SANDIA REPORT
SAND2000-2380
Unlimited Release
Printed October 2000

A Design Patterns Analysis of the Umbra Simulation Framework

Daniel E. Small, Eric J. Gottlieb, Kim Edlund, and Cara Slutter

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

When printing a copy of any digitized SAND Report, you are required to update the markings to current standards.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent 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, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

SAND2000-2380
Unlimited Release
Printed October 2000

A Design Patterns Analysis of the Umbra Simulation Framework

Daniel E. Small and Eric J. Gottlieb
Intelligent Systems and Robotics Center
Sandia National Laboratories
P.O. Box 5800, MS-1010
Albuquerque, NM 87185-1010
desmall@sandia.gov, ejgottl@sandia.gov

Kim Edlund and Cara Slutter
University of New Mexico
Albuquerque, NM 87123
kedlund@unm.edu, caras@unm.edu

ABSTRACT

In this paper we present an analysis of the Umbra Simulation Framework. Umbra was developed by researchers at Sandia National Laboratories Intelligent Systems and Robotics Center. The framework allows users to quickly generate simulations with three-dimensional graphics and models. The analysis is performed on the standard design patterns used in developing the framework. The second chapter of the book, "Design Patterns" [Gam95], was used as a template for the analysis of framework. Nine different design patterns are discussed.

Contents

1.0 Introduction 6

2.0 Design Problems 7

3.0 Flexible Simulation Structure 8

4.0 Easy to Configure Simulation Structure 10

5.0 Dynamic Instantiation and Parameterization 13

6.0 Globally Available Objects 15

7.0 Integration of New and Legacy Simulation Codes 16

8.0 Notification of Changes 18

9.0 Encapsulation of Callback Functions 20

10.0 Framework Efficiency 22

11.0 Dynamic Module Loading 24

12.0 Summary 25

References 26

Figures

Figure 1. Screen Shot of Umbra Simulation of S-700 Robot 6

Figure 2. An Example of a Complex Architecture with Modules Connected Together in a Data-Flow Network..... 10

Figure 3. Modules for the S-700 Simulator and Controller 11

Figure 4. Module Configuration to Simulate a Swarm of UAVs..... 14

Figure 5. Screen Shot of Multiple UAVs in the Umbra Framework 15

Figure 6. The Class Hierarchy of the S-700 Module 17

Figure 7. The Relationships Between the Connector Class and the Callback Functions..... 19

Figure 8. Command Factory Showing the 'Manufacture' of Concrete Callbacks 21

Figure 9. Logical Data Flow 22

Figure 10. Flyweight Representation of Data Flow 22

1.0 Introduction

This report was written in the form of chapter two in the textbook, *Design Patterns* [Gam95]. Here, the report presents a case study in the design of a simulation framework called Umbra. A framework defines a subsystem that is customizable or extensible. Frameworks must be simple enough to be learned, yet must provide enough features to be useful.

Umbra is currently used at Sandia National Laboratories Intelligent Systems and Robotics Center for simulating small smart machines, mobile robots, and telerobotic arms, as well as a control environment for a robotic painting system for the F-117 aircraft and a research tool for cooperative automation research.

Frameworks are reusable designs described by a set of abstract classes that constrain the shape of a design while enabling the user to customize the details for a particular application [Cop92]. A framework user supplies the code to fill in the abstractions making it a complete system that solves a problem. When many problems need to be solved in a domain, frameworks can be cost-effective. They represent an excellent example of code reuse and programming to an interface rather than an implementation.

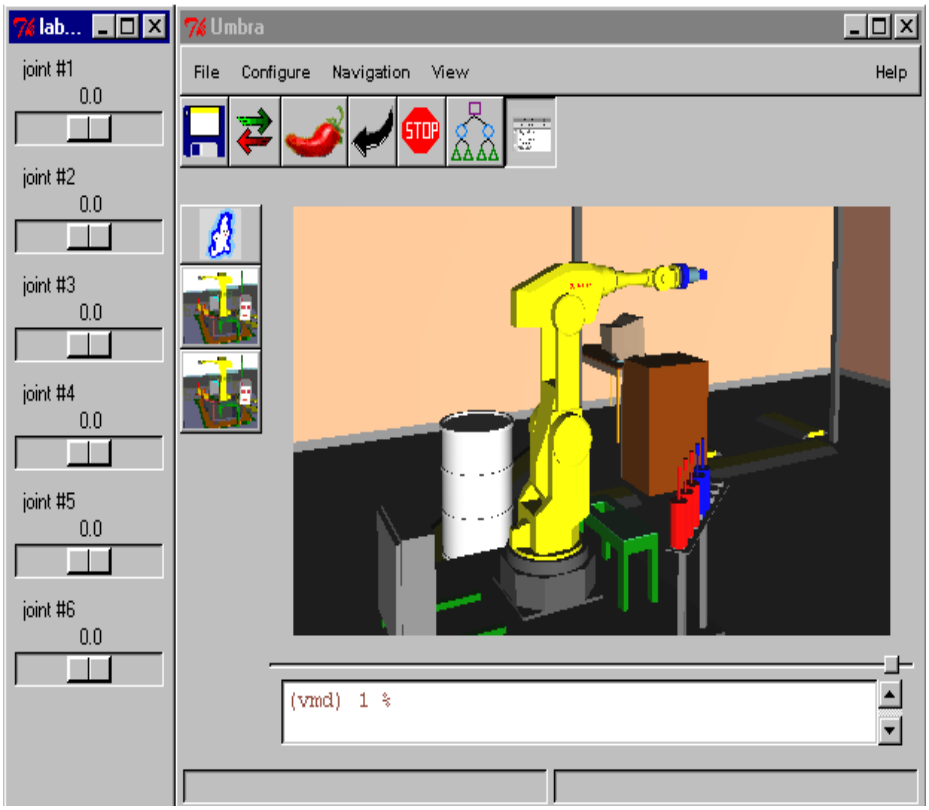

Figure 1. Screen Shot of Umbra Simulation of S-700 Robot.

The Umbra 'world' consists of a set of interconnected modules that represent data-flow networks and a scripting language interface that allows us to dynamically parameterize the modules. It builds upon existing 3D visualization software and the Tcl/Tk scripting language. Figure 1 shows an example screen shot of an Umbra simulation of the S-700 Robot.

Umbra modules are linked together by connectors in a data-flow network that has the characteristics of a directed acyclic graph. Currently, the components that make up an Umbra simulation are composed with the Tcl scripting interface. Future development efforts may focus on implementation of a graphical user interface (GUI) to represent the data-flow network (much like Khoros or AVS).

Umbra supports a diverse set of users who require a wide range of capabilities from Umbra. This creates a wide range of design problems that we need to address. We wish to satisfy all of the following:

1. Module-Developers: Users that write algorithms to implement new behaviors will require Umbra to be extensible. These are developers that implement the data-flow modules that other people may use in simulations.
2. Simulation Developers: Other users will write scripts to instantiate existing modules, and connect them to 3D models. This type of user would like the framework to be reconfigurable.
3. Non-technical Users: Our last type of user might be a person that has little or no knowledge of computer simulation. This person might need to run a simulation that is easily parameterized by a GUI.

We will introduce the design problems that Umbra set out to solve, and we will take a close look at the design patterns used in the framework. Umbra utilizes object-oriented design to create a component-based framework for users to rapidly build 3D simulations.

2.0 Design Problems

We will examine the design problems that needed to be addressed in Umbra's development:

1. *Flexible Simulation Structure*. The Umbra Framework should support incremental development of simulations for prototyping and easy reuse of modules. The framework needs to have a standard interface for communicating information between modules in the data-flow network. The connections between modules in the data-flow network and between the 3D models with which they are associated form the architecture of Umbra's design.

2. *Easy to Configure Simulation Structure*. An Umbra simulation is defined by a script that instantiates the modules, defines their connections, and associates them with models in the 3D visualization. The Umbra Framework needs a good scripting interface. This will allow users to communicate directly with the framework and gives us the ability to support a diverse set of users. The system needs to be both rapidly configurable and reconfigurable.

3. *Dynamic Instantiation and Parameterization*. For each type of Module in the network, we will need a unique function that dynamically allocates instances of that Module.

4. *Globally Available Objects*. The framework needs to have global access to the unique objects that create the Module-derived class instances. There will be only one instance for each of these and they need to be global in two domains: the interpreter and in C++.

5. *Integration of New and Legacy Simulation Codes*. The framework needs to have a well-defined way of reusing legacy simulation codes that may apply to current problems, as well as implementing modules that reflect new behavior.

6. *Notification of Changes*. The framework needs to have an automatic way to notify interested modules of changing information in the network.

7. *Encapsulation of Callback Functions*. The framework will rely heavily on callback functions to propagate information. We need a clean, object-oriented method for specifying and invoking callback functions.

8. *Framework Efficiency*. Simulations have large amounts of data that need to flow between modules and computationally intensive algorithms. The Umbra Framework should be efficient and should not impose undue overhead on the simulation.

9. *Dynamic Module Loading*. The modules need to be dynamically loaded at runtime. The user must also have the capability of swapping modules in and out during simulation execution.

We will cover each of these problems in the sections that follow. Each problem and its design constraints are explained in more detail. The solutions will utilize one or more design patterns and the reader will be introduced to the relevant design patterns used in the Umbra Framework.

3.0 Flexible Simulation Structure

The Umbra modules perform the mathematics behind the 3D models in a simulation. They can represent motion controllers for robots, physics-based models such as particle simulations, natural phenomena, or just about anything that you can model with mathematics. They are implemented as C++ classes that inherit from a standard abstract interface. All modules have input and output connectors that determine data flow. By connecting the modules together in a data-flow network we can define complex, high-level behaviors [Bus96].

The purpose of the data-flow network is to pass data through a sequence of transformations (filters, modules, etc.) that are connected by input/output channels (connectors). The output channel of one module is connected to the input channel of subsequent processing steps.

This design will provide flexibility as transformations are decoupled from each other. Modules do not directly depend on other modules so they may be independently added or deleted. The overall topology of the network can be completely changed and adapted to achieve different functionality.

The graph propagates data from its inputs (sources) to its outputs (sinks) by executing the Update function within each module. The `Update()` function is responsible for:

- reading data from its input connectors
- processing/transforming/filtering the data
- writing the data to the output connectors

The `Update()` function for each module is executed as part of a tight loop within Tcl. The activity of each module is triggered by the event loop and the input connector pulls data from the previous module's output connection.

Each module is designed to do one thing well. By allowing the modules to perform smaller amounts of processing we add flexibility to the system. Modules may be rearranged dynamically to perform a variety of different functions.

Different sources of input data exist but produce the same type of data. For instance, the system might have four different ways of getting input for a set of joint values for a robot:

1. Input comes in "live" directly from a robot controller.
2. Input is recorded from previous sessions and played back.
3. Input comes from a dynamic simulation.
4. Input is received over the network.

The design of the system also adds the flexibility to present or store the output results of the modules:

1. Outputs of joint values get mapped onto 3D models.
2. Outputs of joint values go into a database.
3. Outputs of joint values get transmitted over the network.

This architecture also allows for the modeling of complex computational models, such as neural networks, as shown in Figure 2. Note that the output of one module can be passed to the input connection of many other modules. As shown in Figure 2, a module may also have multiple output connectors.

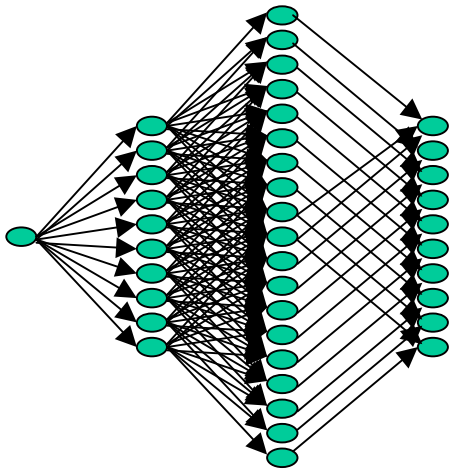

Figure 2. An Example of a Complex Architecture with Modules Connected Together in a Data-flow Network.

This architecture enables the future implementation of a multi-processing system, via multiple CPUs on the same machine, or distributed CPUs within a network. It also supports incremental development of simulations for prototyping and easy reuse of modules.

Pipes and Filters Pattern

The Pipes and Filters architectural pattern [Bus96] provides the overall structure for the Umbra data-flow network. We have extended the single-input, single-output filter specification described in [Bus96] to allow filters with more than one input and more than one output, and processing is set up as a directed graph.

Templated DataConnectors can be of any data type and they control the processing stages of the data flow. The network of modules and connectors can be combined in many ways because of the standard interface between modules. This allows the framework to be customized for the application. Modules will act as filters and they can change data and pass it via an OutputConnector to another module for processing. The output data of one step is the input data to the next step.

4.0 Easy to Configure Simulation Structure

The structure of a simulation written in the Umbra Framework consists of three basic elements: Umbra Modules/Components written in C++, 3D models that may be attached to those modules, and a script written in the Tcl language that instantiates the components and connects them together at run-time. These elements capture the entire structure of the simulation.

The following diagram shows an example of an Umbra simulation of an S-700 robot and the high-level relationships between the modules and the connectors.

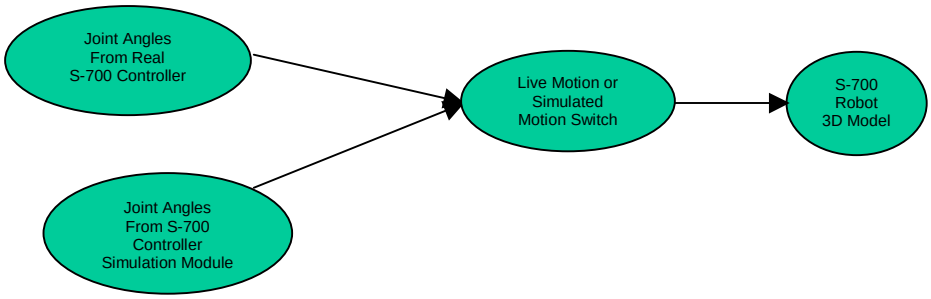

Figure 3. Modules for the S-700 Simulator and Controller.

In this example, we have data flowing out of two sources into a switch module that decides which input data to allow through to the 3D model of the robot. The data that is being propagated is an array of six doubles that corresponds to the six joint values of the S-700 robot. The switch module has two input connectors and one output connector. Every green circle represents a subclass of the module class. The 3D model is a special subclass of module that knows how to display itself in the 3D viewer.

Each module subclass determines how many input and output connectors it will have. This may be determined statically when the class is written, or dynamically by declaring that each module has a vector of input and output connectors.

The following code shows the C++ interface to the Module class.

```
namespace umb
{
  /** base class for all umbra modules. **/
  class Module : public slu::TclInstance
  {
  public:
    /// …
    /// Update gets data from the input connectors, processes it and
    /// sends data to the output connectors
    virtual void Update() = 0;
    /// set a connector value.
    int tclSetConnector(Tcl_Interp* interp, int objc, Tcl_Obj* const objv[]);
    /// get a connector value.
    int tclGetConnector(Tcl_Interp* interp, int objc, Tcl_Obj* const objv[]);
    ///other Tcl methods
  private:
    std::map<std::string,Connector*> _connectors;
    /// …
    friend class Connector;
  };
}
```

The module class will have two separate interfaces. The first is a C++ interface that is defined in the class header. The second interface is the interface to the scripting language. The functions that begin with a lowercase “tcl” in the class definition implement C++ functionality that is

accessible via the scripting interface. It is through these functions that the modules are parameterized and connected to other modules. Because this occurs at runtime, we require that the modules have no a priori knowledge of which modules are sending them data or which modules are receiving their data.

Modules communicate with each other via their connectors. A typical Umbra module will inherit from the Module base class and will include in its public interface a set of InputConnector and OutputConnector objects. These objects allow the Module to receive data from Modules connected to its InputConnectors, process the data, and then send the data to the Modules connected to the OutputConnectors. This functionality is handled by the pure virtual `Update()` function that is implemented by the subclass.

The user interface for the framework should allow users to view the simulation using either a 3D or a textual interface. The interface needs to be flexible enough to allow advanced users to program their own GUIs. Each GUI will be specifically tailored to the needs of the specific simulation that they are developing (i.e., a GUI for an unmanned autonomous vehicle simulator is going to need a different user interface than a simulation for a robot controller). Therefore, the system requires a graphical user interface that can be easily reconfigured to provide functionality specific to a user's application. The Tcl language provides a widely used extension for graphical user interfaces called Tk that is an appropriate tool for the purposes of this system. The combined system is known as Tcl/Tk.

Scriptable Components Pattern

We will use Tcl/Tk to implement the Scriptable Components design pattern [UofLondon]. This is an architectural pattern that is used to loosely bind together the modules via their input and output connectors. The modules are fully decoupled from all other modules in the system except the language interpreter itself.

The Tcl language makes it easy to compose modules into a complete application. The scripting language is much simpler to use than a full-blown system control language such as C++. Tcl can be used by novice programmers to compose simulations out of lower-level modules. The module development can be left to experienced software engineers. The module developer defines the number and types of input and output connectors that a module will use, as well as an interface between the Tcl interpreter and the module.

This design pattern provides a level of flexibility that is necessary for this framework. Simulations can now be rapidly assembled and modified on the fly to meet the needs of a particular customer.

Related Patterns

Another related pattern to Scriptable Components is the Configuration Script Pattern [UofLondon]. This is actually a pattern often contained within a Scriptable Components pattern and, indeed, it is also the case in our example. The Configuration Script Pattern is used when objects need to communicate in more efficient ways than are provided by the scripting language.

In this pattern, binary interfaces are defined between components to facilitate speed of data transfer. In Umbra, these binary interfaces are the input and output connectors.

5.0 Dynamic Instantiation and Parameterization

The system needs to provide a dynamic interface in the scripting language for two basic operations: instantiation and parameterization of modules.

Dynamic instantiation of modules is necessary for several reasons:

- Simulating swarms of the same types of objects
- Providing an easy way for the user to add new objects on the fly
- Allowing the system to automatically adapt itself to changing conditions in the simulation by instantiating or deleting modules in the data-flow network (i.e., a self-adjusting neural network)

Dynamic parameterization of modules is necessary because we may need to set up modules using a custom set of parameters for every simulation. In addition, we need the capability to modify elements of the modules via the GUI. Since the GUI is also implemented by the scripting language, this is an easy goal to accomplish. The base class of Module is TclInstance. There is another class called TclClass that cooperates with TclInstance derived classes to provide a dynamic representation of the C++ objects as commands in the Tcl shell. There should be one TclClass for each Module subclass. Normally this is arranged by declaring a static TclClass instance in the module's source file (as shown below). The constructor for the Module class is passed implicitly via the template parameter and the name and methods via the other parameters. This information is used to add a Tcl command for generating instances of the S-700Module class. This new command can be executed multiple times to generate many instances of the S-700Module class. The TclClass will also arrange for the methods declared in the S-700Module class to be associated with parameters of the new Tcl command (i.e. the method `tclConnect` is associated with the Tcl command parameter `"connect")`.

```
static int initializer =
  TclClass<S-700Module>:: Instance()->init( "S-700Module",
      "setConnector",S-700Module::tclSetConnector,
      "getConnector",S-700Module::tclGetConnector,
      "update",S-700Module::tclUpdate,
      "connect",S-700Module::tclConnect,
      "unconnect",S-700Module::tclUnconnect,
      "connectors",S-700Module::tclConnectors,
      "connected",S-700Module::tclConnected);
```

In this example, the TclClass is instantiated statically with the class S-700Module as its template parameter. This will associate all instances of S-700Module with the list of commands in the parameter list. The first command, "S-700Module", is the new Tcl command that creates instances of the S-700Module class. It is executed in the Tcl shell as a base command with a parameter that will be the name of the S-700Module instance that is created. For example, to create an S-700Module named myRobot1, we execute:

```
Tclsh> S-700Module myRobot1
```

To create an associated model, we type:

```
Tclsh> Model myS-700Model
Tclsh> myS-700Model load "/home/desmall/3Dmodels/Fanuc/S700.wrl"
```

where "Model" is a special kind of Umbra module that knows how to represent itself in the 3D viewer.

The new instance may then be parameterized by calling the method functions associated with that class. For example, to connect the output of myRobot1 to the input of myS-700Model, we execute:

```
Tclsh> myRobot1 connect output1 myS-700Model input1
```

This code will instantiate two modules and connect them, all within the context of the Tcl script. It gives the component developer an easy way to associate dynamic instances of modules with their individual object methods.

In the previous example (S-700) there is one instance each of two different modules. The system also needs to support the capability to have multiple instances of the same modules. Let's say that we want to simulate a swarm of unmanned autonomous vehicles (UAV) flying through the air. We will have one basic trajectory that the UAVs will follow, but each one will have its own slight perturbation off of the base path that corresponds to its place in the overall formation. This can be accomplished with one instance of the TrajectoryModule that controls the main trajectory and multiple instances of the PathModule that control the relative offsets for each UAV in the formation. These two types combine to feed each instance of the UAV-Module, as shown in Figure 4. Figure 5 shows the screen shot of the multiple UAVs.

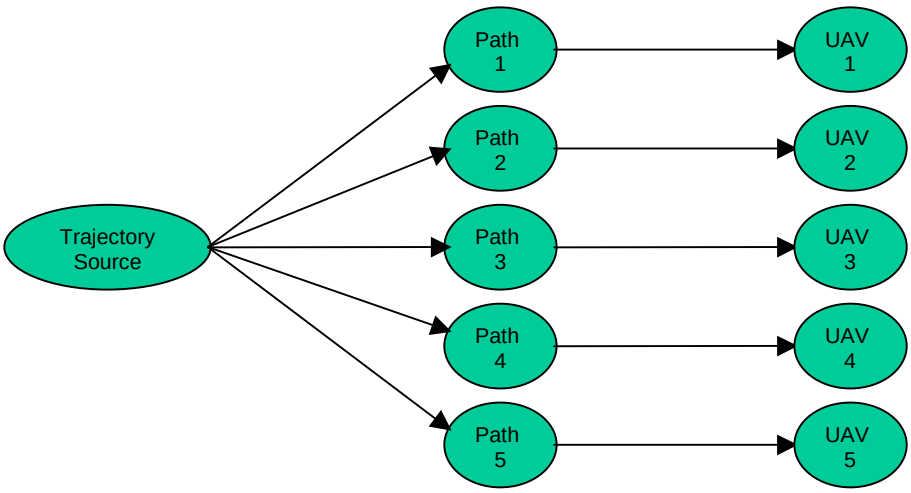

Figure 4. Module Configuration to Simulate a Swarm of UAVs.

Factory Method

The capability to dynamically add new module instances at run-time is an example of a Factory Method [Gam95]. By using a templated class we avoid having to subclass a base creator class. The creator class is now the template class, TclClass. The concrete product is the result of instantiating the template with the Module subclass as a parameter.

Figure 5. Screenshot of Multiple UAVs in the Umbra Framework.

6.0 Globally Available Objects

In the previous section we implemented a Factory Method for each type of module that we have available in the system. We statically allocate the Factory Method for each module as follows:

```
static int initializer =
  TclClass<S-700Module>::Instance()->init( "S-700Module",
      "setConnector",S-700Module::tclSetConnector,
      "getConnector",S-700Module::tclGetConnector,
      .
      .
      . );
```

This instantiates a new command in the Tcl interpreter that serves as the Factory Method and there is one Factory Method per class. In the case above, `TclClass` creates a unique instance once the template is instantiated. There is one for each Module-derived class. You can get the

unique instance with this static method `Instance()` (code for `Instance()` is shown below). Normally this type of method would construct an instance if one did not already exist. However, because TclClass needs more information from the constructor (one built with a null constructor wouldn't be of much use) we provide lazy construction using the `TclClass<S-700Module>::Instance()->init()` method that takes all of the information necessary to complete the specification of the unique instance.

The following code shows how `dynamic_cast` can be used in a templated `Instance()` function to determine the appropriate TclClass Singleton object.

```
static TclClass<Owner>* Instance()
{
      TclClass<Owner>* that;
      TclGenClass* base = TclGenClass::the(&typeid(TclClass<Owner>));
      if (base == NULL)
          that = new TclClass<Owner>();
      else
          that = dynamic_cast<TclClass<Owner>*>(base);
      return that;
}
```

Singleton Pattern

This is an example of the Singleton Pattern [Gam95], which is useful for creating objects that will have one unique instance. Since the TclClass is templated there are many different kinds of TclClasses; each one is a unique Singleton and each is globally accessible.

7.0 Integration of New and Legacy Simulation Codes

The Umbra Framework requires an easy way to integrate new modules into the system, as well as an easy way to integrate legacy simulation codes. Legacy codes may include specific functionality such as simulating robot path motion, identifying collisions, and monitoring joint angles in a real-time, 3D environment. These legacy codes that were written for older systems need to be *adapted* so that their interfaces are compatible with the standard module interface that the Umbra Framework expects.

For example, a simulation of an S-700 robot controller is created to monitor real-time joint angles. A legacy S-700Controller class reads in analog voltages from the robot controller that corresponds to joint angles. This class cannot be used directly in an Umbra simulation since it doesn't have the standard Module interface. Umbra modules communicate with each other through input and output connectors.

To use the legacy S-700Controller class in a simulation, an S-700Module class is written to wrap the S-700Controller class. This is needed to provide a standard interface so that input data obtained from the robot controller can be communicated to the rest of the Umbra simulation. The S-700Module inherits from the abstract base class Module and overrides the virtual `Update()` method. The general purpose of the `Update()` method is to read from the input

connectors, perform the processing, and write to the output connectors. In this case, however, our input comes directly from the robot hardware. Therefore, when the `Update()` method is called it will invoke the S-700Controller `getJointValues()` method. This method will read input voltage values from the robot controller and convert them to joint angles. Now that we have the six joint angles, we can communicate them using the standard interface. This list of joint angles is loaded into the S-700Module's output connector. Other modules that are connected to this output connector will see the joint angles during their next `Update()` cycle. Figure 6 illustrates the class hierarchy.

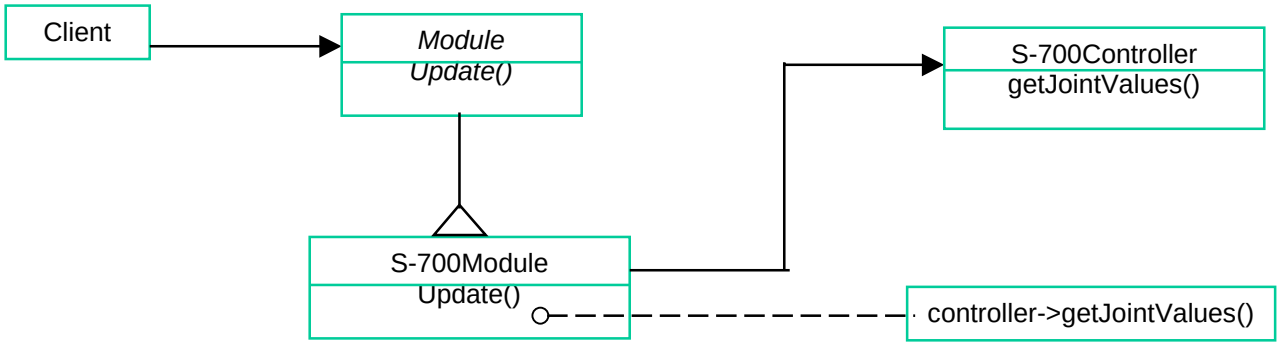

Figure 6. The Class Hierarchy of the S-700 Module.

The code example that follows shows how this class hierarchy is implemented.

```
//Code to be wrapped in a module
class S-700Controller
  {
  public:
    /// ...
    /// get joint angles
    bool getJointValues(double *jointAngles);
};

// Wrapper module
class S-700Module : public umb::Module
  {
  public:
    ///...
  protected:
    // update is required if you inherit from Module.
    void Update();
  private:
    S-700Controller *controller;
    std::vector<double> _jointOutValue;
    umb::OutputConnector< std::vector<double> > _jointOut;
    double _joints[6];
  };
}
```

```
S-700Module::S-700Module():
    // data associated with output connector
    _jointOutValue(6,0.0),
    // associates data with output connector on construction
    _jointOut(this,"jointOut",&_jointOutValue)
{
  // initialize data
  for (int i = 0; i < 6; i++)
      _joints[i] = 1.0;

  // allocate module
  controller = new S-700Controller();
}

void S-700Module::Update()
{
      int index;
      // invoke the getJointValues method on the controller object
      controller->getJointValues( _joints );
      // propagate the information to the output connector
      for (index=0; index<6; index++)
      {
          _jointOutValue[index] = _joints[index];
      }
}
```

The S-700Controller class is the code to be adapted. The S-700Module class wraps the S-700Controller code to look like a module to the rest of the Umbra simulation. The S-700Module constructor allocates a new controller, initializes the data, and associates the controller's data with the S-700Module's output connector.

Adapter Pattern

User-written Module classes will utilize the Adapter Pattern [Gam95] to provide the standard interface that the Umbra Framework Modules expect.

8.0 Notification of Changes

The framework needs to have an automatic way to propagate changing information through the network. Normally this is done using a pull method within a module to read the input connectors, process the data, and write it to the output connectors. This was described earlier in the Flexible Simulation Structure section (Pipes and Filters Pattern).

Consider what will happen if we have several modules of varying computational complexity. If every module must be executed on every update cycle, then the most expensive module dominates the entire network execution time. For example, if Module A takes 1000 times longer to run than any of the other modules, the speed to compute one cycle for the total network is limited by the speed and complexity of Module A.

However, let's say that the data associated with the input connectors for Module A rarely changes. This implies that the output connectors of this module need to be updated only when the input changes. We now introduce a new way of implementing the pull model in the graph, which states:

```
if the InputConnector data has changed
      read the InputConnector data
      process the InputConnector data
      write to the OutputConnector
```

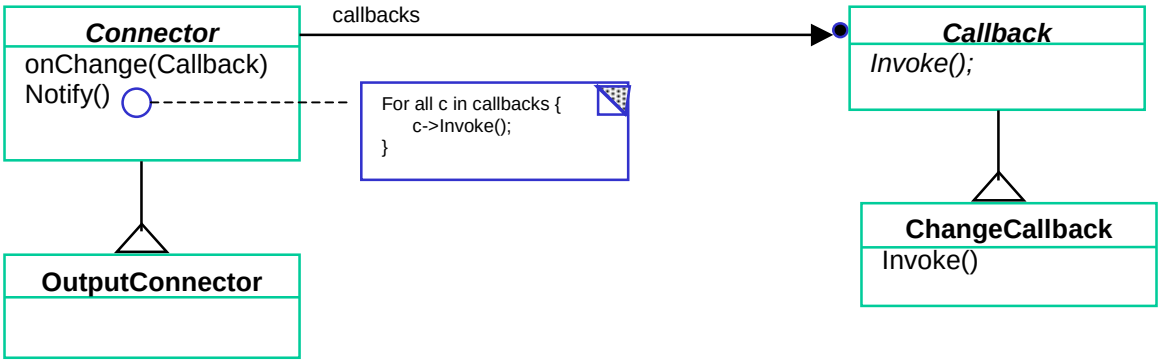

Figure 7. The Relationships Between the Connector Class and the Callback Functions.

Because the `Update()` functions for each module are executed continuously, we require a new way of executing the update logic. This logic will no longer be contained in the `Update()` function of the Module-derived class because we know that this function is computationally complex and that its input data will not be changing that often. The `Update()` function for Module A will now be implemented as a stub function. Since the function should only be executed if the input data changes, it makes sense to set it up as a callback function. We also need to account for multiple modules being interested in the same OutputConnector, and must accommodate for it in the design.

We can accomplish this by adding a new function to the Connector base class called `onChange(slu::Callback* cb)`, which takes a pointer to a standard callback method as a parameter. This function is for registering the update function that performs the pull functionality. This function will be invoked whenever the connector data has changed, and can work for both InputConnector and OutputConnector classes.

Observer Pattern

The preceding is an example of the Observer Pattern [Gam95]. This pattern should be used when a Subject needs to notify other interested Observers without regard to what those Observers are and when the Subject and Observer are loosely coupled.

In our example, the Callback is the Observer, and the Connector is the Subject. The Callback class is subclassed to ChangeCallback, which is now contained in the Module class and performs the functionality of the stub `Update()` function. This allows for the creation of specialty Modules that will only be called when their InputConnectors are changed.

9.0 Encapsulation of Callback Functions

The design of the Umbra callback mechanism focuses on the encapsulation of callback functions combined with the clever use of the Factory Method to create different callback objects. Callback functions give a module programmer flexibility to organize the order of calls in a way that is appropriate for the application.

Modules, InputConnectors, OutputConnectors, and our graphical models will all need to have callbacks associated with them in order to make the system robust. The callback functions need to be invoked in a standard way that decouples the callback objects from the process that is invoking them.

First we define an abstract Callback class to provide an interface for invoking a callback request. The basic interface is the invoke function.

```
class Callback
{
  public:
    virtual void invoke() = 0;
    // . . .
};
```

Next we need concrete procedure callbacks. For clients calling `invoke()`, all callbacks are the same, however, subclasses of Callback can use callbacks in various ways. In Figure 8, MethbackZero stands for method callback with zero parameters and creates a pointer to a function without parameters.

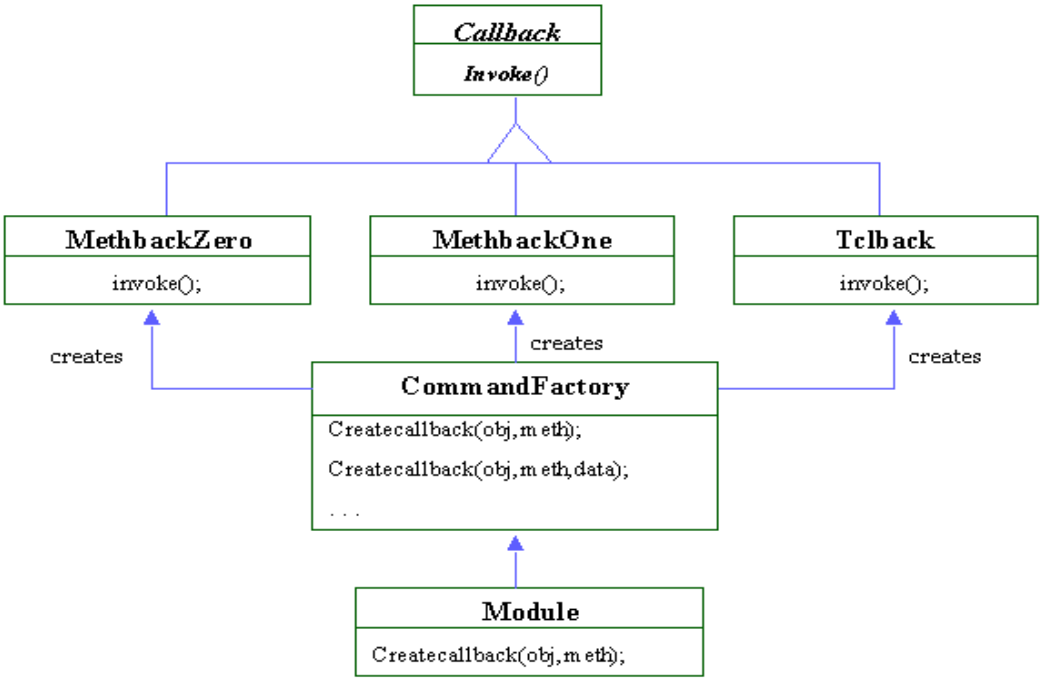

Figure 8 Command Factory Showing the 'Manufacture' of Concrete Callbacks.

Likewise, MethbackOne will create a pointer to a function that can accept a data parameter. Tclback creates a pointer to a callback function that can be defined by a user in a tcl script. All of the concrete callback classes create functions that return pointers to Callback.

Our encapsulation of callbacks utilizes a Command Factory to parameterize the types of callbacks. We use a variation of the Factory Method mentioned in [Gam95] where the factory creates multiple types of callback objects based on the parameters sent to it. With this design we also avoid subclassing by using templated classes and an overloaded function, `Createcallback()`.

In the code below, `obj` is a pointer to an instance of the object on which callback invokes the method. The parameter `meth` is the method to invoke. In the case shown below, the method should accept zero parameters.

```
template<class ObjType, class MethType>
Callback* Createcallback(ObjType* obj, MethType meth)
{
   return new MethbackZero<ObjType, MethType>(obj,meth);
}
```

Command Factory Pattern

This combined use of the Command Pattern and the Factory Method moves the decision process of which type of callback to create out of the framework. Clients don't need to know the exact class to create because the Factory Method is responsible for 'manufacturing' the correct callback object [Gra99] [Gam95].

The Command Pattern gives us an object-oriented way of encapsulating callbacks to provide clients with a standard interface. Concrete callback class names do not even appear in client code. Only Callback pointers are created, deleted and passed around. By encapsulating callbacks, Umbra has a centralized factory that any object (i.e., modules or connectors) needing a callback mechanism can use.

10.0 Framework Efficiency

Many simulations will require numerous connections between Modules, which could consume lots of time in propagating data and slow down the simulation. Our design requires that these modules process information without incurring an unacceptable run-time overhead.

We wish use to the data in the Connector objects without prohibitive costs. Logically, the network will consist of a series of source Modules with OutputConnectors to other Modules with InputConnectors (see Figure 9). Here the dark blue boxes represent InputConnectors, the light blue boxes represent OutputConnectors, and the arrows represent the data flow.

Physically, there will be only one shared data object owned by the OutputConnector of the source Module. This is accomplished by passing pointers to data rather than copies of the data. The InputConnectors of the receiver Modules point to the proper OutputConnector (see Figure 10). In this figure the arrows represent pointers to the data object.

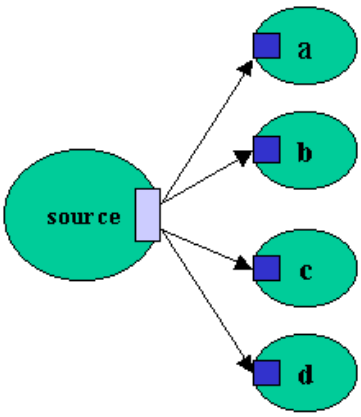

Figure 9: Logical Data Flow

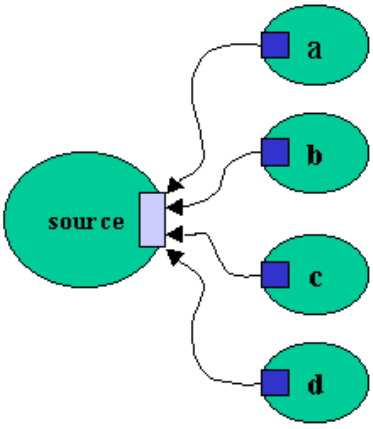

Figure 10: Flyweight Representation of Data Flow

To accomplish this we have an inheritance hierarchy that starts with a base class called Connector. Connectors get a pointer to each other (i.e., this is connected to that).

```
bool Connector::connect(Connector* that)
{
  bool done = true;
  // if the data types are compatible, connect the output to the input
  if (this->connectable(that) && that->connectable(this))
    {
      this->rawConnect(that);
      that->rawConnect(this);
  }
  else
    {
      done = false;
    }
  return done;
}
```

None of the functions in the Connector class need to know the data type that is being passed. The constructor is protected and only derived classes will create Connector objects. The DataConnector class inherits from Connector. It is templated so that users have the flexibility to create many different types of DataConnectors.

We have minimized the content of this template class to include only methods that need to know the type of data that is being propagated. We wanted to minimize the amount of code in the template class because a template class is harder to debug. This use of templates broadens the scope of our connector abstraction to accommodate the needs of most applications. Users can create DataConnectors of many different types (i.e., they could be integers, doubles, user-defined objects, etc.).

```
template <class Data>
class DataConnector : public Connector
  {
  public:
    /// return a reference to the data associated with the connector.
    /// This is how data is pulled from a source
    const Data& get() const
      {
      return *_data;
      }

    /// set the pointer to the data associated with the connector.
    /// This is how data is written to a sink
    void set(const Data& data)
      {
      *_data = data;
      doSetCallbacks();
      }
   protected:
```

```
    /// constructor
    DataConnector(Module* module, const std::string& name, Data* data)
      : Connector(module,name),_data(data) {}
    ///  is this connectable to that? (i.e. same data types?)
    virtual bool connectable(Connector* that)
      {
        return (Connector::connectable(that) &&
            0 != dynamic_cast<DataConnector<Data>*>(that));
      }
  private:
    Data* _data;
};
```

To instantiate a Connector (which is a member of a Module-derived subclass) we need to associate the Connector with a private data item in the class. For example, here we have the private data of the S-700Module class which contains the OutputConnector `_jointOut`, and a vector of doubles `_jointOutValue`.

```
private:
    S-700Controller *controller;
    std::vector<double> _jointOutValue;
    umb::OutputConnector< std::vector<double> > _jointOut;
```

In the constructor for the S-700Module class we initialize the `_jointOut` Connector with a pointer to this S-700Module and a reference to the instance data that is associated with the Connector.

```
S-700Module::S-700Module() :
    // data associated with output connector
    _jointOutValue(6,0.0),
    // associated data with output connector on construction
    _jointOut(this,"jointOut",&_jointOutValue) {…}
```

Flyweight Pattern

Using flyweights create substantial timesaving in a memory-intensive application like Umbra. This design saves space especially if the simulation gets larger and more connectors are used to propagate data. By using the Flyweight Pattern [Gam95] we will avoid the copying of data and this will give us the efficiency required. We also covered the use of an interesting use of a template class to provide the flexibility needed by users of the Umbra Framework.

11.0 Dynamic Module Loading

Umbra modules need to be dynamically loaded at runtime. Users must also have the capability of swapping modules in and out during simulation execution. This flexibility will allow system developers to create large libraries of modules that can be made available on the fly.

To implement this effectively, modules must be instantiated in a standard way. Because different operating systems implement dynamic loading using their own unique set of system calls, we will use the facilities available in Tcl to dynamically load Umbra modules.

The steps to dynamically load a new module in Tcl are as follows:

The first step is to execute the Tcl script command load.

```
Tclsh> load ModuleLibaryName tclPackageName
```

This Tcl shell command "load" takes a base filename (ModuleLibaryName) with no extension (i.e., .dll, .so, etc.) and a packageName which is associated with a registry of user-supplied packages in Tcl.

The next step is for C++ to run the static initialization in the loaded binary code, such as the Singleton Factory initialization that was discussed earlier in the "Globally Available Objects" section.

Finally, an initialization procedure (whose name is associated with the tclPackageName) is executed, and all of the other necessary initialization is performed.

There are two levels of interface standardization that are implemented in this example. The first is the standard C++ static initialization (that we use to instantiate the Factory Methods), and the second is the standard Tcl `Init()` function that gets executed.

Dynamic Loading Pattern

In the Dynamic Loading Pattern [Gra98], executable library code is loaded into the main process and the routines respect a common interface. Typically, this is done through class interface inheritance, where the modules would all inherit from the same abstract base class. Our system implements the pattern in a similar fashion, except the system executes common initialization code (both statically and explicitly) instead of interface inheritance. The initialization code implements new commands in the Tcl shell which become available to the user.

By using the Tcl shell to implement dynamic loading, we avoid some of the common issues associated with developing cross-platform code. The Tcl scripts look exactly the same, and the module code just needs some `#defines` that differentiate platform-specific aspects of their implementations.

12.0 Summary

We have analyzed nine different patterns used in Umbra's design:

1. *Pipes and Filters* to create a simple, standard interface for data flow in the network and provide a flexible design for prototyping and code reuse.

2. *Scriptable Component and Configuration Script* to create an easy-to-use simulation structure that supports a diverse set of users.

3. *Factory Method* to support dynamic instantiation and parameterization of modules that allows for the creation or modification of many different simulations.

4. *Singleton* to provide a globally accessible instance of each unique TclClass Factory Method.

5. *Adapter* to provide a wrapper around new legacy code that requires a standard interface to act as an Umbra module.

6. *Observer* to notify interested modules of changes in the data-flow network.

7. *Command Factory* to generically encapsulate callback functions that are all executed in the context of their abstract base class.

8. *Flyweight* to support efficient simulations by eliminating the copying of data.

9. *Dynamic Loading* to allow modules to be loaded at runtime as well as swapped during simulation execution.

References

[Bus96] Frank Buschmann, Regine Meunier (Contributor), Hans Rohnert (Contributor), Peter Sommerlad. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Son Ltd, 1996.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley Publishing Company, Reading MA, 1992.

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, Reading MA, 1995.

[Gra98] Mark Grand. *Patterns In Java*. John Wiley & Sons, 1998.

[Gra99] Mark Grand. *Personal Patterns Homepage, Visio drawings*, http://www.mindspring.com/~mgrand/pattern_synopses.htm

[UofLondon] The Distributed Software Engineering Group of the Department of Computing, Imperial College of Science Technology and Medicine, University of London, England. Patterns for Scripted Applications, Pattern: Scripted Components. *http://www-dse.doc.ic.ac.uk/~np2/patterns/scripting/scripting.html*

DISTRIBUTION:

1 Cara Slutter
University of New Mexico
Computer Science Department
Farris Engineering Center, Room 157
Albuquerque, NM 87131

1 Kim Edlund
University of New Mexico
Computer Science Department
Farris Engineering Center, Room 157
Albuquerque, NM 87131

1 Charles Crowley
University of New Mexico
Computer Science Department
Farris Engineering Center, Room 157
Albuquerque, NM 87131

1	MS1002	P. Eicker, 15200
1	MS1004	R. Harrigan, 15221
1	MS1004	P. Bennett, 15221
1	MS1004	S. Gladwell, 15221
1	MS1004	E. Gottlieb, 15221
1	MS1004	M. McDonald, 15221
1	MS1004	F. Oppel, 15221
1	MS1004	R. Peters, 15221
1	MS1004	B. Rigdon, 15221
1	MS1004	D. Schoenwald, 15221
1	MS1004	J. Trinkle, 15221
1	MS1004	P. Xavier, 15221
5	MS1004	ISRC Library, 15221
1	MS1006	P. Garcia, 15271
1	MS1010	M. Olson, 15222
1	MS1010	D. Small, 15222
1	MS1221	R. Skocypec, 15002
1	MS9018	Central Technical Files, 8945-1
1	MS0161	Patent & Licensing Center, 11500
2	MS0899	Technical Library, 9616
1	MS0612	Review & Approval Desk, 9612 For DOE/OSTI