns("Hello-sheaf");

// Write its name to cout.

cout << lns.name();

return 0;

}

This code is in the SheafSystemProgrammersGuide module in examples/sheaf/example1.cc along with a Makefile:

#

# Full path to your C++ compiler, for instance /usr/bin/g++

#

CXX = /usr/bin/g++

#

# Full path the SheafSystem installation include and library directories

#

SHEAF\_INC\_DIR =<sheaf\_dir>/include

SHEAF\_LIB\_DIR = <sheaf\_dir>/Debug-contracts/lib

example1: example1.cc

$(CXX) -o example1 -I$(SHEAF\_INC\_DIR) -L$(SHEAF\_LIB\_DIR) example1.cc -lsheaves

To compile and link the example, you first have to configure the Makefile to your installation by setting the 3 variables CXX, SHEAF\_INC\_DIR, and SHEAF\_LIB\_DIR to the actual values for your installation. Then we can compile and link by:

>make example1

This command will compile example1.cc and link it with the shared library libsheaves.so to create an executable example1 in the current directory. We have to tell the dynamic loader where to find the shared library by setting the environment variable LD\_LIBRARY\_PATH to contain the path to the SheafSystem library directory, that is, the same value we set SHEAF\_LIB\_DIR to in the Makefile, for instance:

>setenv LD\_LIBRARY\_PATH <sheaf\_dir>/Debug-contracts/lib

Now we can execute the example:

>./example1

Hello-sheaf

That's the basic mechanics of creating an application with the SheafSystem. We've created a sheaves\_namespace in this example, but before we can do much with it, we need to learn a few programming patterns that the SheafSystem uses repeatedly.

# Programming patterns

There are a few design features shared by all the classes in the SheafSystem. In this section we will give a quick introduction to the most ubiquitous of these patterns. We'll introduce some more patterns later, as we need them, and also go into some of these initial patterns in more detail.

## Design by contract

The sheaf system is implemented using the "design by contract" programming paradigm. We'll cover the essentials of the method and how they are used in the SheafSystem. For a more detailed introduction, see the excellent book Design By Contract, by Example by Richard Mitchell and Jim McKim.

When using design by contract, each class is equipped with an invariant, a set of assertions that must be true at any time control returns the client. (The invariant is not defined when control is within a member function of the class.) Every member function is equipped with preconditions and postconditions. The preconditions are assertions that must be true when control enters the member function; the postconditions must be true when control leaves the member function. The "contract" in "design by contract" is between the client and the member function: if the client guarantees the preconditions are true, the member function ensures the invariant and the postconditions are true.

The invariant, precondition, and postcondition assertions are specified using "invariance", "require", and "ensure" macros, respectively, in the source code. If contracts are enabled when the library is compiled, these clauses will be evaluated as part of the execution of the member functions. If the conditions specified in the clauses are not true, execution throws an exception with an error message, which usually terminates the program.

The contracts are extremely useful for detecting improper use of the classes and member functions and are thus an important debugging tool. Once client code is correct, the contracts can be disabled to improve efficiency.

The SheafSystem Debug-contracts and release-contracts configurations are compiled with contracts enabled. The Debug-no-contracts and Release-no-contracts are compiled with contracts disabled.

The contracts are also published as an essential part of the reference documentation and are critical to using the sheaf system correctly. Let's look at the reference documentation for the sheaves\_namespace constructor we used in example1. The reference documentation is generated in html, so you can open it with your browser. The main page is <sheaf\_dir>/documentation/C++/index.html. If you browse to the documentation for class sheaves\_namespace and click on the constructor sheaves\_namespace(const string& xname), you'll find:

**sheaf::sheaves\_namespace::sheaves\_namespace ( const string & *xname* )**

Creates a sheaves namespace with name xname.

Precondition

* poset\_path::is\_valid\_name(xname)

Postcondition

* [invariant()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#a952742bdad45c56c22fd9509a00e9c07)
* [name()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#aec09bcd260a52a459c8a35ae5bc1bef5) == xname
* !in\_jim\_edit\_mode()
* [host()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#adc8f6d6d2b952a6842a1d09de75bff9a) == 0
* !index().[is\_valid()](http://192.168.4.199/comp-tutorial-dev-4/df/d4b/namespacesheaf.html#a3dd8f96a360e1b63c6caa744e5ccf7b3)
* [index()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#a9a283b1819bc8e75b212bff26fc645b0).same\_scope(member\_hub\_id\_space(false))
* [has\_standard\_subposet\_ct()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#af5786ce90013ec6e72dbacd9b67e1c13)
* [current\_namespace()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#ae8ca3a11bc745cf0b275a70ab71b2d70) == this
* [state\_is\_not\_read\_accessible()](http://192.168.4.199/comp-tutorial-dev-4/d1/d3c/classsheaf_1_1read__write__monitor__handle.html#adc32a6090b2df1e5673444d5170539f3)

So what does this tell us? The precondition:

* poset\_path::is\_valid\_name(xname)

tells us exactly what conditions the argument xname has to satisfy if we want this call to the constructor to work correctly, namely is\_valid\_name(xname) has to be true. Well, what does that take? If we look up poset\_path∷is\_valid\_name we find:

**static bool sheaf::poset\_path::is\_valid\_name( const string &  xname )**

True if xname is not empty and contains only name legal characters.

Postcondition

* result == (!xname.[empty()](http://192.168.4.199/comp-tutorial-dev-4/d0/d38/classsheaf_1_1poset__path.html#a05ee8f14bcc22701b551059341f16749) && (xname.find\_first\_not\_of([name\_legal\_characters()](http://192.168.4.199/comp-tutorial-dev-4/d0/d38/classsheaf_1_1poset__path.html#a34019af3a5bee6f34d3ec2c2657a8671)) == string::npos))

So xname can't be empty and can't contain any characters not in name\_legal\_characters(). If we click on name\_legal\_characters we find:

**static const string & sheaf::poset\_path::name\_legal\_characters( )**

The characters a name is allowed to contain.

Postcondition

* result == "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789\_ -,.=+()\*:?"

So xname has to be non-empty and not contain any of the above characters.

If xname satisfies these conditions, which it does in example1, then the postcondition gives a great deal of information about what state the sheaves\_namespace object is in after construction.

The first postcondition is:

* [invariant()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#a952742bdad45c56c22fd9509a00e9c07)

that is, the invariant has to be satisfied. As we said above, this is an implicit postcondition of every member function, even if we don't explicitly provide it as part of the contract. So what does this mean for sheaves\_namespace? Well, click on invariant() to find:

**virtual bool sheaf::namespace\_poset::invariant ( ) const**

Class invariant.

Invariant

* poset\_state\_handle::invariant()
* is\_attached() ? primitives().is\_attached() : true
* is\_attached() ? (primitives().index() == PRIMITIVES\_INDEX) : true
* state\_is\_read\_accessible() ? primitives().state\_is\_read\_accessible() : true
* is\_attached() ? primitives\_schema().is\_attached() : true
* is\_attached() ? (primitives\_schema().index() == PRIMITIVES\_SCHEMA\_INDEX) : true
* state\_is\_read\_accessible() ? primitives\_schema().state\_is\_read\_accessible() : true

Sheaves\_namespace inherits namespace\_poset and doesn't override the invariant, which is a virtual function, so the invariant of sheaves\_namespace is the invariant of namespace\_poset. The invariant in a derived class must be at least as strong as the invariant in the base space, so the invariant of namespace\_poset calls the invariant of its base class, poset\_state\_handle. Beyond whatever poset\_state\_handle∷invariant() ensures, the namespace\_poset invariant ensures several properties of the data members primitives() and primitives\_schema().

The reader is encouraged to examine the poset\_state\_handle invariant to learn what additional invariances sheaves\_namespace has inherited, but we'll move on to the rest of the postcondition of the constructor. The next postcondition is one you'd expect:

* [name()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#aec09bcd260a52a459c8a35ae5bc1bef5) == xname

that is, the name of the namespace is the name we gave it.

The remainder of the postconditions ensure various arcane properties of the namespace that we're not very interested in right now. But when your tackling a tough debugging problem, any of these may be just the piece of information you need!

The power of the design by contract method comes from the great amount of detailed information contained in the assertions and two further properties. First, if contracts are turned on, that is if you are using either the Debug-contracts or Release-contracts configuration of the library, the pre- and post-conditions of a function are executed whenever the function is called. Second the contracts exhibited in the documentation are extracted directly from the source code. The combination of the two allows you to reason about the behavior of the code with great confidence while designing, programming, and especially while debugging.

So what happens if the contract for a member function is not satisfied? Let find out by trying to create a sheaves\_namespace without a name.

### Example 2: contract for sheaves\_namespace constructor.

#include "sheaves\_namespace.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

// Attempt to create a standard sheaves namespace

// with an empty name. This violates the preconditions

// of the constructor and will throw an exception and abort.

sheaves\_namespace lns("");

// Done.

return 0;

}

## Concurrency control

Monitor paradigm. global control, basic protocol. Enforce by preconditions.

Example 3: sheaves\_namespace∷name

Contrast poset\_state\_handle∷name with poset\_state\_handle∷name(bool). Show blown assertion

## Index spaces and scoped indices, part 1

## Index spaces

Basic notion, iterators.

Example 4: iterating over members of name space

Iterate over all members of namespace, then over member posets.

Demostrate need for scoped ids by printing out pods in both cases.

## Scoped ids

Basic notion. Examples using namespace member functions.

# Storage\_agent

Basic idea.

Example 5: write namespace to file

# Viewing Namespaces

## Stream insertion operator

Example 6: write namespace to cout

Write name to std out as a poset and as a namespace.

## Read.t

Example: view namespace with read.t

## SheafScope

Example: view namespace with SheafScope

# Posets

Example : creating, accessing, and deleting posets

## Creating posets

Create a poset, write to cout.

## Accessing posets

Three common signatures, poset path

Example: access poset and write to cout.

## Deleteing posets

Delete the poset.

# Poset members

Example 8: creating andd manipulating poset members with the poset interface

## Creating poset members

Create a jim

## Accessing poset members

Naming, various queries

## Ordering poset members

## Deleting poset members

## Handles

Repeat all the same examples with handles

## Schema posets