IV. UG Open Architecture

IV.1. UG Open Essentials

"Open architecture" is software design that allows access to program functionality without requiring access to the source code. We have already done this in a way, with the libraries that we have used (e.g., iostream). We have used classes and functions from libraries without actually seeing or compiling the source code. All we needed was the class and function declarations from header files. Open architectures are similar to this, but with one difference: the open architecture program must call *your* functions. Therefore, to write an extension to an open architecture program, your classes and functions must follow the patterns that have been established in the open architecture.

If you are not familiar with using Unigraphics NX it will be helpful to go through a quick introductory tutorial before continuing. The tutorial is available on line: Start Unigraphics NX, and from the menu select "Help" "Training…" This will start a web browser with the tutorials. Complete the online tutorial *Design An Overview of Unigraphics*. (Tutorial parts are available in the /usr/arch/apps/ugs190/ugcast/parts directory.)

Instructions for using the Unigraphics open architecture are also provided on-line: After starting UG NX select "Help" "Documentation…" This will start a web browser with the user manuals. Under the "Open" heading are manuals that you will find useful in this course: "API Programmer's Guide," "API Reference Guide," "Open C++ Programmer's Guide," "Menuscript User's Guide" and "User Interface Styler." The two "API" guides are for C program access using the "UG/Open API." API stands for "Application Programmer's Interface." This is a common acronym referring to open architecture classes and functions – another acronym that is often used is SDK: "Software Development Kit." The C++ guide is for C++ access using "NX Open C++." Note that not all functionality is available through NX Open C++ and therefore UG/Open API will also be used. The latter two guides (menuscript and UI styler guides) will be helpful if you choose to use the UG windowing functionality rather than the Windows toolkits we have previously discussed. "GRIP" provides a facility for writing interpreted programs in a UG programming language. GRIP will not be covered in this course.

UG open architecture programs can be written as "External" or "Internal" programs. External programs run independently from the UG executable. That is, it is not necessary to start UG to run External programs. External programs use UG code simply for the numerous libraries. Since UG is not running, no graphical interaction is possible unless it is specially programmed in. On the other hand, Internal programs are compiled as dynamically linked libraries and must have a function that matches the prototype:

```
void ufusr(char *param, int *retcod, int parm_len);
```

This function is started from UG by selecting "File" "Execute UG/Open" "User Function" and selecting the dynamic link library in the dialog box. It is also possible to modify the UG installation to allow executing this function through menu selections or toolbar buttons. This course will only address Internal programs.

The best way to start creating an open architecture program is to study and adapt an existing program that is similar to the one you are creating that you have already compiled and shown to work. This will not only illustrate you how the program should be written, but will allow you to use a Makefile that already has the correct compile settings and libraries. In the ME-EM department, many sample programs are available in the directory /usr/arch/apps/ugs190/ugopen, mixed in with all of the UG/Open API header files.

IV.2. Typical Application

The general form of an Internal program using UG/Open API is:

```
#include <uf.h>    // uf.h has declaration for ufusr()

// Additional include files as required


void ufusr(char *param, int *retcod, int parm_len)
{

   // variable declarations

   UF_initialize();

   // body

   UF_terminate();
}
```

The arguments in `ufusr()` are usually not used. `param` gives parameters that are passed into the program from UG. `parm_len` is the number of characters in `param`. `retcod` is a pointer to a variable that holds the return code which may be passed back to UG. `UF_initialize()` and `UF_terminate()` are called before and after the body of the function, to allocate and deallocate a UG/Open API license respectively.

An actual program will also provide error checking, so `ufusr()` may be written as:

```
void ufusr(char *param, int *retcod, int parm_len)
{

   if (!UF_CALL(UF_initialize())
   {

      // body

      UF_CALL(UF_terminate());
   }
}
```

`UF_CALL` is a macro that automatically flags errors, including printing the filename and line number of the error.

In NX Open C++, the initialization and termination is accomplished automatically through the constructor and destructor of the UgSession class:

```
#include <uf.h>
#include <ug_session.hxx>

// Additional include files as required


void ufusr(char *, int *, int )
{

   // variable declarations

   UgSession my_session (true);

   // body

}
```

To use the UgSession class it is necessary to include the header file ug_session.hxx. The Boolean argument that is passed to the constructor of the UgSession object tells whether or not UF_terminate() should be called automatically in the destructor.

To add error handling the C++ way, the following code is used:

```
#include <uf.h>
#include <ug_session.hxx>
#include <ug_exception.hxx>

// Additional include files as required


void ufusr(char *, int *, int )
{

   // variable declarations

   try
   {
      UgSession my_session (true);

      // body
   }
   catch (UgException oops)
   {
      // error message to user
   }

}
```

The try/catch expression is for "exception handling." Instead of returning true or false from a function to indicate an error, as is normally done in C, exception handling requires the "throw" command to be used to create an exception object. In this example an exception can be thrown in the UgSession constructor or in the body of the function, resulting in the creation of a UgException object “oops.” This object contains information about the error, which can be displayed to the user in an information window:

```
#include <uf.h>
#include <ug_session.hxx>
#include <ug_exception.hxx>
#include <ug_info_window.hxx>

// Additional include files as required


void ufusr(char *, int *, int )
{

   // variable declarations

   try
   {
      UgSession my_session (true);

      // body
   }
   catch (UgException oops)
   {
      UgInfoWindow info;
      info.open();
      info.write(oops.askErrorText());
      info.write("\n");
      return;
   }

}
```

The body of the function can evaluate information stored in the user’s model, can modify the information, can delete information, or create new information. The body of the function can also change how the information is displayed (e.g., changing the viewing direction). These tasks are accomplished by accessing API functions or functions in UG classes.

There are four classes of objects available:

- Application classes such as UgSession, UgInfoWindow, and UgException, are used to control the general operation of the application.
- Object classes such as UgPart, UgArc, UgFace, and UgExpression, are used to access and control the information.
- Helper classes such as ThruPoint and Evaluator are used in other objects.
- Math classes such as for points, vectors, matrices, and coordinate systems, allow easily representing the mathematical calculations.

Each class is generally declared in its own header file named the same as the class.