Chapter 8.
Extending the Modeler

Topic: *Extending ACIS

Developers can enhance the functionality of ACIS and modify the nature of the ACIS geometric model. ACIS is *extensible* through the addition of new:

APIs . . . . . Combine underlying functionality with application support such as argument error checking and roll back.

Classes . . . . . Extend the geometric model and add objects specific to the application domain.

Attributes . . . . . Attach user-defined data to model entities, and add information and relationships to the model in a way that ensures proper behavior under all geometric modeling circumstances.

Applications developed in Scheme can also extend ACIS through the creation of new Scheme extensions.

Whenever ACIS is extended, the developer can make the new functionality available in the test environments by creating new Scheme extensions that can be used in Scheme AIDE.

This chapter describes how to create new APIs, derive new classes, and define run-time virtual methods. For more information, refer to the following manuals:

Scheme Support Component Manual . . . . . Contains information on extending the Scheme interface.

Kernel Component Manual . . . . . Contains information on attributes.

Naming Conventions

Topic: Extending ACIS, C++ Interface

When you extend ACIS with the addition of your own C++ code, you need to follow certain naming conventions to avoid problems.

Sentinels

Topic: Extending ACIS, C++ Interface

To avoid naming collisions between developers who create new C++ code that is linked with ACIS, *Spatial* requires the use of a *sentinel.* A sentinel is a two- or three-character string which is embedded in class and API function names, and identifies the development organization. Every new API and class derived from ENTITY or ATTRIB uses this sentinel as part of its name (this is described in specific documentation about creating new APIs and deriving classes). The sentinel ensures that developers do not apply the same name to a function or class.

Note *Contact Spatial's customer support department to have a unique sentinel assigned to your ACIS development project.*

Filenames

Topic: Extending ACIS, C++ Interface

To avoid naming collisions, the filenames you use when extending ACIS must be unique (they cannot conflict with the names of files existing in ACIS). It is recommended that filenames follow the "8.3" format, which means the name should be no more than eight characters long, plus a three character extension.

Extending the Application Procedural Interface

Topic: *Extending ACIS, *C++ Interface

The *Application Procedural Interface* (API) is a layer of functions that expose the underlying functionality of the ACIS modeler. The API guarantees a stable interface, regardless of modifications to low-level ACIS data structures or functions, and bundles model management and error handling features. When an error or interrupt occurs in an API routine, ACIS automatically rolls the model back to the state when that API routine was called. This ensures that the model is not left in a corrupted state.

The API follows the general guidelines of the CAM-I Application Interface Specification (AIS), but is broadened to handle the wider range of entities that are represented in ACIS models.

Developers can create their own API functions to extend the interface to ACIS. This section describes how to create API functions that conform to ACIS standards.

Creating a New API Function

Topic: Extending ACIS, C++ Interface

Each API function should be placed in a single file. The filename should reflect the routine name if possible. An API function name begins with the prefix api_. This prefix should be followed by a two or three character sentinel that identifies the owner of the API.

This section describes how to create an API. It uses api_make_cuboid as an example and describes the major features used in APIs.

API Function Characteristics

Topic: Extending ACIS, C++ Interface

An API function performs an ACIS modeling or management task. APIs:

1. Have a name beginning with api_.
2. Are called by an application or a component.
3. Obtain their arguments from caller.
4. Error check each argument.
5. Perform the modeling or other required task.
6. Return a result (of type outcome).

Steps

Topic: Extending ACIS, C++ Interface

The following steps describe how to create API functions that conform to ACIS standards.

API Source .cxx File

Topic: Extending ACIS, C++ Interface

1. Copy an existing API file and rename the file according to its new function.
2. Name the API function api_<sentinel>_<function> where <sentinel> is the two- or three-letter code unique to the component or application (refer to the *Sentinels* section).
3. Create a comment header block that describes the API in the standardized format.
4. Include standard API .hxx header files and your API-specific header files.
5. Declare the API function as returning an outcome.
6. Declare all output arguments to be passed by reference, not by value.
7. Check arguments to the function using standard argument checking functions. Refer to the *Argument Error Checking* section.

```
"if (api_checking_on){check_body(arg)...};".
```

8. Bracket the body of the API function with macros API_BEGIN and API_END, API_SYS_BEGIN and API_SYS_END, or API_NOP_BEGIN and API_NOP_END.
9. Perform the specific functions for which this API was created.
10. Return a result.

API .hxx Header File

Topic: Extending ACIS, C++ Interface

1. Edit the API header file that exists in the source directory in which the .cxx file was created or, if one does not already exist, create a header file in this directory.

   The header file is named <*dir_name*>/<*hdr_name*>.hxx, where <*dir_name*> is the name of the source code directory in which the API source code .cxx file is located and <*hdr_name*> is the name of that directory's API header file.

2. Place the prototype for the API function into the header file.

   For support of DLLs, the declaration of the API (prototype) must be prefaced by its appropriate DECL_<*MODULE*> macro, where <*MODULE*> is an abbreviation for the module (component). Refer to Chapter 3, *Object Libraries*, for more information.

Example API Function

Topic: Extending ACIS, C++ Interface

Example 8-1 shows the function prototype (from the header file) while Example 8-2 shows most of the source file for the API function api_make_cuboid. The prototype is in the header file <*install_dir*>/cstr/constrct/kernapi/api/cstrapi.hxx:

C++ Example

```
DECL_CSTR outcome api_make_cuboid(
            double,             // size in x coordinate direction
            double,             // size in y coordinate direction
            double,             // size in z coordinate direction
            BODY *&             // body constructed
        );
```

Example 8-1. Header File for api_make_cuboid

Example 8-2 contains most of the source file, <*install_dir*>/cstr/constrct/kernapi/api/mkcuboid.cxx:

C++ Example

```
#include "kernel/acis.hxx"
#include "kernel/dcl_kern.h"
#include "constrct/kernapi/api/cstrapi.hxx"
#include "kernel/kernapi/api/api.hxx"
```

```
// {... Documentation code template not shown ...}

// Include files:
#include <stdio.h>
#include ”kernel/logical.h”
#include ”kernel/kernapi/api/check.hxx”
#include ”kernel/kernapi/api/api.err”
#include ”constrct/kernbody/primtive/primtive.hxx”
#include ”kernel/kerndata/top/body.hxx”
#include ”kernel/kernutil/debug/module.hxx”

#define MODULE() api
MODULE_REF(KERN);

// *********************************************************
outcome api_make_cuboid(
    double      width,
    double      depth,
    double      height,
    BODY*&      body )
{
    DEBUG_LEVEL(DEBUG_CALLS)
        fprintf(debug_file_ptr, ”calling api_make_cuboid\n”);

    API_BEGIN

        if (api_checking_on)
        {
            check_pos_length(width, ”width”);
            check_pos_length(depth, ”depth”);
            check_non_neg_length(height, ”height”);
        }

        body = make_parallelepiped( width, depth, height );
        result = outcome( body == NULL ? API_FAILED : 0 );

    API_END

    DEBUG_LEVEL(DEBUG_FLOW)
        fprintf(debug_file_ptr, ”leaving api_make_cuboid: %s\n”,
        find_err_ident(result.error_number()));

    return result;
}
```

Example 8-2. Source File for api_make_cuboid

Include Files

Topic: Extending ACIS, C++ Interface

Example 8-3 shows some include definitions. These are typical of the header files that are not specific to the API that are required for most API functions.

The acis.hxx file must always be included, must always be the first ACIS include file, and must be before any application include files.

C++ Example

```
#include <stdio.h>
#include ″kernel/acis.hxx″
#include ″kernel/logical.h″
#include ″kernel/dcl_kern.h″
#include ″kernel/kernapi/api/api.hxx″
#include ″kernel/kernapi/api/check.hxx″
#include ″kernel/kernapi/api/api.err″
#include ″kernel/kernutil/debug/module.hxx″
```

Example 8-3. Header File Declarations

Function Return Type and Error Handling

Topic: Extending ACIS, C++ Interface

Most API routines return a record of type outcome. For example:

```
outcome api_make_cuboid(...)
```

An outcome holds an error number of type err_mess_type, which is an integer. If the outcome is successful, the error number is 0. A calling function may examine the outcome using the following functions:

ok ..... Returns TRUE if the outcome was successful, (the error number is 0).

error_number ..... Returns the error number.

find_err_mess ..... Returns the text for the error message.

Example 8-4 illustrates an API function call.

C++ Example

```
outcome result = api_make_cuboid(width, depth, height, body);
if(! result.ok())
{
    err_mess_type err_no = result.error_number();
    printf("Error in make_cuboid %d : %s\n",
        err_no, find_err_mess(err_no));
}
```

Example 8-4. Example API Call

Every API routine includes an error trap to catch error exits from within the modeler.

The API routine may detect some warning conditions that do not cause the operation to exit. The warning messages are not returned with the outcome. Rather, the application must ask the error system for a list of the warnings caused.

Example 8-5 shows how to print warning messages.

C++ Example

```
err_mess_type* warnings;
int nwarn = get_warnings(warnings);
for (int i = 0; i < nwarn; ++i) {
    printf("Warning %d : %s\n",
        warnings[i],
        find_err_mess(warnings[i]));
}
```

Example 8-5. Example Warning Message Handling

If logging for roll back is in operation, an error in an API routine causes the model to be rolled back to its state prior to the call. If logging is off, the modeler halts.

Function Arguments

Topic: Extending ACIS, C++ Interface

An API may have arguments that are input or returned. No argument is used for both purposes. Arguments given should precede arguments returned in an argument list. Output arguments are passed by pointer reference, not by value. For example, api_make_cuboid takes three doubles for width, depth, and height. The fourth argument is a pointer to the body that is returned.

Making APIs C–Callable

Topic: Extending ACIS, C++ Interface

A C++ function is not necessarily made C-callable by prototyping the C++ function with the extern "C" keywords. While the extern "C" keyword prevents the name of the C++ from being mangled, there are other issues to be considered.

C does not use classes. Therefore, it cannot pass classes by value. If a C++ API function requires that a class be passed by value as its input, it is not C callable, regardless of whether it is prototyped with the extern “C” keyword. This leads to the following rule: API functions must not pass classes by value.

C does have void pointers. A C void pointer points to the address of a class just as it points to a structure. Therefore, a C++ API function would be C-callable if its class arguments were passed by address (BODY*) or by reference (BODY&). Either method allows the API function to be called from C.

ENTITY Class Arguments

Topic: Extending ACIS, C++ Interface

Model entities are referred to by pointers. For example, if a routine requires a body as an input argument, it passes an argument of type BODY*. Similarly, if a body is to be returned, it is returned via a variable of type BODY*&. After an error, output argument values are undefined.

ENTITY_LIST Class Arguments

Topic: Extending ACIS, C++ Interface

When a group of similar arguments must be returned and the number of arguments is not known in advance, they are returned as a list of entities using class ENTITY_LIST. For instance, when the routine api_cover_sheet is used to find every simple loop of external edges of a sheet body, the faces made are put into an ENTITY_LIST, which is returned.

The ENTITY_LIST implementation uses hashing, so look up is fast if the lists are not long. It is also efficient for repeated lookups of the same ENTITY.

ENTITY_LIST provides constructor (which creates an empty list) and destructor functions to add an entity to the list, look up an entity by pointer value, to remove an entity from the list, and to count the number of entities in the list. It also provides an overloaded [ ] operator for access by position, and two methods, init and next, for faster sequencing through the array.

The preferred way of accessing the items in the list is through ENTITY_LIST::init, which rewinds the list, and ENTITY_LIST::next, which returns the next undeleted item, as is shown in Example 8-6.

C++ Example

```
ENTITY_LIST my_face_list;
api_cover_sheet(sheet_body, new_surface, my_face_list);
ENTITY* my_list_item
my_face_list.init();
while ((my_list_item=my_face_list.next()) != NULL){
    if is_FACE(my_list_item){
    .
    .
    .
    }
}
```

Example 8-6. ENTITY_LIST Using init and next

The ENTITY_LIST class is a variable length associative array of ENTITY pointers. When using the subscript operator, a cast is required to coerce the ENTITY pointer into a pointer of the correct type. Refer to Example 8-7. For best performance in loops that step through this list by index value, have the loops increment rather than decrement the index counter. Internal operations for methods like operator[] and remove store the index counter from the previous operation, allowing faster access to the next indexed item when indexing up.

C++ Example

```
ENTITY_LIST my_face_list;
api_cover_sheet(sheet_body, new_surface, my_face_list);
int number_of_faces = my_face_list.count();
for (int i = 0; i < number_of_faces; i++){
    FACE* face = (FACE*)my_face_list[i];
    .
    .
    .
}
```

Example 8-7. ENTITY_LIST using Cast and the Operator []

Begin and End Macros

Topic: Extending ACIS, C++ Interface

The body of every API is enclosed within a pair of begin and end macros. The pair of macros used depends on the type of API. These macros set up error handlers and alter the behavior of the bulletin board. The text for macros is in the files <install_dir>/kern/kernel/kernapi/api/api.hxx and <install_dir>/kern/kernel/kernutil/errorsys/errorsys.hxx.

The possible pairs of macros are described in the following sections.

API_BEGIN and API_END Macros

Topic: Extending ACIS, C++ Interface

The purposes of an API_BEGIN and API_END block are to:

1. Protect the contained logic block with ACIS exception handling.

2. Provide rollback behavior in the event a modeling operation fails.

The API_BEGIN macro declares a variable ”result” of class ”outcome”. When performing modeling operations within an API_BEGIN and API_END block, the result variable should be used to contain the success or failure of those operations. Take for example the following fictional customer function, which returns true or false if the following set of operations succeeds/fails:

```
bool do_something(BODY *& retbody)
{
    bool ret;
    BODY *prism = NULL;
    BODY *cone = NULL;

    API_BEGIN

    result = api_make_prism(..., prism);

    if(result.ok())
        result = api_make_frustum(..., cone);

    // prism is tool, cone is blank.  The result of
    // this call is the blank (cone); the tool (prism)
    // is deleted.
    if(result.ok())
        result = api_intersect(prism, cone);

    // If a NULL body is returned, then there was no
    // overlap – we consider this failure.  Force a rollback
    // by setting ”result” to a failing value.
    if (cone == NULL)
        result = outcome(API_FAILED);

    API_END

    // Setup our return values.
    ret = result.ok();
    retbody = cone;
```

```
    return ret;
}
```

There are some significant points to note about this simple example:

- The variable result is used for determining the success or failure of the entire API_BEGIN/API_END logic block. It is even set to API_FAILED if the prism and cone don't intersect. By doing this, the API_END will examine result and, if it is non–zero, then rolls back all changes (assuming this is the outermost API_END).
- In the event of any type of failure, no cleanup is needed. The rollback will perform all cleanup.
- There are no return's or goto's inside the API_BEGIN/API_END block. This is important because once an API_BEGIN block is entered, it must be exited through it's corresponding API_END block.

Let us take this example one step further by considering how this routine is called:

```
void main(void)
{
    API_BEGIN

    BODY *bod;
    bool r = do_something(bod);

    result = (r == true) : outcome(0) ? outcome(API_FAILED);

    // If the result is not ok, no cleanup is needed; all
    // results will be rolled back when we pass through API_END.
    if (result.ok())
        api_del_entity(bod);

    API_END
}
```

This illustrates that nesting of API_BEGIN/API_END blocks is perfectly acceptable. However, there are important behavioral differences:

- If any failures occur within do_something, it's API_END does NOT rollback the changes. The rollback always occurs at the outermost API_END, which is now the API_END in main.
- It is important for do_something to propagate the knowledge of success or failure using it's bool return value. It allows main() to realize success or failure and set it's result appropriately. The knowledge of success or failure must be propagated upward through API_BEGIN/API_END blocks to ensure that the rollback occurs at the outermost API_END.

API_NOP_BEGIN and API_NOP_END Macros

Topic: Extending ACIS, C++ Interface

The macros API_BEGIN and API_END

The macros API_NOP_BEGIN and API_NOP_END begin and end every ACIS API that does not change the model. They provide a way to use the bulletin mechanism to make changes to an ENTITY, then throw away the changes when done. This is useful when you want to evaluate the model without changing it. Changes are allowed to happen as part of the operation, but they are then undone by API_NOP_END.

API_NOP_BEGIN creates a stacked bulletin board. The stacked bulletin board is logically nested inside an already existing bulletin board, if one exists. Nesting of the associated macros must be strictly maintained.

Because API_NOP_BEGIN and API_NOP_END make use of the bulletin facilities, the effect is lost if bulletin board logging is turned off. Therefore, bulletin board logging is temporarily turned on in API_NOP_BEGIN and reset in API_NOP_END.

API_NOP_BEGIN creates a new bulletin board but does not roll back any failed previous bulletin boards. This ensures that the bulletin board is in the same state after the NOP macros as it was before. API_NOP_END rolls over the bulletin board opened by API_NOP_BEGIN. The bulletin pointers of the entities in the previous bulletin are restored to their original state, leaving the bulletin board in the state it was before the API_NOP_BEGIN macro.

API_TRIAL_BEGIN and API_TRIAL_END Macros

Topic: Extending ACIS, C++ Interface

Macros API_TRIAL_BEGIN and API_TRIAL_END can be used to bracket code that may or may not have results you want to keep. Like API_NOP_BEGIN, API_TRIAL_BEGIN creates a stacked bulletin board. In API_TRIAL_END, the outcome result is checked. If the result is good, the model edits are retained and the stacked bulletin board is merged with the main bulletin board. If the outcome is bad, the results are rolled away as if API_NOP_BEGIN and API_NOP_END had been used. These “trial” macros only work if logging is turned on. Therefore, bulletin board logging is temporarily turned on in API_TRIAL_BEGIN and reset in API_TRIAL_END.

API_TRIAL_BEGIN creates a stacked bulletin board. The stacked bulletin board is logically nested inside an already existing bulletin board, if one exists. Nesting of the associated macros must be strictly maintained.

The main difference between these “trial” macros and the “normal” API_BEGIN/API_END macros is in the timing and in what gets rolled back. The trial macros create a stacked bulletin board so the rolling section is exactly what is between the two macros, regardless of how they may be nested inside other sets of macros. In the trial case, the roll back occurs in API_TRIAL_END, whereas in the normal case, the roll back is delayed until the next bulletin board is opened. In the normal case, the roll back is also dependent on the error eventually getting propagated to the outermost API_END. The trial case always rolls back on failure without having to propagate to the outermost level.

API_SYS_BEGIN and API_SYS_END Macros

Topic: Extending ACIS, C++ Interface

Use API_SYS_BEGIN and API_SYS_END for API system routines. System routines manipulate bulletin boards and roll back, and therefore do not return bulletin boards, or change the model.

Argument Error Checking

Topic: Extending ACIS, C++ Interface

Argument checking is conditional in ACIS. The api_checking_on function controls whether the validity of the API arguments is checked before continuing with the API operation. In the file api.hxx, a line of code defines api_checking_on:

```
#define api_checking_on (api_check_on())
```

If api_checking_on is TRUE, the checking functions determine if the arguments are valid. API checking is disabled. The function api_checking_on is TRUE when the option is TRUE. The checking functions in this example check only for greater than 0.

Conditionally check arguments:

```
if (api_checking_on)
{
    check_pos_length(width, ”width”);
    check_pos_length(depth, ”depth”);
    check_pos_length(height, ”height”);
```

Argument Checking Functions

Topic: Extending ACIS, C++ Interface

The API error checking functions that can be called are as follows:

check_array_exists ..... Error if NULL pointer to array

check_array_length ..... Error if zero length

check_body ..... Error if NULL pointer

check_coedge ..... Error if NULL pointer

check_delta ..... Error if NULL pointer

check_edge ..... Error if NULL pointer

check_entity ..... Error if NULL pointer

check_face ..... Error if NULL pointer

check_graph . . . . . Error if NULL pointer

check_plane . . . . . Error if NULL pointer

check_sheet . . . . . Error if NULL pointer

check_wire . . . . . Error if NULL pointer

check_non_neg_length . . . . Error if length less than SPAresabs

check_non_zero_length . . . Error if absolute length less than SPAresabs

check_pos_length . . . . . Error if length less than SPAresabs

check_3sides . . . . . Check # of sides > 3

check_wire_body . . . . . Checks wire pointer not NULL and shell pointer NULL

When to Check Arguments

Topic: Extending ACIS, C++ Interface

The question that must be evaluated regarding argument checking are:

- Does the programmer presume that the arguments being passed into the routine are valid?
- Does the programmer start by doing a check to see whether the arguments are reasonably valid?

Argument checking slows down the performance of the routine; however, when the check is omitted, an important debugging tool is lost. The developer must weigh these factors when writing APIs.

Error Handling

Topic: Extending ACIS, C++ Interface

In the following example, API make_parallelepiped is called to create a cuboid, and the result of the API is determined (using a conditional statement):

```
body = make_parallelepiped(width, depth, height);
result = outcome(body == NULL ? API_FAILED : 0);
```

If the operation was successful, the result is set to 0; otherwise, the result is set to a generic API failed condition. If some other internal error occurred, the result is set by the error handling code in the API_BEGIN macro. API_END closes the roll back and error handling that API_BEGIN established.

Refer to Chapter 6, *Error Handling and Messaging*, for more information about error handling.

Debugging Macro

Topic: Extending ACIS, C++ Interface

The following macro will print “calling” followed by the name of the API being called, if the user sets the debug level to print DEBUG_CALLS statements.

```
DEBUG_LEVEL(DEBUG_CALLS)
    fprintf(debug_file_ptr, ”calling api_make_cuboid\n”);
```

The preprocessor expands this macro call to:

```
if (api_module_header.debug_level >= 10)
    fprintf(debug_file_ptr, ”calling api_make_cuboid\n”);
```

The second call to the macro sets up debugging to print an additional “leaving” message, if the user sets the debug level to print DEBUG_FLOW statements.

```
DEBUG_LEVEL(DEBUG_FLOW)
    fprintf(debug_file_ptr, ”leaving api_make_cuboid: %s\n”,
            find_err_ident(result.error_number()));
```

The preprocessor expands this macro call to:

```
if (api_module_header.debug_level >= 20)
    fprintf(debug_file_ptr, ”leaving api_make_cuboid: %s\n”,
            find_err_ident(result.error_number()));
```

Deriving Classes

Topic: *Extending ACIS, *C++ Interface, *Entity

The class interface consists of all public ACIS classes and their methods (member functions). Class derivation allows developers to add new data to derived classes, and to define new methods for accessing the data.

This section describes how to extend ACIS by deriving new classes from existing ACIS classes. It includes an example of the classes that might be derived for assembly modeling.

Levels Of Derivation

Topic: Extending ACIS, C++ Interface

Many developers from different organizations derive new classes from existing ACIS classes. To reduce the potential for name collisions and to identify owning organizations of derived classes, the first derivation from an ACIS class is always an “organization class.” The organization class merely identifies the organization using a two or three-character *sentinel*, which is a unique identifier that prevents name collisions and identifies the “owner” of the class. The organization sentinel may identify a company or component. Refer to section *Sentinels* in this chapter.

Organization classes define no methods or data of their own. *Organization classes are never instantiated.* Application-specific classes are derived from the organization class. They define data and methods and can be instantiated.

The assembly modeling example illustrates the organization class, ENTITY_XYZ, and two application-specific classes, ASSEMBLY and INSTANCE.

Class Names

Topic: Extending ACIS, C++ Interface

Derived organization classes are generally named <*BASE–CLASS*>_<*sentinel*>, where <*BASE–CLASS*> is the name of the base class from which this class is derived, and *sentinel* is the organization sentinel. Application-specific and subsequent derived classes do not have any such naming restrictions.

In the assembly modeling example, the organization is named XYZ, and the organization class derived from ENTITY is named ENTITY_XYZ. The examples in this section derive two application-specific classes, ASSEMBLY and INSTANCE.

Files

Topic: Extending ACIS, C++ Interface

Each class has a .hxx header file that declares the class, includes necessary header files, and prototypes related functions. The header file contains a comment block that describes the class in a standard format. (ACIS header files are shipped in readable format with both object-code and source-code shipments.)

In the assembly modeling example, the application-specific class, ASSEMBLY, is defined in the assembly.hxx file.

Each class has a .cxx source file that implements all of the methods of the class that were not defined as inline methods within the body of the class. The methods file defines class-specific macros, and includes the .hxx class header file.

In the assembly modeling example the method for the ASSEMBLY class are implemented in the assembly.cxx file.

ENTITY Class Derivation

Topic: Entity, Extending ACIS, C++ Interface

ENTITY is the common class for all data structure entity definitions. It defines the base class for all permanent model types.

Macros are provided in the ENTITY class header file (entity.hxx) for simplifying the definition of user entities as classes derived from ENTITY. There are many class methods required for any entity. Many of them are the same for all entities, and many of the others have elements in common. The macros help with this commonality. Refer to Chapter 9, *Entity Derivation Macros*, for descriptions of these macros and how to use them.

Assembly Modeling Class Derivation Example

Topic: Extending ACIS, C++ Interface

This example is a simple implementation of the derived classes for an assembly modeler. This example is *not* an exhaustive treatment of assembly modeling. It just provides an example of classes derived from ENTITY that have some use.

The organization class is ENTITY_XYZ, and there are two application-specific classes, ASSEMBLY and INSTANCE.

Assembly Modeling

Topic: Extending ACIS, C++ Interface

Assemblies are a way of making complex parts that consist of repeated building blocks. For example, in steel frame manufacture, bolt groups frequently have to be modeled. It is highly advantageous to be able to model a single bolt and then *instance* the bolt many times as though it were in the different locations that form the bolt group. Two benefits of assembly modeling are that model space is reduced and that many instances are modified by editing or replacing the body to which an instance refers.

In the simple assembly example, an assembly contains a pointer to an instance and a transform. The transform allows the assembly to be moved cheaply with bodies. Each instance is made up of a pointer to a body and a transform. Instances are connected in a NULL-terminated doubly-linked list. Refer to the diagram in Figure 8-1.

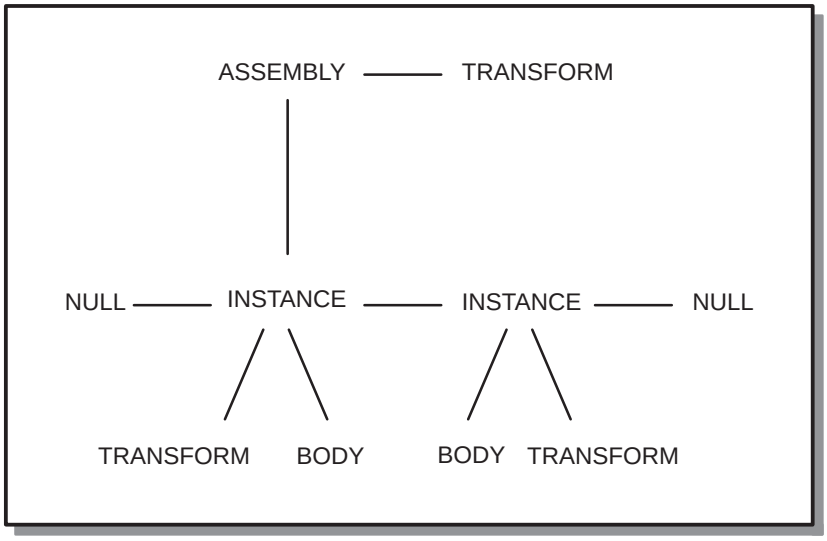

Figure 8-1. Assembly Pointers

To model a car as an assembly, one might create two bodies, CHASSIS and WHEEL, and then five instances, one pointing to CHASSIS and the other four pointing to WHEEL. The transform pointed to by each instance arranges the assembly so that there is an instance of WHEEL at each corner of the instance of CHASSIS.

Organization Class

Topic: Extending ACIS, C++ Interface

The organization class is called ENT_XYZ. Its purpose is to identify all classes derived from it as coming from the organization, in this example, XYZ. The abstract class is defined in the ent_xyz.hxx header file and the ent_xyz.cxx source file.

C++ Example

```
// ent_xyz.hxx

#if !defined( ENTITY_XYZ_CLASS )
#include ”kernel/logical.h”
#include ”kernel/kerndata/data/en_macro.hxx”

// Identifier used to find out (via identity() defined below) to
// what an entity pointer refers.
extern int ENTITY_XYZ_TYPE;

// Identifier that gives number of levels of derivation of this
// class from ENTITY.
#define ENTITY_XYZ_LEVEL (ENTITY_LEVEL + 1)

// The XYZ master data structure entity, of which all its private
// specific types are subclasses.
MASTER_ENTITY_DECL( ENTITY_XYZ, NONE )

#define ENTITY_XYZ_CLASS
#endif
```

Example 8-8. ent_xyz.hxx

C++ Example

```
//ent_xyz.cxx

#include <stdio.h>
#include ”kernel/acis.hxx”
#include ”kernel/dcl_kern.h”
#include ”kernel/kerndata/data/datamsc.hxx”
#include ”ent_xyz.hxx”
```

```
// Identifier used externally to identify a particular entity
// type. This is only used within the save/restore system for
// translating to/from external file format.

// Macros used to tell the master macro who we are and where we
// stand in the hierarchy
#define THIS() ENTITY_XYZ
#define THIS_LIB NONE
#define PARENT() ENTITY
#define PARENT_LIB KERN
#define ENTITY_XYZ_NAME "xyz"

// Now a grand macro to do the rest of the implementation.
MASTER_ENTITY_DEFN( "xyz master entity" );
```

Example 8-9. ent_xyz.cxx

Application–Specific Classes

Topic: Extending ACIS, C++ Interface

An application-specific class derivation defines each of the data elements with the class, and each of the methods that create, destroy, modify, or inquire these data elements. Application-specific classes are instantiated.

The two classes in this example (ASSEMBLY and INSTANCE) encapsulate the functionality of an assembly.

ASSEMBLY Class

Topic: Extending ACIS, C++ Interface

The assembly.hxx file contains the class declaration, data definitions, inline methods, and prototypes for non-inline methods. The assembly.cxx source file contains implementations for all of the class's non-inline methods.

C++ Example

```
// assembly.hxx

#if !defined( ASSEMBLY_CLASS )
#define ASSEMBLY_CLASS

#include "ent_xyz.hxx"
#include "kernel/kerndata/geom/transfrm.hxx"

class INSTANCE;

// Identifier that gives number of levels of derivation of this
// class from ENTITY
extern int ASSEMBLY_TYPE;
#define ASSEMBLY_LEVEL (ENTITY_XYZ_LEVEL + 1)
```

```
// ASSEMBLY declaration proper.
class ASSEMBLY: public ENTITY_XYZ
{
    // Pointer to the start of a list of instances
    INSTANCE *instance_ptr;

    // This transformation relates the local coordinate system
    // to the global one in which the assembly resides.
    TRANSFORM *transform_ptr;

    // Include the standard member functions for all entities.
    ENTITY_FUNCTIONS( ASSEMBLY, NONE )

    // Search a private list for this object, used for debugging.
    LOOKUP_FUNCTION

    // Now the functions specific to ASSEMBLY.

    // The assembly constructor initializes the record, and makes
    // a new bulletin entry in the current bulletin board to
    // record the creation of the assembly.
    ASSEMBLY( INSTANCE * = NULL );

    // Data reading routines.
    INSTANCE *instance() const { return instance_ptr; }
    TRANSFORM *transform() const { return transform_ptr; }

    // Data changing routines. Each of these routines checks
    // that the record has been posted on the bulletin–board
    // before performing the change. If not, the routine provokes
    // an error, so that the omission (forgetting to call backup()
    // first) can be rectified in the program. In production
    // versions of the program, these checks may be disabled, to
    // improve efficiency.
    void add_instance( INSTANCE * );
    void set_instance( INSTANCE * );
    void set_transform( TRANSFORM * );
};
#endif
```

Example 8-10. assembly.hxx

C++ Example

```
// assembly.cxx

#include <stdio.h>
#include <string.h>
#include <memory.h>
#include ”kernel/acis.hxx”
#include ”kernel/logical.h”
#include ”kernel/kerndata/geom/transfrm.hxx”
#include ”kernel/kerndata/data/datamsc.hxx”
#include ”assembly.hxx”
#include ”instance.hxx”

// Include the standard member functions via the usual macros.
// First declare our class name and that of our parent.
#define THIS() ASSEMBLY
#define THIS_LIB NONE
#define PARENT() ENTITY_XYZ
#define PARENT_LIB NONE

// External name for this entity type, for save/restore and
// general communications with the user.
#define ASSEMBLY_NAME ”assembly”

ENTITY_DEF( ASSEMBLY_NAME )
    // Print out a readable description of this body.
    debug_new_pointer( ”Instance list”, instance(), fp );
    debug_new_pointer( ”Transform”, transform(), fp );

LOOKUP_DEF

SAVE_DEF
    write_ptr( instance(), list );
    write_ptr( transform(), list );

RESTORE_DEF
    instance_ptr = (INSTANCE *)read_ptr();
    transform_ptr = (TRANSFORM *)read_ptr();

COPY_DEF
    instance_ptr = (INSTANCE *)list.lookup( from->instance() );
    transform_ptr = (TRANSFORM *)list.lookup( from->transform() );

SCAN_DEF
    list.add( instance() );
    list.add( transform() );
```

```
FIX_POINTER_DEF
    set_instance( (INSTANCE *)read_array( array, (int)instance() )
);
    set_transform( (TRANSFORM*)read_array( array, (int)transform()
) );

TERMINATE_DEF

// ********************************************************
//
// Now implement the members specific to ASSEMBLY, and those which
// are not part of the macros.
//
// ********************************************************

ASSEMBLY::ASSEMBLY(INSTANCE *inst)
{
    //printf("Assembly Constructor In\n");
    set_instance(inst);
    set_transform(NULL);

    //printf("Set back pointers in instances\n");
    // Set back pointers in instances
    INSTANCE *this_inst = instance();
    while (this_inst != NULL)
    {
        this_inst->set_assembly( this );
        this_inst = this_inst->next();
    }
    //printf("Assembly Constructor Out\n");
}

// Fix up a copy of this object for a change bulletin, after the
// object is constructed and copied memberwise.
void ASSEMBLY::fixup_copy(
            ASSEMBLY *rollback
        ) const
{
    PARENT()::fixup_copy( rollback );
}
```

```
// User ”destructor” for a ASSEMBLY. Performs type–specific work,
// then leaves the rest to the generic ENTITY destructor.
void ASSEMBLY::lose()
{
    transform_ptr–>lose();
                        // This record is itself rolled back, so we
                        // don’t set the pointer to NULL
    ENTITY::lose();
}


// Final record discard.
ASSEMBLY::~ASSEMBLY()
{
    check_destroy();
}


// Member–setting functions. Each ensures that the object has been
// backed up for rollback before making any change.
void ASSEMBLY::add_instance( INSTANCE * inst)
{
    //printf(”Assembly – Adding instance pointer\n”);
    backup();
    if ( inst == NULL )
    {
        printf(”Error – Cannot add a NULL instance\n”);
        return;
    } else {
        inst–>set_previous(NULL);
        inst–>set_next(instance());
        if ( instance() != NULL )
            instance()–>set_previous(inst);
        set_instance(inst);
        inst–>set_assembly(this);
    }
}


void ASSEMBLY::set_instance( INSTANCE * inst)
{
    //printf(”Assembly – setting instance pointer\n”);
    backup();
    instance_ptr = inst;
}
```

```
void ASSEMBLY::set_transform( TRANSFORM * trfm )
{
    //printf(”Assembly – Setting transform pointer\n”);
    backup();
    transform_ptr = trfm;
}
```

Example 8-11. assembly.cxx

INSTANCE Class

Topic: Extending ACIS, C++ Interface

Once the assembly classes and methods are defined, instances are created and initialized with proper data. The INSTANCE class is derived from ENTITY.

If this example were used to construct a room (assembly) full of chairs (instances), each instance’s body pointer would point to the template of a chair, the assembly pointer would point to the room, and the transform would locate each individual chair somewhere in the room.

When a constructor creates an instance, the instance’s template body, the assembly to which the instance belongs, and the spatial transformation are all initially NULL. Member routines provide a way to set and to inquire these parameters.

The instance.hxx header file prototypes the functions and includes other necessary header files. The instance.cxx source file implements the functions for instantiation.

C++ Example

```
// instance.hxx
#if !defined( INSTANCE_CLASS )
#define INSTANCE_CLASS

#include ”ent_xyz.hxx”
class BODY;
class TRANSFRM;
class ASSEMBLY;

// Identifier that gives number of levels of derivation of this
// class from ENTITY
extern int INSTANCE_TYPE;
#define INSTANCE_LEVEL (ENTITY_XYZ_LEVEL + 1)

// INSTANCE declaration proper.
class INSTANCE: public ENTITY_XYZ
    {
    // Pointer to the start of a list of instances
    ASSEMBLY *assembly_ptr;
```

```
    INSTANCE *next_ptr;
    INSTANCE *previous_ptr;

    // This transformation relates the local coordinate system to
    // the global one in which the assembly resides.
    TRANSFORM *transform_ptr;
    BODY *body_ptr;

    // Include the standard member functions for all entities.
    ENTITY_FUNCTIONS( INSTANCE, NONE )

    // Search a private list for this object, used for debugging.

    LOOKUP_FUNCTION
    // Now the functions specific to INSTANCE.
    // The instance constructor initializes the record, and makes
    // a new bulletin entry in the current bulletin board to
    // record the creation of the instance.
    INSTANCE( ASSEMBLY *ass = NULL, BODY *b = NULL,
        TRANSFORM *t = NULL);

    // Data reading routines.
    ASSEMBLY *assembly() const { return assembly_ptr; }
    INSTANCE *previous() const { return previous_ptr; }
    INSTANCE *next() const { return next_ptr; }
    TRANSFORM *transform() const { return transform_ptr; }
    BODY *body() const { return body_ptr; }

    // Data changing routines. Each of these routines checks
    // that the record has been posted on the bulletin-board
    // before performing the change. If not, the routine provokes
    // an error, so that the omission (forgetting to call backup()
    // first) can be rectified in the program. In production
    // versions of the program, these checks may be disabled, to
    // improve efficiency.
    void set_assembly( ASSEMBLY * );
    void set_previous( INSTANCE * );
    void set_next( INSTANCE * );
    void set_transform( TRANSFORM * );
    void set_body( BODY * );
};
#endif
```

Example 8-12. instance.hxx

C++ Example

```
// instance.cxx
#include <stdio.h>
#include <string.h>
#include <memory.h>
#include ”kernel/acis.hxx”
#include ”kernel/logical.h”

#include ”kernel/kerndata/geom/transfrm.hxx”
#include ”kernel/kerndata/top/body.hxx”
#include ”kernel/kerndata/data/datamsc.hxx”
#include ”assembly.hxx”
#include ”instance.hxx”

// Include the standard member functions via the usual macros.
// First declare our class name and that of our parent.
#define THIS() INSTANCE
#define THIS_LIB NONE
#define PARENT() ENTITY_XYZ
#define PARENT_LIB NONE

// External name for this entity type, for save/restore and
// general communications with the user.
#define INSTANCE_NAME ”instance”

ENTITY_DEF( INSTANCE_NAME )
    // Print out a readable description of this entity.
    debug_new_pointer( ”Owning assembly”, assembly(), fp );
    debug_new_pointer( ”Next instance”, next(), fp );
    debug_new_pointer( ”Previous instance”, previous(), fp );
    debug_new_pointer( ”Transform”, transform(), fp );
    debug_new_pointer( ”Body”, body(), fp );

LOOKUP_DEF

SAVE_DEF
    write_ptr( assembly(), list );
    write_ptr( transform(), list );
    write_ptr( body(), list );
    write_ptr( previous(), list );
    write_ptr( next(), list );

RESTORE_DEF
    assembly_ptr = (ASSEMBLY *)read_ptr();
    transform_ptr = (TRANSFORM *)read_ptr();
    body_ptr = (BODY*)read_ptr();
    previous_ptr = (INSTANCE*)read_ptr();
    next_ptr = (INSTANCE*)read_ptr();
```

```
COPY_DEF
    assembly_ptr = (ASSEMBLY *)list.lookup( from->assembly() );
    transform_ptr = (TRANSFORM *)list.lookup( from->transform() );
    body_ptr = (BODY*)list.lookup( from->body() );
    previous_ptr = (INSTANCE*)list.lookup( from->previous() );
    next_ptr = (INSTANCE*)list.lookup( from->next() );


SCAN_DEF
    list.add( assembly() );
    list.add( transform() );
    list.add( body() );
    list.add( previous() );
    list.add( next() );


FIX_POINTER_DEF
    set_assembly( (ASSEMBLY *)read_array( array,
        (int)assembly() ) );
    set_transform( (TRANSFORM*)read_array( array,
        (int)transform() ) );
    set_body( (BODY*)read_array( array, (int)body() ) );
    set_previous( (INSTANCE*)read_array( array,
        (int)previous() ) );
    set_next( (INSTANCE*)read_array( array, (int)next() ) );


TERMINATE_DEF


// ************************************************************
//
// Now implement the members specific to INSTANCE, and those which
// are not part of the macros.
//
// ************************************************************
INSTANCE::INSTANCE(ASSEMBLY *ass, BODY *b, TRANSFORM *t)
{
    //printf("Instance Created\n");
    set_assembly(ass);
    if ( ass != NULL )
    {
        ass->add_instance(this);
    }
    set_body(b);
    set_transform(t);
    set_previous(NULL);
}
```

```
// Fix up a copy of this object for a change bulletin, after the
// object is constructed and copied memberwise.
void INSTANCE::fixup_copy(
            INSTANCE *rollback
        ) const
{
    PARENT()::fixup_copy( rollback );
}

// User ”destructor” for an INSTANCE. Performs type–specific work,
// then leaves the rest to the generic ENTITY destructor.
void INSTANCE::lose()
{
    transform_ptr–>lose();
                            // This record is itself rolled back,
                            // so we don’t set the pointer to NULL
    ENTITY::lose();
}

// Final record discard.
INSTANCE::~INSTANCE()
{
    check_destroy();
}

// Member–setting functions. Each ensures that the object has been
// backed up for rollback before making any change.
void INSTANCE::set_previous( INSTANCE * inst)
{
    //printf(”Instance – Setting previous instance pointer\n”);
    backup();
    previous_ptr = inst;
}

void INSTANCE::set_next( INSTANCE * inst)
{
    //printf(”Instance – Setting next instance pointer\n”);
    backup();
    next_ptr = inst;
}

void INSTANCE::set_assembly( ASSEMBLY * ass )
{
    //printf(”Instance – Setting assembly pointer\n”);
    backup();
    assembly_ptr = ass;
}
```

```
void INSTANCE::set_body( BODY * b )
{
    //printf(”Instance – Setting body pointer\n”);
    backup();
    body_ptr = b;
}

void INSTANCE::set_transform( TRANSFORM * trfm )
{
    //printf(”Instance – Setting transform pointer\n”);
    backup();
    transform_ptr = trfm;
}
```

Example 8-13. instance.cxx

Versioning within ACIS

Topic: Extending ACIS, C++ Interface

ACIS development follows the policy that if there is a change in an ACIS API, which modifies the shape or topology of the model, we will ”version” that functionality.

A ”versioned” ACIS API allows the customer to call that API and specify which version of ACIS they would like to have executed. Providing such a facility within ACIS is important for customers who rely on a particular geometric result to maintain those results even if they migrate to a newer version of ACIS.

Important versioning details

Topic: Extending ACIS, C++ Interface

An ACIS version consists of a 3 digits: major, minor, and point. A major release will cause an increase in the major version, a minor release will cause an increase in the minor version, and an ACIS service pack will cause an increase in the point version.

While it is almost always the case that major and minor releases cause changes to the shape/topology of a model, Spatial service packs try as much as possible not to modify the topology or the shape of a model. However, it is possible that for a specific bug fix we will introduce a topology/shape change in a service pack.

If your application depends on a specific version of ACIS (for shape/topology reasons), you can use ACIS versioning to target that version. However, note that once Spatial adds versioning because of a shape/topology change, we will not make any fixes to older versioned algorithms. Therefore, be aware that using ACIS versioning to target a specific version of ACIS will prevent you from receiving the benefit of bug fixes in future service packs.

There is currently one API whose behavior is intentionally not versioned: api_check_entity. Therefore, customers should not use api_check_entity's results to determine the algorithmic flow of their application. Also, api_check_entity's performance is not guaranteed. Instead, api_check_entity is being continually refined to meet this simple goal: to provide the caller the most comprehensive verification of their model.

How to specify an ACIS version for a single API call

Topic: Extending ACIS, C++ Interface

Every ACIS API has an additional, optional parameter: AcisOptions *. This parameter allows the caller to control both how the API is journaled and how the API is versioned. Versioning an ACIS API may be done as follows:

```
// Run this api as ACIS 7.0
AcisOption ao(AcisVersion(7,0,0));
api_do_something(<param1>, <param2>, ..., <paramN>, &ao);
```

The above logic will construct an AcisOptions object on the stack. It also creates a temporary AcisVersion object and passes it to the AcisOptions constructor. This is the easiest method of constructing an AcisOption object that is set to a particular ACIS version. Here is an alternate method that is functionally equivalent:

```
// Run this api as ACIS 7.0
AcisOption ao;
ao.set_version(AcisVersion(7,0,0));
api_do_something(<param1>, <param2>, ..., <paramN>, &ao);
```

Alternate Methods of Versioning

Topic: Extending ACIS, C++ Interface

Alternate methods of versioning in ACIS are discussed in the following sections:

ALGORITHMIC_VERSION_BLOCK

Topic: Extending ACIS, C++ Interface

It is also possible to control the ACIS version for large blocks of code without having to pass an AcisOptions parameter to each API call. This is defined in base/baseutil/version/vers.hxx Consider the following example customer code:

```
void cust_func(void)
{
    outcome oc;

    oc = api_do_something(...);

    if (oc.ok())
        oc = api_do_something_else(...);

cust_func2(. . .);

}
```

To run the above logic as ACIS 6.3, change the code as follows:

```
void cust_func(void)
{
    outcome oc;

{
// Run everything in the following block as ACIS 6.3.
// When this code block is exited the version will revert
// to it's original setting.
    ALGORITHMIC_VERSION_BLOCK(AcisVersion(6,3,0));

    oc = api_do_something(...);

    if (oc.ok())
        oc = api_do_something_else(...);

    // Even ACIS calls within cust_func2 will be run as ACIS 6.3.
    cust_func2(. . .);
}
}
```

ALGORITHMIC_VERSION_BLOCK's can also be nested. When a nested version block is exited, the ACIS version will be reverted to the version given by the next outermost version block. When all version blocks are exited, the default version is the current ACIS version.

API_VERS_BEGIN

Topic: Extending ACIS, C++ Interface

API_VERS_BEGIN is a way to combine an API_BEGIN and ALGORITHMIC_VERSION_BLOCK in a single statement. An API_VERS_END should terminate the block. this is defined in: kern/kernel/kernapi/api/api.hxx
The following logic:

```
API_BEGIN
ALGORITHMIC_VERSION_BLOCK(AcisVersion(6,3,0));
// insert code here
API_END
```

is equivalent to this logic:

```
AcisOptions ao(AcisVersion(6,3,0));
API_VERS_BEGIN(&ao);
// insert code here
API_VERS_END
```

Run–time Virtual Methods

Topic: *Extending ACIS, *C++ Interface

Application developers often need to perform operations (such as display) that are not provided by ACIS. When the operation depends on the type of entity being operated upon, the developer has two implementation options. If they have source for ACIS, they can add a virtual method to ENTITY and its derived classes. Otherwise, they can provide a function which uses an if–else–if construct to determine the entity type and take appropriate action. Unfortunately, neither of these methods works well when two or more components may optionally be used together. For example, if component *A* defines a method or function to display entities, there is no way, without having source, for component *B* to tell component *A* how to display any new entity types it derives.

The *run-time virtual method* mechanism solves these problems by allowing components to define new virtual methods with run-time binding, similar to the way attributes allow data to be added to entities.

Design

Topic: Extending ACIS

When a new method is defined, an argument list class is derived from METHOD_ARGS to contain the arguments to be passed to the method in addition to the entity pointer. This provides for compile-time type checking. To ensure that the correct argument list class is passed to the method, each argument list class provides an id method which returns a string identifying the class.

A master list is maintained that associates an index value with each combination of method name and argument list identifier requested by an application. This list is maintained as a linked list sorted by method name. Multiple methods with the same name are further sorted by argument list identifier. A METHOD_ID class is used to look up entries in the master list. If the requested combination of method name and argument list identifier is not found, a new entry is added to the list and given the next available index value.

Each class derived from ENTITY has a dynamically allocated array of function pointers associated with it. When a method is registered for the class, this array is expanded if necessary and a pointer to the specified function is stored in the element indicated by the index value in the specified METHOD_ID.

When a method is called, the identifier for the supplied argument list item is first checked against the identifier in the specified METHOD_ID. If they do not match, FALSE is returned. Otherwise, the function at the element specified by the METHOD_ID in the function table for the class is called with the entity pointer and the specified argument list item. If the function table for the class does not contain the specified element of that element is the NULL pointer, the method is called for the parent of the class. If no ancestor of the class provides the method, FALSE is returned.

Usage

Topic: Extending ACIS

When defining a new run-time virtual method, as with defining any function, the developer must decide on the name for the method and its arguments. The only restriction for a run-time virtual method is that it must return a logical value indicating whether the operation was supported for the supplied entity type. This implies that any values to be returned by the operation must be passed back through the argument list.

The developer then derives a class from METHOD_ARGS to contain the arguments for the method. Since the major purpose of this class is to provide some compile-time type checking, not encapsulation, it is recommended that all data members be declared public for simplicity. In addition to a constructor, which may be used to provide default parameters for the method, this class must also provide a virtual id class which returns a character string used to provide additional type safety and allow a form of method overloading. The identifier string should match the class name in order to reduce the possibility of conflict with classes generated by other developers. Also, to reduce dependencies between components, it is recommended that all methods for the argument class be defined inline in the .hxx file.

Example 8-14 defines an argument class with all inline methods:

C++ Example

```
#include ”methargs.hxx”

class DL_item;
class SPAtransf;

class SKETCH_ARGS : public METHOD_ARGS
{
public:
    DL_item *&result;
    int color;
    const SPAtransf *transform;
```

```
    SKETCH_ARGS(DL_item *&res, int col,
        const SPAtransf *trans = NULL) :
        result(res), color(col), transform(trans) {}

    virtual const char *id() const { return ”sketch_args”; }
};
```

Example 8-14. methargs.hxx

Implementing Methods

Topic: Extending ACIS

To implement a method for an entity type, the developer first defines a function to perform the desired operation for a particular entity type or types. This function takes two arguments: a pointer to void containing the this pointer for the calling entity, and a const reference to METHOD_ARGS. These arguments can be cast to the appropriate types inside the function. The function returns a logical indicating whether or not the operation was performed. Refer to Example 8-15.

C++ Example

```
static logical
sketch_edge(
    void *entptr,
    METHOD_ARGS const &argref
    )

{
    EDGE *edge = (EDGE *) entptr;
    SKETCH_ARGS const &args = *(SKETCH_ARGS *) &argref;

    // Use edge, args.color, and args.transform to create a
    // DL_item storing a pointer to the DL_item in args.result

    // The body of this has been omitted for clarity.

    return TRUE;
}
```

Example 8-15. Define the Function

Next, the function is registered using the static add_method method for the appropriate entity types. This can be done using static initializers or an initialization function for the component.

C++ Example

```
// Get the identifier for a method named ”sketch” using
// SKETCH_ARGS
static METHOD_ID sketch_method_id(”sketch”, ”sketch_args”);
```

```
// Associate sketch function with sketch method for EDGE class
static MethodFunction xxx = EDGE::add_method(sketch_method_id,
    sketch_edge);
```

Example 8-16. Registering the Function

Calling Methods

Topic: Extending ACIS

To call a run-time virtual method, the developer calls an entity's call_method method with the appropriate METHOD_ID and METHOD_ARGS. This method returns a logical value indicating whether or not the operation was performed. The caller should check this value before attempting to use any return values from the run-time method. Refer to Example 8-17.

C++ Example

```
// Get the identifier for a method named ”sketch” using
// SKETCH_ARGS
static METHOD_ID sketch_method_id(”sketch”, ”sketch_args”);

// Execute the sketch method associated with this entity type
ok = entity->call_method(sketch_method_id, SKETCH_ARGS
    (item, color, trans));

// Check for supported operation before using result
if (ok) { . . . }
```

Example 8-17. Calling Methods

Third Party Products

Topic: *Extending ACIS

Because ACIS uses software component technology, third party vendors can create software component products that can be used with ACIS. These products are available directly from the third party vendors for such operations as faceting, graphics support, and data exchange.