# *Chapter 5.* Application Interfaces

Topic: Ignore

Most ACIS based applications are developed in C++. However, applications can also be developed in Scheme. C++ applications for Microsoft Windows platforms may also take advantage of the ACIS interface to Microsoft Foundation Classes (MFC).

## C++ Interface

Topic: *C++ Interface

C++ applications may interface to ACIS through Application Procedural Interface (API) functions, C++ classes, and, in some cases, Direct Interface (DI) functions. Developers may also extend ACIS by creating their own APIs and classes.

As Figure 5-1 illustrates, a C++ application is built on top of ACIS, interfacing to the modeler via APIs, classes, and DIs.

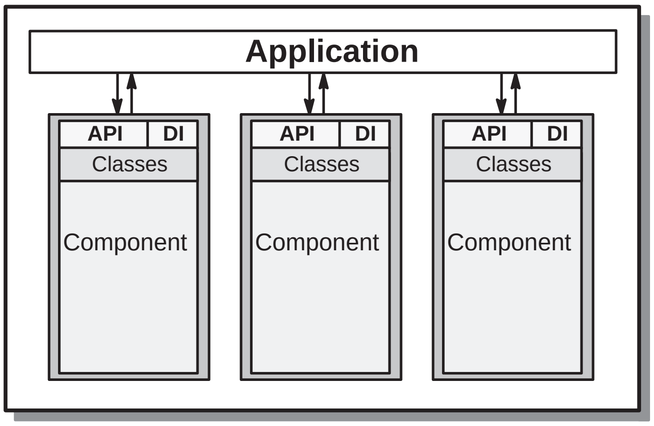

**Figure 5-1. C++ Application Interface to ACIS**

## API Functions

Topic: *C++ Interface

API functions provide the main interface between applications and ACIS. An API is a function that an application calls to create, change, or retrieve data. An API function combines modeling functionality with application support features such as argument error checking and roll back. When an error occurs in an API routine, ACIS automatically rolls the model back to the state before that API routine was called. This ensures that the model is left in an uncorrupted state.

The *callable interfaces* to API functions remain consistent from release to release, regardless of modifications to low-level ACIS data structures or functions.

Refer to the function summaries and reference templates for a complete list of all API functions and a detailed description of each. API function names begin with the prefix api_.

## Classes

Topic: *C++ Interface

The class interface is a set of C++ classes that are used to define the ACIS model geometry, topology, and other characteristics. The classes may be used by an application to directly interact with ACIS through their public and protected data members and methods (member functions). Refer to the class summaries and reference templates for a complete list of all classes and a detailed description of each.

The class ENTITY is a base class from which many ACIS classes are derived. It implements common data and functionality that is mandatory in all classes that are permanent objects in the model, although ENTITY does not itself represent any specific object. The ATTRIB class, which is derived from ENTITY, is used to derive specific attribute classes for system and user attributes. Refer to the *Kernel Component Manual* for information on attributes.

Developers may also derive application-specific classes from the ACIS classes for special purposes. Refer to the *3D ACIS Application Development Manual* for information about deriving classes.

## DI Functions

Topic: *C++ Interface

A DI function provides access to modeler functionality without the additional application support features of an API. Unlike APIs, these functions are not guaranteed to remain consistent from release to release.

DI functions are not available to access all of the functionality in ACIS. They are generally useful for performing operations that do not change the model, such as inquiries. DI functions are documented in function reference templates.

# Scheme Interface

Topic: *Scheme Interface

Scheme is an interpretive, public domain language, derived from LISP, that provides rapid and easy prototyping capabilities. Unlike many interpretive languages, it runs very efficiently. Most developers who use Scheme find little or no need to translate their Scheme code into C++ or another compiled language.

Because Scheme is interpretive, there is no compile or link phase for Scheme procedures and it can be used immediately for training. Scheme is particularly suited to control tasks such as user interface command processing. It is extensible and portable to all ACIS supported platforms.

The Scheme interface is a collection of functions that allows a Scheme-based application to access ACIS functionality. Applications interface to ACIS via the ACIS Scheme Interpreter (which is based on the Elk version of the Scheme language), which processes Scheme commands. Scheme commands may be native Scheme commands, ACIS Scheme extensions, or application-specific Scheme extensions created by the developer.

Figure 5-2 illustrates how a Scheme application is built on top of ACIS, interfacing to the Scheme support component via the Scheme Interpreter, which then interfaces to the rest of ACIS using C++ calls.

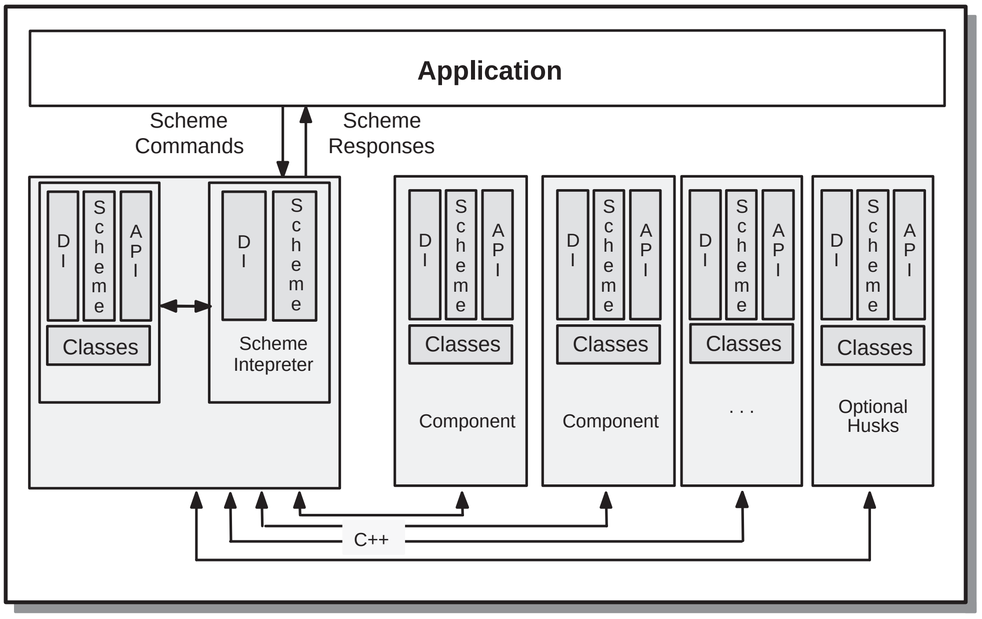

**Figure 5-2. Scheme Application Interface to ACIS**

Developers can create their own Scheme extensions and data types to extend Scheme. Refer to the *Scheme Support Component Manual* for more information. The Scheme data types and extensions are described in reference templates in online help.

Scheme ACIS Interface Driver Extension (Scheme AIDE) is a Scheme based ACIS demonstration application. Refer to Chapter 6, *Using Scheme AIDE*, and the *Scheme AIDE Main Program Component Manual* for information about using this application.

Figure 5-3 illustrates the flow of control between the end user and ACIS when using the Scheme interface.

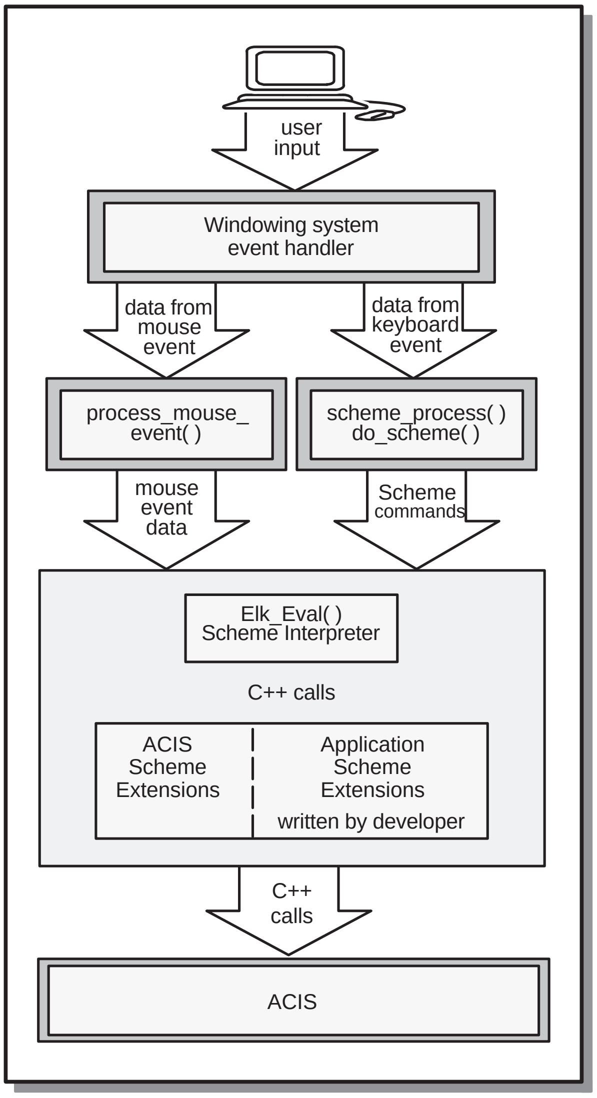

**Figure 5-3. Scheme Interface Flow of Control**