The Factory Approach to Creating TSTT Meshes

Tom Epperly

October 21, 2003
Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States
Government. Neither the United States Government nor the University of California nor any of their
employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for
the accuracy, completeness, or usefulness of any information, apparatus, product, or process
disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any
specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise,
does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United
States Government or the University of California. The views and opinions of authors expressed herein
do not necessarily state or reflect those of the United States Government or the University of California,
and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U.S. Department of Energy by University of
The Factory Approach to Creating TSTT Meshes

Tom Epperly <tepperly@llnl.gov>

Lawrence Livermore National Laboratory

http://www.llnl.gov/CASC/components/

October 21, 2003

Advantages & Costs of the Factory Approach

The factory approach (a.k.a. virtual constructor) hides the details of the class implementing the TSTT from TSTT users. In version 0.5 of TSTT.sidl, the client hard codes the name of the implementing class into their code. For example, the client had to call

TSTT::CompleteTSTTMesh::_create(), TSTT::SubsettableTSTTMesh::_create() or the constructor for one of the other concrete classes defined. The client is forced to choose from the small set of possible concrete classes defined in TSTT.sidl. This approach makes it impossible to support multiple implementations of the TSTT in a single process because each implementation has to implement the same class.

The factory approach hides the details of mesh creation from the client. The client does not need to know the name of the implementing class, and the client can dynamically determine which interfaces are supported by the new mesh. A factory can support multiple TSTT implementation because each implementation defines its own concrete classes to implement.

The factory approach does require the TSTT compliant mesh packages to implement a MeshFactory interface, and everyone needs to link against an implementation of the Registry. The Registry only has 7 methods that are fairly easy to implement, and everyone can share one implementation of the Registry.

How to create a TSTT mesh

There are two main ways to create a mesh. The first uses the factory, and the second uses a service provided by Babel. I provide two examples below on how the factory can be used. In the first example, the client uses the current default factory. I expect this to be the most common case because most programs will only have one TSTT implementation available. In the second example, the client chooses one of several available TSTT implementations. In the third example, I show how a client can use Babel's SIDL::Loader interface to create a TSTT mesh.

In the first example, the client requires a mesh that supports the AdvancedTSTTMeshQuery and ModifiableMesh interfaces.

Example 1

TSTT::Mesh myMesh; TSTT::AdvancedTSTTMeshQuery atmq;
TSTT::ModifiableMesh mm;
try {
    myMesh = TSTT::Registry::getInstance().
      getDefaultFactory().newMesh();
    atmq = myMesh; // try casting to AdvancedTSTTMeshQuery
    mm = myMesh;   // try casting to ModifiableMesh
    if (atmq._not_nil() && mm._not_nil()) {
        // rest of code goes here
    }
} catch (TSTT::Error err) {
    // some step of the creation failed
}

In the second example, the client needs CoreEntitySetQuery and BooleanSetOperations. The example iterates through all available MeshFactory's until it finds one that has both interfaces.

**Example 2**

TSTT::Mesh myMesh;
TSTT::CoreEntitySetQuery cesq;
TSTT::BooleanSetOperations bso;
SIDL::array<TSTT::MeshFactory> factories;
try {
    factories = TSTT::Registry.getInstance().getFactories();
    for(int32_t i = factories.lower(0);
        i <= factories.upper(0); ++i) {
        myMesh = factories.get(i).newMesh();
        cesq = myMesh; // cast to CoreEntitySetQuery
        bso = myMesh; // cast to BooleanSetOperations
        if (bso._not_nil() && cesq._not_nil()) break;
    }
    if (bso._not_nil() && cesq._not_nil()) {
        // insert rest of code using bso & cesq here
    }
} catch (TSTT::Error err) {
    // some step of the creation failed
}

In both these examples, I assume that the TSTT implementation has registered itself in the Registry before the example code is run.

There is another way to create an instance of a TSTT mesh if the client knows the fully qualified name of the class that implements the TSTT interfaces. The client can use the SIDL::Loader::createClass method assuming the implementation is available as a shared
library or statically linked into the main executable. In this example, the client needs the `ModifiableMesh` and `EntitySetRelations` interfaces.

**Example 3 – the sneaky Babel trick**

```cpp
const std::string className = "local.TSTT.ImplClass";
SIDL::BaseInterface bi;
bi = SIDL::Loader::createClass(className);
if (bi._not_nil()) { // load succeeded
    TSTT::EntitySetRelations esr = bi;
    TSTT::ModifiableMesh mm = bi;
    if (esr._not_nil() && mm._not_nil()) {
        // insert rest of code using esd & mm here
    }
}
```

The main weakness of this approach is how does the application determine the class name without limiting itself to a single hardwired TSTT implementation.

**What Implementations Need to Do**

This section describes what developers implementing the TSTT need to do in order to make their mesh implementation available for clients to use. In this example, the implementation supports the `CoreEntitySetQuery` and `Tag` interfaces. All of the code included here will work for any non-empty set of supported TSTT interfaces. My hypothetical implementation is called Gilga. Here is the SIDL file:

```idl
package Gilga version 0.0.1 {
    class Mesh implements-all TSTT.CoreEntitySetQuery, TSTT.Tag {
    }
    class Factory implements-all TSTT.MeshFactory { }
}
```

Here is how the Gilga::Factory makes the Gilga::Mesh. I've stripped out the doc comments and splicer block comments for the sake of brevity.

```cpp
::std::string
Gilga::Factory_impl::name ()
throw ()
{
    return "Gilga";
}

::TSTT::Mesh
Gilga::Factory_impl::newMesh ()
throw (
::TSTT::Mesh m = Gilga::Mesh::_create();
    return m;
}

The implementation of Gilga::Mesh is not affected by the use of the factory.

The last part of the puzzle is registering the factory with the Registry. This must be done early in the program execution, so here is an example where it is done first thing in main()..

int main(int argc, char *argv[])
{
    Gilga::Factory gf = Gilga::Factory::_create();
    TSTT::MeshFactory tf = gf; // cast to generic factory
    TSTT::Registry::getInstance().setDefaultFactory(tf);
    // insert the rest of your application here
}

Some applications may choose a plug-in architecture. In a plugin architecture, TSTT implementations would be stored in a shared library or dynamically loaded library. Typically, a framework is responsible for dlopen'ing the shared library and then calling an initialization routine whose name is a function of the plug-in name. For example, the gilga plug-in might live in libgilga.so and have an initialization routine named init_gilga that might look something like the following:

extern “C” int init_gilga(TSTT::FactoryCollection *fc
    /* perhaps other arguments */);
int init_gilga(TSTT::FactoryCollection *fc
    /* perhaps other arguments */) {
    if (registry) {
        Gilga::Factory gf = Gilga::Factory::_create();
        TSTT::MeshFactory mf = gf; // cast to generic factory
        fc->setDefaultFactory(mf);
        return 0; // here I assume 0 means success
    }
    return 1; // ERROR
}

**SIDL for TSTT::MeshFactory & TSTT::Registry**

Here are the SIDL descriptions of TSTT::MeshFactory and TSTT::Registry.

//===========================================================================
// Interface for creating mesh objects
//===========================================================================
/**
 * This interface can create empty mesh objects. Each implementation of the
TSTT implements a MeshFactory and registers it with the Registry.

```java
interface MeshFactory {
    //==========================name====================
    /**
     * This method returns the common name of the TSTT
     * implementation. This could be "AOMD", "CUBIT", "Frontier",
     * "Mesquite", "NWGrid", "Opt-MS", "Overture", "Trellis" or the name
     * of some other TSTT implementation. The name is only significant
     * when more than one MeshFactory is registered in the
     * registry. Clients will use the name to select which MeshFactory
     * they want.
     */
    String name();

    //==========================newMesh====================
    /**
     * Create a new, empty Mesh object. This method will either return an
     * allocated Mesh or throw an exception. It will never return a NULL
     * object handle.
     */
    Mesh newMesh() throws Error;
}
```

```java
class Registry {

    //==========================getInstance====================
    /**
     * Return the singleton Registry object. This will either return a
     * non-NULL object or throw Error. This method never returns a
     * NULL object handle.
     */
    static Registry getInstance() throws Error;

    //==========================addFactory====================
    /**
     * Add a factory to the registry. Adding the same factory
     * twice (or more) is equivalent to adding it once.
     * @param mf  the MeshFactory to be registered.
     */
    void addFactory(in MeshFactory mf) throws Error;

    //==========================removeFactory====================
    /**
     * Remove a factory from the registry. If this factory was the default
     * factory, the default factory becomes NULL. It is not an error
     * remove a factory that isn't registered.
     * @param mf  the MeshFactory to be removed.
     */
    void removeFactory(in MeshFactory mf) throws Error;
}
```
void setDefaultFactory(in MeshFactory mf) throws Error;

MeshFactory getDefaultFactory() throws Error;

MeshFactory lookupFactory(in string name) throws Error;

array<MeshFactory> getFactories() throws Error;