CAD Services Specification

March 2002

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

The OMG documentation is organized as follows:

OMG Modeling

- ***Unified Modeling Language (UML) Specification*** defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.
- ***Meta-Object Facility (MOF) Specification*** defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models.
- ***OMG XML Metadata Interchange (XMI) Specification*** supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information.

Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

CORBA: Common Object Request Broker Architecture and Specification

Contains the architecture and specifications for the Object Request Broker.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include specifications such as *Collection*, *Concurrency*, *Event*, *Externalization*, *Naming*, *Licensing*, *Life Cycle*, *Notification*, *Persistent Object*, *Property*, *Query*, *Relationship*, *Security*, *Time*, *Trader*, and *Transaction.*

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include specifications such as *Internationalization and Time*, and *Mobile Agent Facility.*

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA "plumbing" infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Currently, specifications are available in the following domains:

- *CORBA Business*: Comprised of specifications that relate to the OMG-compliant interfaces for business systems.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Healthcare*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.
- *CORBA Transportation*: Comprised of specifications that relate to the OMG-compliant interfaces for transportation systems.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Boeing Company
- CFD Research Corporation
- Dassault Systems
- Ford Motor Company
- Fraunhofer Institute - Production Systems and Design Technology
- GE
- IBM
- KAIST - Korean Advanced Institution of Science and Technology
- MSC.Software Corporation
- NASA, Glenn Research Center
- NIST
- Open Cascade
- Structural Dynamics Research Corporation (SDRC)
- TranscenData, An ITI Business
- Unigraphics Solutions

Overview *1*

Contents

This chapter contains the following sections.

1.1 Objective of this Specification

This specification is an interface standard for Mechanical Computer Aided Design (CAD) systems that enable the interoperability of CAD, Computer Aided Manufacturing (CAM) and Computer Aided Engineering (CAE) tools. The aim is to provide users of design and engineering systems the ability to seamlessly integrate, best-in-class, software across a wide variety of CAD/CAM and CAE applications through CORBA interfaces. These standard interfaces enable a distributed product design environment that includes a variety of CAD systems.

This specification focuses on establishing Mechanical CAD system interfaces that provide Geometry and Topology data to Analysis and Manufacturing applications and tools. The intent is to establish a series of high-level engineering interfaces that do not require low-level data structures to answer mechanical engineering queries. To avoid many of the problems associated with data translation, this specification provides CORBA interfaces with consistent functionality across native CAD implementations. To the maximum extent, all queries use native CAD system geometry kernels and associated software as illustrated below.

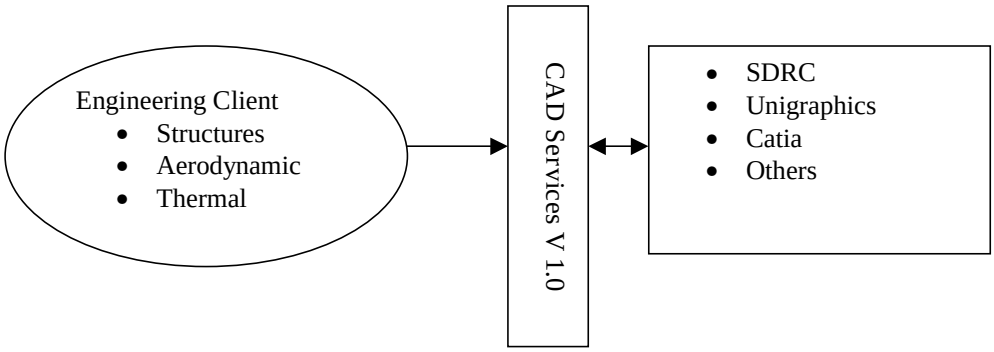


Figure 1-1 Illustration of CAD Services V1.0 being used as a neutral interface to native CAD geometry kernels.

1.2 Compliance Discussion

The compliance points are intended to help implementers and users of the standard identify the level of functionality provided by a given implementation. We assume that vendors will provide various levels of functionality depending on their business case and other considerations.

This specification provides several levels of compliance, as listed below.

Table 1-1 Table of Compliance Points.

Identifier	**Description**
Base	All interfaces in all modules except for CadGeometryExtens and the CadUserInterface interface in CadConnection Module.
GeometryExtensions	Base, plus all interfaces in CadGeometryExtens.
PersistantIdentifiers	Base, plus support for persistent identifiers as described in Section 2.2.2.2, “Model Operations,” on page 2-14 and Section 2.3.2.2, “Entity Operations,” on page 2-27.
UserInterface	Base, plus support for a user interface as described in Section 2.1.3.1, “CadSystem Operations,” on page 2-6.
Parametrics	Base, plus support for parametric regeneration of CAD entities as described in Section 2.6.2, “DesignFeature Interface,” on page 2-63.

If the UserInterface compliance point is not supported, the CadUserInterface interface becomes inaccessible (PdmSystem::get_gui throws GuiUnsupported) and its implementation becomes moot. Similarly, if the Parametrics compliance point is not

supported, the Parameter interface becomes inaccessible (DesignFeature::get_parameter_set returns an empty sequence). Otherwise, all interfaces called out by Base are mandatory.

Support for persistent identifiers, a user interface, or parametric regeneration of CAD entities are indicated by declaring support for the respective compliance points. Additionally, implementations may vary in their support for accuracy, presentation properties, and session-level identification of CAD entities. However, these variations are exposed through specified behaviors of the mandatory interfaces.

1.3 Overall Interface Model

CAD Services interfaces are divided into eight different modules with a significant inter-dependency between modules (as illustrated in Figure 1-2 on page 1-4).

The first of these modules is the **CadConnection** module that provides standard interfaces to connect with the CAD Services server. **CadConnection** interfaces provide access to **CadMain** module interfaces. **CadMain** interfaces include Model interfaces that provide support for assemblies. The **CadFoundation** module provides primary interfaces that are inherited by geometric entities. It also provides a general Attributable interface that can be used to "tag" geometric entities with application specific information through the use of DynAny local data structures. Basic geometric (tessellation) data structures and interfaces can be found in the **CadGeometry** module. The **CadGeometryExtens** module provides additional geometric entities that are subtypes of those in **CadGeometry**. Boundary representations (BREPs) can be found in the **CadBrep** module. **CadBrep** contains interfaces for solid models that are represented through Bodies, Faces, Edges, and others. These solid models expose parametric features that allow shape regeneration through interfaces in the **CadFeature** module. Finally, **CadUtility** provides a series of basic data structures used throughout the standard. One of these structures is a **CadError** exception. This exception can be raised by almost all of the CAD Services interfaces. This general exception is needed due to the wide dissimilarity of CAD system implementations and the variability of native CAD API support.

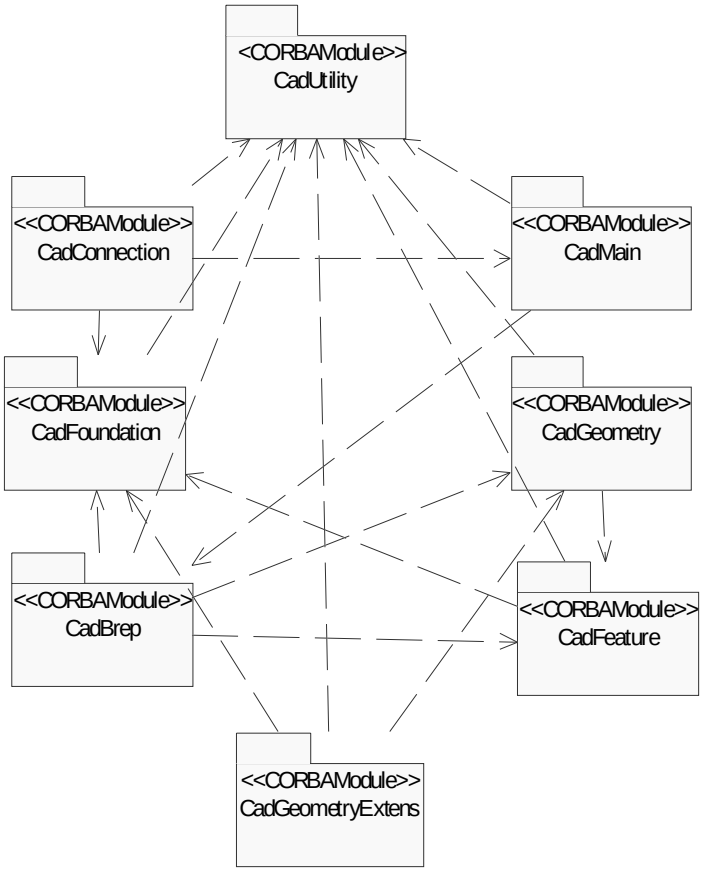


Figure 1-2 Module Interdependencies and Overall Module Structure

CAD Modules and Interfaces 2

Contents

This chapter contains the following sections.

2.1 CadConnection Module

This module provides high-level connectivity to the **Model** interface in the **CadMain** module. The first interface used by any client is the **CadServer** interface. This is a lightweight interface used primarily to connect with the **CadSystem** interface and provides two readonly attributes that contain information on the native CAD system and properties needed to activate the underlying native CAD system. Uniform connection operations allow high-level consistency between dissimilar native CAD implementations.

2.1.1 UML Diagram

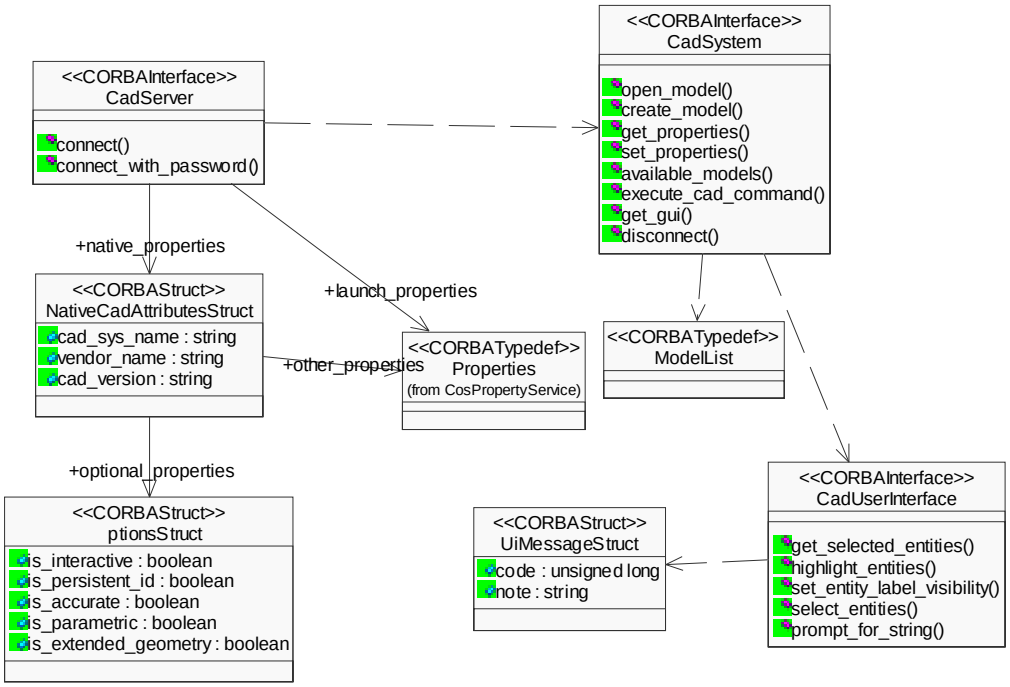


Figure 2-1 UML Diagram of the CadConnection Interfaces and Data Structures

2.1.2 CadServer Interface

CadServer is the factory object to any CAD Services implementation. Its sole intent is to provide a client application with a mechanism to connect and launch the desired CAD system's implementation of CAD Services. This connectivity approach was designed primarily to allow each **CadServer** to support only one native CAD system; however, there is sufficient flexibility in the interfaces to allow connection to more that one native CAD system. For example, one **CadServer** might provide access to multiple native CAD systems by using information passed through the **CosPropertyService::Properties** data sequence (see below for a description of the **CosProperties::Properties** data sequence). How many and what types of native CAD systems are supported by the **CadServer** depends on the implementation.

```
module CadConnection
{

//forward references

interface CadServer;
interface CadSystem;
interface CadUserInterface;
```

```
interface CadServer
 {
  readonly attribute NativeCadAttributesStruct native_properties;
  // NativeCADSys contains CAD Vendor name, version, etc.

  readonly attribute CosPropertyService::Properties
      launch_properties;
  // This attribute contains all needed information to launch
  // the CAD system

   CadSystem connect( in CosPropertyService::Properties props)
        raises ( CadConnectionFault );
   // secure connection

   CadSystem connect_with_password(  in string user,
        in string password,
        in CosPropertyService::Properties props )
        raises ( CadConnectionFault );
   // open wire (unsecure connection)
 };
```

2.1.2.1 CadServer Attributes

The **CadServer** interface has two readonly attributes. The first attribute **native_properties** is a data struct that identifies the native CAD system.

```
struct NativeCadAttr
 {
    string  cad_sys_name;
    string  vendor_name;
    string  cad_version;
    OptionsStruct optional_properties;
    CosPropertyService::Properties  other_properties;
 };
 struct Options
 {
   boolean is_interactive;
   boolean is_persistent_id;
   boolean is_accurate; // for some calculated properties
   boolean is_parametric; // design feature support
boolean is_extended_geometry;
 };
```

- The name (**cad_sys_name**), vendor (**vendor_name**) and native version number (**cad_version**) are represented by strings. The **OptionsStruct** data structure identifies support for compliance points. The final value is a reference

(**other_properties**) to a **CosPropertyService::Properties** data structure. This reference provides additional property information that is not specified in the standard, but may be useful to implementers.

- The **OptionsStruct** data structure identifies support for optional features in this **CadServer**:
  - **is_interactive** – TRUE indicates support for the UserInterface compliance point defined in Section 1.2.
  - **is_persistent_id** – TRUE indicates support for the PersistentIdentifiers compliance point defined in Section 1.2.
  - **is_accurate** – TRUE indicates support for the accuracy parameter in many calculated properties (for example, see the area operation of the **CadBrep::Face** interface).
  - **is_parametric** – TRUE indicates support for the Parametrics compliance point defined in Section 1.2.
  - **is_extended_geometry** - TRUE indicates support for the GeometryExtensions compliance point defined in Section 1.2.
- The second attribute is a **CosPropertyService::Properties** (**launch_properties**). This data type provides all information needed to activate the underlying native CAD geometry kernel. Often a user logging in to a CAD system must specify data like project, work group, CAD data repository (PDM), etc. If the information in the Properties sequence is incomplete for the target system, an exception (CadConnectionFault) is thrown. This exception is also provided with more specific names such as: ValidationError, InvalidProperties, and PermissionDenied.

```
typedef string PropertyName;
    struct Property
    {
        PropertyName property_name;
        any property_value;
    };
  sequence<Property>Properties;
```

The **CosPropertyService::Properties** data structure is shown above. It provides a simple sequence of name – value pairs.

2.1.2.2 CadServer Operations

To work effectively with CORBA security services, two different login semantics are available to the client – both of which can be controlled by the system administrator. If CORBA security services are available, and the system administrator does not want user names and passwords to be passed across the network, a client can use **CADServer::connect** (in **CosPropertyService::Properties** props) and the **CADServer** will use the userId and password from the security services as the CAD user name and the CAD password. This assumes that the system administrator has set up the correct logins in the CAD system and in the CORBA security service so these names and passwords are synchronized. To prevent clients from using the **connect_with_password**, implementers may wish to use a local interceptor.

In environments where the security service is unavailable the **connect_with_password** operation may be used. It is identical to the connect interface but requires a user name and password.

2.1.2.3 Exception

A CadConnectionFault exception shall be thrown to clearly identify connection problems. Similar exceptions are also provided with more specific names such as: ValidationError, InvalidProperties, and PermissionDenied. Each exception provides an unsigned long and a string to convey information on the fault as noted in Section 2.1.3.3, “Exceptions,” on page 2-7.

```
exception CadConnectionFault
 {
   unsigned long error_code;
string error_text;
 };
```

2.1.3 CadSystem Interface

CadSystem is an implementation of a particular CAD system, geometry engine, or kernel. It provides high-level functionality enabling a client to open and operate on a CAD model. This interface also provides a mechanism to execute operations not defined in this specification and a mechanism to list various CAD models that are available. The standard does not specify directory information or any hierarchical approach to listing the available models. It is expected that each CAD Server implementation may wish to use a configuration file (or similar mechanism) to specify one or several directories where appropriate models will be placed.

```
enum ActivationMode
 { ACTIVE_READONLY, ACTIVE_CHECKOUT, ACTIVE_DETAILED };

interface CadSystem
 {
   CadMain::Model  open_model(in string model,
      in ActivationMode access)
      raises (InvalidModel, PermissionDenied);
   // CAD model related operations


  CadMain::Model create_model(in string new_name,
      in CadUtility::MassUnit m_unit,
      in CadUtility::LengthUnit l_unit,
       in CadFeature::ParameterSeq model_params)
      raises (PermissionDenied, BadParameters, CadUtility::CadError);
  // Creates and opens a new model in native CAD system

  CosPropertyService::Properties get_properties();
 void set_properties( in CosPropertyService::Properties props );
```

```
    // Allows reading and changing of CadSystem properties

    ModelList  available_models();
    // returns a list if model available to the active  CAD System.

    void execute_cad_command(in string command_string ,
       inout any comm_out ) raises (BadCommand);
    // extensible Cad system command interface

  CadUserInterface get_gui() raises(CadFoundation::GuiUnsupported);
    // access to GUI interface

    void disconnect()
       raises ( CadConnectionFault );
    // disconnect from this CadSystem
  };
```

2.1.3.1 CadSystem Operations

open_model	Returns a **CadMain::Model** interface for a given input string (model) and **ActivationMode** (access). The enumeration **ActivationMode** identifies whether the client needs read-only access (ACTIVE_READONLY) or full read-write access (ACTIVE_CHECKOUT). ACTIVE_DETAILED can be used to assign implementation-specific policies. These implementation specific policies may use the authorization credentials enforced by the **connect()** operation of the **CadServer** interface. For example, a user may have authority to modify only one part in an assembly and this authorization would be assigned during the connect operation as part the input **CosPropertyService::Properties** sequence (props). This operation loads the specified model into memory and can be unloaded by invoking **close_model()** on the **Model** object reference.
create_model	Returns a new **CadMain::Model** interface with the input string **new_model** used as the name of the new model. This input string has the same semantics as a string returned from **ModelList**, but may not require directory information (a default directory may be concatenated). The client must have full read-write access to the **CadServer**. **CadUtility::MassUnit m_unit** and **CadUtility::LengthUnit l_unit** must be specified for the new model. Model level parameters can be specified through a **CadFeature::ParameterSeq model_params**. Incorrect parameters shall result in an error, **BadParameters,** that clearly identifies any faults.
get_properties	Returns a **CosPropertyService::Properties** sequence detailing implementation defined properties similar to or the same as the properties used in the **CadServer:Connect** operation.

set_properties	Accepts a **CosPropertyService::Properties** sequence that sets implementation defined properties similar-to or the-same-as the properties used in the **CadServer:Connect** operation.
available_models	Returns a **ModelList** (a sequence of strings). The strings capture directory path information or any other unique identifying information for each native CAD model.
execute_cad_command	Accepts a command string that will be executed in the native CAD system. Additional input information and a response can be passed through the **comm_out** parameter. This operation provides access to some functionality that may be supported by the underlying native CAD system, but is not provided for in this standard.
get_gui	Returns a reference to the **CadUserInterface**. An implementation that does not support the UserInterface compliance point shall throw **CadFoundation::GuiUnsupported** when this operation is invoked.
disconnect()	Disconnect from the **CadServer** and perform clean-up operations that are implementation specific.

2.1.3.2 ModelList

This sequence of strings conveys information that uniquely identifies the models available to the **CadSystem** interface. Implementations shall provide any necessary directory and name information in each string.

```
typedef sequence<string> ModelList;
```

2.1.3.3 Exceptions

These exceptions are thrown by **CadSystem** operations to identify specific errors and faults. An unspecified **error_code** and **error_text** provide additional error documenting properties.

```
// various exceptions

 exception ValidationError
 {
  unsigned long error_code;
  string error_text;
 };

 exception InvalidProperties
 {
  unsigned long error_code;
  string error_text;
 };

 exception PermissionDenied
```

```
 {
  unsigned long error_code;
  string error_text;
 };

 exception InvalidModel
 {
  unsigned long error_code;
  string error_text;
 };

 exception BadCommand
 {
  unsigned long error_code;
  string error_text;
 };
exception BadParameters
 {
   CadFeature::ParameterSeq flawed_params;
   CadUtility::StringSeq reasons;
 };
```

2.1.4 CadUserInterface Interface

This is an optional interface that supports interaction with the User Interface of the underlying native CAD system. As with many CAD Services V1.0 interfaces, a CadUtility::CadError exception can be thrown from several operations.

```
struct UiMessageStruct
 {
   unsigned long code; //recommended
   string note;
 };

 interface CadUserInterface
 {

  CadFoundation::EntitySeq get_selected_entities()
      raises (CadUtility::CadError);
  // returns a sequence of entities selected in UI

  CadUtility::WarningStructSeq highlight_entities(
      in CadFoundation::EntitySeq marked);
  // highlight these entities in the UI. returns warning if all
  // entities cannot be highlighted

  void set_entity_label_visibility (
     in CadFoundation::EntitySeq entities,in boolean visibility)
     raises (CadUtility::CadError);
  // Sets the visibility of entity labels.
```

```
    CadFoundation::EntitySeq select_entities (in UiMessageStruct prompt,
        out UiMessageStruct error_message)
        raises (CadUtility::CadError);
    //prompt explains the selection request

    UiMessageStruct prompt_for_string (in UiMessageStruct prompt)
        raises (CadUtility::CadError);
    // Prompts the user to input a value (in UI) which is returned as
    // an unsigned long (recommended) string(optional).
  };
};
```

2.1.4.1 Operations

get_selected_entities	Returns a sequence of the geometry entities selected in the native CAD user interface. Returns an empty sequence if no CAD Services geometric entities are selected within the native user interface.
highlight_entities	Highlights, in the native CAD user interface, the input sequence of CAD Services entities. The native CAD user interface shall refresh (or repaint) after highlighting. The returned Boolean will be false, if all entities cannot be highlighted.
set_entity_label_visibility	Sets the visibility (TRUE = visible) of the input sequence of CAD Services entities' labels (**native_label** on **CadFoundation::Entity**). The native CAD user interface is shall (or repaint) after changing the label visibility.
select_entities	Returns a sequence of entities selected in the native CAD user interface after a *prompt* (**UiMessageStruct**) is used to display information in this user interface. The **error_message** data struct provides a mechanism to return any native error messages. **UiMessageStruct** is a data structure that can input or output an unsigned long as a message code (recommended) that can also be supported by a string *note*. The unsigned long is recommended to minimize international language constraints.
prompt_for_string	Returns a **UiMessageStruct** data struct for a similar input struct. This operation allows a client to display, in the native user interface, something (for example, a dialog box) created from the **UiMessageStruct** *prompt*. Information transfer using the unsigned long is recommended to minimize international language constraints.

2.2 CadMain Module

CadMain interfaces include the **Model** interface that is a recursive interface supporting parts and assemblies through operations that identify, if this model contains other Models (ModelInstances). For example, the **CadSystem::available_models()** operation might return a sequence of strings that identifies an assembly file that contains various

parts within that assembly. A **CadSystem::open_model** operation on this assembly file would also load the various parts contained within the *Model* and make them available to the client through a **Model::model_children()** operation.

2.2.1 UML Diagram

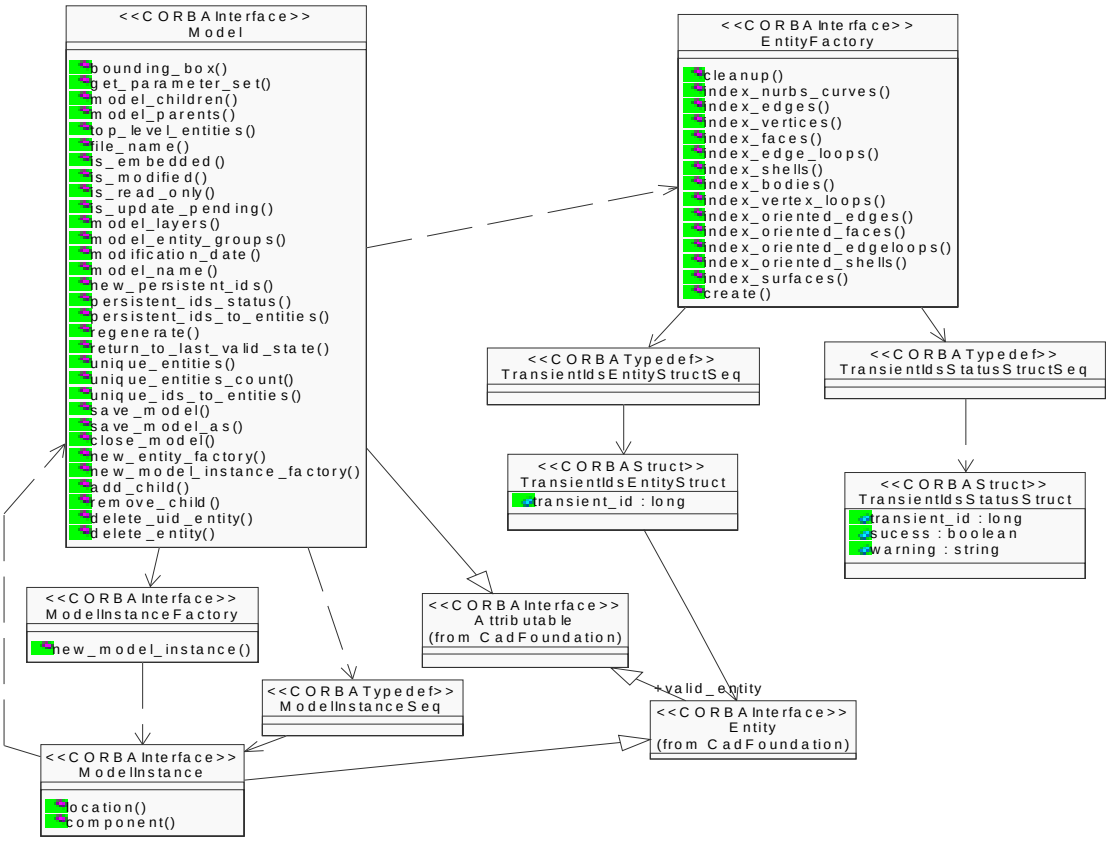


Figure 2-2 UML Diagram of CadMain Module Interfaces and Data Structures

2.2.2 Model Interface

The model interface is an aggregation of all CAD entities and high-level behaviors that represent a single CAD model. This interface includes product structure, boundary representations, geometric entities, features, text entities, and datums. It also supports assemblies by containing **ModelInstances** that are models contained within the **Model**. Much as a top-level **Model** might be an assembly that contains several parts that are also models. All entities within a CAD model are arc wise connected unless related to each other through a **ModelInstance**.

The **Model** interface has access operations to read the mass and length units. It also provides several operations to retrieve references to the CAD geometry entities contained in the **Model**. Several of the operations in this interface can throw CadUtility::CadError exceptions.

```
module CadMain

{
// forward references

  interface ModelInstance;
interface Model;

interface  Model : CadFoundation:: Attributable
  {
    // An aggregation of all entities and high-level behaviors that
    // represent a single CAD model.  Includes product structure,
    // boundary representations, geometric entities, features, text
    // entities, and datums.   All entities within a CAD model are
    // arcwise connected unless related through an instance.

    readonly attribute CadUtility::MassUnit  mass_unit;
    readonly attribute CadUtility::LengthUnit length_unit;
    // Defines units used in the Model.

    CadUtility::BoundingBox bounding_box (
        in CadUtility::TypeCodeSeq entity_types) raises
        (UnboundedEntity, NotValidCadType,CadUtility::CadError);
    // Returns an approximate BoundingBox around all entities of
    // the specified type(s) in the model.

CadFeature::ParameterSeq get_parameter_set()
raises (CadUtility::CadError);
    // Returns a sequence of parameters for this model.

    ModelInstanceSeq model_children()
        raises (CadUtility::CadGeneralError);
    // Returns a sequence of any ModelInstances contained in this
    // model.

    ModelInstanceSeq model_parents()
        raises (CadUtility::CadGeneralError);
    // Returns a sequence of parent models.

    CadFoundation::EntitySeq top_level_entities (
        in CadUtility::TypeCodeSeq entity_types)
        raises (NotValidCadType, CadUtility::CadError);
```

```
    // Returns a sequence of the top level entities of the specified
    // type(s).

    string file_name () raises (CadUtility::CadError);
    // Returns the complete name, including absolute path (if
    // possible), of the physical file that stores this model.
    // Returns an empty string if the model is not defined in a file.

    boolean is_embedded();
    // indicates if Model is an embedded part (e.g. CATIA Dittos,
    // ACAD blocks, etc) or a non-embedded part (e.g. parts in a ProE
    // assembly).

    boolean is_modified () raises (CadUtility::CadError);
    // Queries if the model has been modified since last saved.

    boolean is_read_only ()  raises (CadUtility::CadError);
    // Queries whether the model can be modified.

    boolean is_update_pending () raises (CadUtility::CadError);
    // Queries if the model is being updated (regenerated).

    CadFoundation::LayerSeq model_layers ()
        raises (CadUtility::CadError);
    // Returns a sequence of the Layers defined in this model.

  CadFoundation::EntityGroupSeq model_entity_groups (
  out CadUtility::StringSeq group_names)
        raises (CadUtility::CadError);
    // Returns a sequence of the EntityGroups defined in this model.

    string modification_date () raises (CadUtility::CadError);
    // Returns the date and time the model was last modified.

    string model_name () raises (CadUtility::CadError);
    // Returns the user-interpretable name of this model.

    CadUtility::StringSeq new_persistent_ids (
        in CadUtility::StringSeq persistent_ids)
        raises (CadUtility::CadError);
    // Returns a sequence of new persistent ID's for any entities not
    // referenced in the specified sequence of persistent ID's
    // (e.g. new or modified IDs).

    PidStatusSeq persistent_ids_status (
        in CadUtility::StringSeq persistent_ids)
        raises (CadUtility::CadError);
    // Returns whether the entities a sequence of persistent Ids
    // reference are unmodified, modified, deleted.  The returned
    // sequence of status enumerations are in
    // the same order as the input sequence of persistent IDs.
```

```
CadFoundation::EntitySeq persistent_ids_to_entities (
    in CadUtility::StringSeq persistent_ids)
    raises (CadUtility::CadError);
// Returns a sequence of entity objects corresponding to a sequence
// of persistent IDs. The returned sequence of entity objects
// is in the same order as the input sequence of persistent IDs.
// A NULL item in this sequence means an entity was not available
// for the corresponding persistent ID.

void regenerate () raises
    (RegenerationException, CadUtility::CadError);
// If any DesignFeatures exist and have been modified, forces a
// regeneration of modified Entities in the model.  Otherwise
// this operation does nothing. Throws an exception
// if the regeneration process is unsuccessful.



void return_to_last_valid_state ()
    raises (ReturnToValidFail, CadUtility::CadError);
// Returns the model and all entities it contains to their state
// just after the last successful regeneration
// Throws an exception if unable to return to a valid state.

CadFoundation::EntitySeq unique_entities (
     in CadUtility::TypeCodeSeq entity_types)
     raises (NotValidCadType, CadUtility::CadError);
// Returns a sequence of the unique entities of the
// specified type in sequential order.

unsigned long unique_entities_count (
      in CadUtility::TypeCodeSeq entity_types)
      raises (NotValidCadType, CadUtility::CadError);
// Returns the count of unique entities of the specified
// type(s) in this model.

CadFoundation::EntitySeq unique_ids_to_entities (
    in CadUtility::LongSeq unique_ids)
    raises (NotValidCadType, CadUtility::CadError);
// Returns (in sequential order) a sequence of entities
// corresponding to an input sequence of unique IDs.

void    save_model() raises (SaveFault);
void    save_model_as(in string new_name) raises (SaveAsFault);
void    close_model() raises ( CloseFault );
// operations for saving and terminating  an active session

EntityFactory new_entity_factory ()
    raises (CadUtility::CadError);
// Entity creation factory interface - called to create new CAD
```

```
    // entities in current model

  ModelInstanceFactory new_model_instance_factory ()
  raises (CadUtility::CadError);
    //Creates the ModelInstanceFactory, which is used to add ModelInstances

    void add_child(in Model child_model) raises (CadUtility::CadError);

    void remove_child(in Model child_model) raises (CadUtility::CadError);

    void delete_uid_entity(in long uid)
      raises (EntityOutOfModel, CadFoundation::UidUnsupported,
      CadUtility::CadError);
    // Removes ModelInstance, BrepEntity, Curve or Surface from the model

    void delete_entity(in CadFoundation::Entity entity)
       raises (EntityOutOfModel, CadUtility::CadError);
    // Removes ModelInstance, BrepEntity, Curve or Surface from the model
  };
```

2.2.2.1 Model Attributes

- **CadUtility::MassUnit mass_unit** - This readonly attribute describes the mass units used in the model.
- **CadUtility::LengthUnit length_unit** - This readonly attribute describes the length units used by all entities in the model.

2.2.2.2 Model Operations

bounding_box	For a given sequence of entity types **entity_types**, this operation returns a bounding box that describes the spatial limits of all the entities of the specified type. For example, a reference to a Body type would return a bounding box for all geometric bodies within the model. **entity_types** is a sequence of **CORBA::TypeCode**. This operation throws a NotValidCadType exception if the type in the **TypeCode** is not a valid geometric entity; that is, derived from **CadFoundation::Entity**. Also throws an UnboundedEntity exception for unbounded geometric entities. An empty **entity_types** sequence implies all types are to be processed and a bounding box for all entities should be returned.
get_parameter_set	Returns a sequence of parameters (Section 2.6.2, “DesignFeature Interface,” on page 2-64) associated with this model.
model_children	Returns a sequence of **ModelInstances** (Section 2.2.3, “ModelInstance Interface,” on page 2-17) contained in this model (children). This operation provides support for assemblies.

model_parents	Returns a sequence of **ModelInstances** (Section 2.2.3, "ModelInstance Interface," on page 2-17) that contain this model (parents). This operation provides support for assemblies.
top_level_entities	Returns a sequence of the top level CAD entities of the specified type(s). Type information is passed through a **CadUtility::TypeCodeSeq** that provides a **TCKind** enumeration of CORBA types (only the types derived from **CadFoundation::Entity** are valid geometric entities). The returned top level entities should identify separate entities that can be navigated to lower levels of the geometric hierarchy. An empty input **CadUtility::TypeCodeSeq** shall result in a system exception (**BAD_PARAM**).
file_name	Returns the complete name, including absolute path (if possible), of the physical file that stores this model. Returns an empty string if the model is not defined in a file.
is_embedded	Returns a Boolean that indicates if this Model is an embedded part (for example, CATIA Dittos, ACAD blocks, etc.) or a non-embedded part (for example, parts in a ProE assembly). If TRUE, save operations shall throw an exception.
is_modified	Returns a Boolean flag that is true if the model has been modified; that is, the regeneration operation has been successfully called. False indicates unmodified.
is_read_only	Queries whether the model can be modified. True indicates that the model can only be read. False indicates full read-write access.
is_update_pending	Flag indicating if the model is currently being updated; that is, the model has been modified, but not regenerated. TRUE indicates that there has been a feature modification to the model (or part of the model) that requires a regeneration operation request (see below).
model_layers	Returns a sequence of the Layers defined in this model. Layers are commonly used by CAD systems to group various CAD entities.
model_entity_groups	Returns a sequence of the EntityGroups contained in this model. The out parameter **group_names** is a sequence of names (**CadUtiltiy::StringSeq**) in the same order as the EntityGroups in the returned **EntityGroupSeq**.
modification_date	Returns a string indicating the date of the latest modification for this model. The ISO 8601 standard is mandatory, providing a format: YYYY-MM-DD hh:mm:ssZ (for example, 2001-02-26 13:20:55Z). With Z indicating "Zulu time" or Greenwich Mean Time (GMT).
model_name	Returns a string providing a user-interpretable name of this model

new_persistent_ids	Returns a sequence of new persistent IDs for any entities not referenced in the specified sequence of persistent IDs, presumably new or modified IDs. An implementation that does not support the PersistentIdentifiers compliance point shall throw PidUnsupported when this operation is invoked.
persistent_ids_status	Returns a sequence of status enumerations (Section 2.2.7, “Data Structures,” on page 2-23) that indicate if the input sequence of persistent IDs are UNMODIFIED, MODIFIED, DELETED. The returned sequence of status enumeration is in the same order as the input sequence of persistent IDs. The “unmodified” status should be as accurate and robust as possible; that is, any uncertainty should err toward a “modified” status. If no determination can be made, an UNDEFINED is returned for each reference. An implementation that does not support the PersistentIdentifiers compliance point shall throw PidUnsupported when this operation is invoked.
persistent_ids_to_entities	Returns a sequence of entity object references (**CadFoundation::EntitySeq**) corresponding to a sequence of persistent Ids (**CadUtility::StringSeq**). The returned sequence of entity objects is in the same order as the input sequence of persistent IDs. A NULL item in this sequence means an entity was not available for the corresponding persistent ID. An implementation that does not support the PersistentIdentifiers compliance point shall throw PidUnsupported when this operation is invoked.
regenerate	If any of the entities contained in this model support **CadFeature::DesignFeatures** and have been modified, this operation forces a regeneration of modified entities in the model. Otherwise this operation does nothing. Throws an exception (RegenerationException) if the regeneration process is unsuccessful.
return_to_last_valid_state	Returns the model and all entities it contains to their state just after the last successful regeneration. This does not provide an undo capability due to the variability between native CAD systems. This operation should be implemented completely above the native API (by saving / stacking parameter values when they are modified). Throws an exception (ReturnToValidFail) if unable to return to a valid state.
unique_entities	Returns a sequence of the unique entities of the specified type in sequential order. If the input type is not a valid geometric entity, throws an exception (NotValidCadType) that indicates the incorrect types. An empty input **CadUtility::TypeCodeSeq** shall result in a system exception (**BAD_PARAM**).

unique_entities_count	Returns the count of unique entities of the specified type(s) in this model. If the input type is not a valid geometric entity, throws an exception (NotValidCadType) that indicates the incorrect types. An empty input **CadUtility::TypeCodeSeq** shall result in a system exception (**BAD_PARAM**).
save_model	Performs a save operation in the native CAD system to preserve any changes in the model. Throws an exception (SaveFault), if the operation fails.
save_model_as	Accepts a new string name to save the model in the native CAD system. Throws an exception (SaveFault), if the operation fails.
close_model	Operation that provides for any clean-up activities and closes the model in the native CAD system. This operation does not save any changes.
new_entity_factory	Operation that instances an **EntityFactory** (Section 2.2.4, “EntityFactory Interface,” on page 2-18) to create new CAD entities in current model.
new_model_instance_factory	Returns a **ModelInstanceFactory** (Section 2.2.5) that enables the creation of new **ModelInstance**s.
add_child	Operation permits the insertion of a Model into the existing Model. Typically used to support assemblies.
remove_child	Operation permits the removal of a Model from the existing Model. Typically used to support assemblies.
delete_uid_entity	Operation deletes the geometric entity that is specified using a Unique ID. Deletion of a high-level entity deletes all dependant, lower-level entities.
delete_entity	Operation deletes the specified geometric entity (**CadFoundation::Entity**, section 2.3.2). Deletion of a high-level entity deletes all dependant, lower-level entities.

2.2.3 ModelInstance Interface

This interface provides placement and component information to support assemblies. Each *ModelInstance* is a *Model* and provides geometry services through the *Model* that is returned from the *ModelInstance::component()* operation.

```
interface ModelInstance : CadFoundation:: Entity
 {
  CadUtility::TransformationStruct location()
     raises (CadUtility::CadError);
  // returns location information

  Model component() raises (CadUtility::CadError);
  // Returns the Model that defines this instance.
 };
```

2.2.3.1 ModelInstance Operations

location	Returns a **CadUtility::TransformationStruct** (Section 2.4.2, "Tessellation Data Structures," on page 2-34) that provides exact location information.
component	Returns the model defining this instance.

2.2.4 EntityFactory Interface

The **EntityFactory** interface provides services to create multiple geometric entities. These geometric entities are built upon NURBS defined structures (as described in Section 2.4, "CadGeometry Module," on page 2-32). Due to the differing underlying CAD systems, the creation process is done in two steps. First, the various geometric entities are indexed through a index_xxx operation. The returned indexes (**CORBA::long**) can be used as input parameters to other index_xxx operations to generate "interconnected" geometric entities. Second, create is called on the **EntityFactory** to create the actual geometric entities in the native CAD system. This interface provides CadUtility::CadError and IncorrectIndex exceptions for most operations.

Vendors may implement this interface to allow the use of Unique IDs (that can be accessed through the CadFoundation::Entity interface) in place of the index values returned from a particular index_xxx operation. Support for this feature shall be clearly identified in implementation notes from the vendor.

```
interface EntityFactory
 {
   void cleanup()  raises (CadUtility::CadGeneralError);
   // clean-up any expensive book-keeping following multiple
   // index_xxxx(), create() cycles

   CadUtility::LongSeq index_nurbs_curves(
      in CadUtility::NurbsCurveStructSeq nurbs)
      raises (CadUtility::CadError);
   CadUtility::LongSeq index_edges (
      in CadUtility::LongSeq start_vertices,
      in CadUtility::LongSeq end_vertices,
      in CadUtility::LongSeq curve, in CadUtility::BooleanSeq sense)
      raises (IncorrectIndex, CadUtility::CadError);
   CadUtility::LongSeq index_vertices(
      in CadUtility::PointStructSeq pts)
      raises (CadUtility::CadError);
   CadUtility::LongSeq index_faces(
      in CadUtility::LongSeqSeq oriented_eloops,
      in CadUtility::LongSeq vertex_loops,
      in CadUtility::LongSeq surfaces)
      raises (IncorrectIndex, CadUtility::CadError);
   CadUtility::LongSeq index_edge_loops(
      in CadUtility::LongSeqSeq oriented_edges)
```

```
      raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_shells (
      in CadUtility::LongSeqSeq oriented_faces)
      raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_bodies (
      in CadUtility::LongSeq oriented_shells)
      raises (IncorrectIndex, CadUtility::CadError);
    CadUtility::LongSeq index_vertex_loops (
      in CadUtility::LongSeq vertices)
      raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_edges(
      in CadUtility::LongSeq edges)
      raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_faces(
      in CadUtility::LongSeq faces)
      raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_edgeloops(
      in CadUtility::LongSeq edgeloops, in CadUtility::BooleanSeq sense)
      raises (IncorrectIndex,  CadUtility::CadError);
    CadUtility::LongSeq index_oriented_shells(
      in CadUtility::LongSeq ofaces,
      in CadUtility::in CadUtility::BooleanSeq sense)
      raises (IncorrectIndex, CadUtility::CadError);

    CadUtility::LongSeq index_surfaces(
      in CadUtility::NurbsSurfaceStructSeq nurbs)
      raises (CadUtility::CadError);

    void create (out TransientIdsStatusStructSeq status_flags,
      out TransientIdsEntityStructSeq final_entities)
      raises (CadUtility::CadError);
    // final creation step
  };
};
```

2.2.4.1 EntityFactory Operations

cleanup	Operation cleans-up any indexing stored on the server between multiple index and create cycles.
index_nurbs_curves	Initial operation that generates transient Ids associated with the creation of NURBS curves. Input parameter is a sequence of **NurbsCurveStruct** and the associated transient Ids (**CadUtility::LongSeq**) are returned.
index_edges	Operation that generates transient Ids associated with the creation of **CadBrep::Edge** objects. Input parameters require an Id sequence of starting and ending Vertices (**start_vertices** and **end_vertices**) and an Id sequence of curves (**CadGeometry::Curve**) with associated senses (**CadUtility::BooleanSeq**). A sequence of associated transient Ids for edges is returned.
index_vertices	Operation that generates transient Ids associated with the creation of **CadBrep::Vertex** objects. Input parameters require a sequence of points (**CadUtility::PointStructSeq**). A sequence of associated transient Ids for vertices is returned.
index_faces	Operation that generates transient Ids associated with the creation of **CadBrep::Face** objects. Input parameters require an Id sequence of oriented edgeloops (**CadBrep::OrientedEdgeLoop**), an Id sequence of vertex loops (**CadBrep::VertexLoop**) and surfaces (**CadBrep::Surface**). A sequence of associated transient Ids for faces is returned.
index_edge_loops	Operation that generates transient Ids associated with the creation of **CadBrep::EdgeLoop** objects. Input parameters require an Id sequence of oriented edges (**CadBrep::OrientedEdgeSeq**). A sequence of associated transient Ids for edge loops is returned.
index_shells	Operation that generates transient Ids associated with the creation of **CadGeometry::Shell** objects. Input parameters require an Id sequence of oriented faces (**CadGeometry::OrientedFaceSeq**). A sequence of associated transient Ids for shells is returned.
index_bodies	Operation that generates transient Ids associated with the creation of **CadBrep::Body** objects. Input parameters require an Id sequence of oriented shells (**CadBrep::OrientedShellSeq**). A sequence of associated transient Ids for bodies is returned.
index_oriented_edges	Operation that generates transient Ids associated with the creation of **CadBrep::OrientedEdge** objects. Input parameters require an Id sequence of edges (**CadBrep::EdgeSeq**) and a sequence of booleans (**CadUtiltity::BooleanSeq**) indicating the sense of each edge. A sequence of associated transient Ids for oriented edges is returned.

index_oriented_faces	Operation that generates transient Ids associated with the creation of **CadBrep::Face** objects. Input parameters require an Id sequence of faces(**CadBrep::FaceSeq**) and a sequence of booleans (**CadUtiltity::BooleanSeq**) indicating the sense of each face. A sequence of associated transient Ids for oriented faces is returned.
index_oriented_edgeloops	Operation that generates transient Ids associated with the creation of **CadBrep::EdgeLoop** objects. Input parameters require an Id sequence of edges (**CadBrep::EdgeSeq**) and a sequence of booleans (**CadUtiltity::BooleanSeq**) indicating the sense of each edgeloop. A sequence of associated transient Ids for edge loops is returned.
index_oriented_shells	Operation that generates transient Ids associated with the creation of **CadBrep::OrientedShell** objects. Input parameters require an Id sequence of oriented faces (**CadBrep::OrientedFaceSeq**) and a sequence of booleans (**CadUtiltity::BooleanSeq**) indicating the sense of each oriented face. A sequence of associated transient Ids for oriented shells is returned.
index_surfaces	Operation that generates transient Ids associated with the creation of **CadGeometry::Surface** objects. Input parameters require a sequence of NURBS surfaces (**CadUtility:: NurbsSurfaceStructSeq**). A sequence of associated transient Ids for surfaces is returned.
create	This operation is invoked following the appropriate “indexing” of the various CAD entities to be created. Two data structures are returned. One sequence of status messages (**TransientIdsStatusStructSeq status_flags**) details whether the creation of the desired entity was successful. The second sequence provides a mapping of transient Ids to successfully created CAD entities (**TransientIdsEntityStructSeq final_entities**).

2.2.5 ModelInstanceFactory Interface

The ModelInstanceFactory interface provides the ability to create ModelInstances which are used to support assemblies. A ModelInstanceFactory is created from the Model interface (Section 2.2.4) and allows the input of location information.

```
interface ModelInstanceFactory
 {
   CadMain::ModelInstance new_model_instance (
```

```
    in CadUtility::TransformationStruct  global_location) raises
    (CadUtility::CadError);

  // Creates a new ModelInstance with initial transformation according the global

 // coordinate system

};
```

new_model_instance	Returns a created ModelInstance using the input **global_location** (**CadUtility::TransformationStruct**, Section 2.7)

2.2.6 Exceptions

The **CadMain** module supports a series of exceptions designed to be “self-describing” including the widely used CadUtitltiy::CadError exception (Section 2.7, “CadUtility Module,” on page 2-66).

```
exception RegenerationException
 {
  string reason;
  any support;
 };
 exception ReturnToValidFail
 {
  string reason;
 };
 exception UnboundedEntity {};

 exception  NotValidCadType
 {
  CadUtility::TypeCodeSeq bad_types;
 };

 exception SaveFault
 {
  string error_text;
 };

 exception SaveAsFault{
  string error_text;
 };

 exception CloseFault{
  string error_text;
 };
```

RegenerationException	This exception is thrown whenever the native CAD system fails to regenerate the Cad model. These exceptions are usually associated with incorrect changes to the DesignFeatures of various CAD entities. The *reason* string returns any native CAD system error message with the *support* any data structure provided to identify the precise problem area.
ReturnToValidFail	The ReturnToValidFail exception returns a *reason* string that contains any native CAD system error message.
UnboundedEntity	Unbounded entities throw this exception for operations querying the bounds (limits) of the entity.
NotValidCadType	Identifies **CORBA::TypeCode** references that are not valid CAD entities.
SaveFault	Provides an **error_text** string to list any native CAD system error messages.
SaveAsFault	Provides an **error_text** string to list any native CAD system error messages.
CloseFault	Provides an **error_text** string to list any native CAD system error messages.

2.2.7 Data Structures

```
enum PidStatus
 {
  UNMODIFIED,
  MODIFIED,
  DELETED,
  UNDEFINED
 };

 struct TransientIdsStatusStruct
 {
  // data structures supporting EntityFactory create output mapping

  long transient_id;
  boolean success;
  string warning;
 };

 struct TransientIdsEntityStruct
 {
  long transient_id;
  CadFoundation::Entity valid_entity;
 };
```

```
typedef sequence<TransientIdsStatusStruct>
  TransientIdsStatusStructSeq;
typedef sequence<TransientIdsEntityStruct>
  TransientIdsEntityStructSeq;
typedef sequence<PidStatus>     PidStatusSeq;
typedef sequence<ModelInstance> ModelInstanceSeq;
typedef sequence<Model> ModelSeq;
```

PidStatus	Enumeration of possible states for CAD entities as indexed using a persistent ID.
TransientIdsStatusStruct	Data struct used by EntityFactory interface to identify successful create operations and provide warning messages.
TransientIdsEntityStruct	Data struct that provides references to the created Cad entities.

2.3 CadFoundation Module

CadFoundation defines general elements and behavior that are shared by all CAD geometry entities. It also provides interfaces that support grouping of CAD entities through *Layer* (entities with shared color) or application specific groupings, **EntityGroup**.

2.3.1 UML Diagram

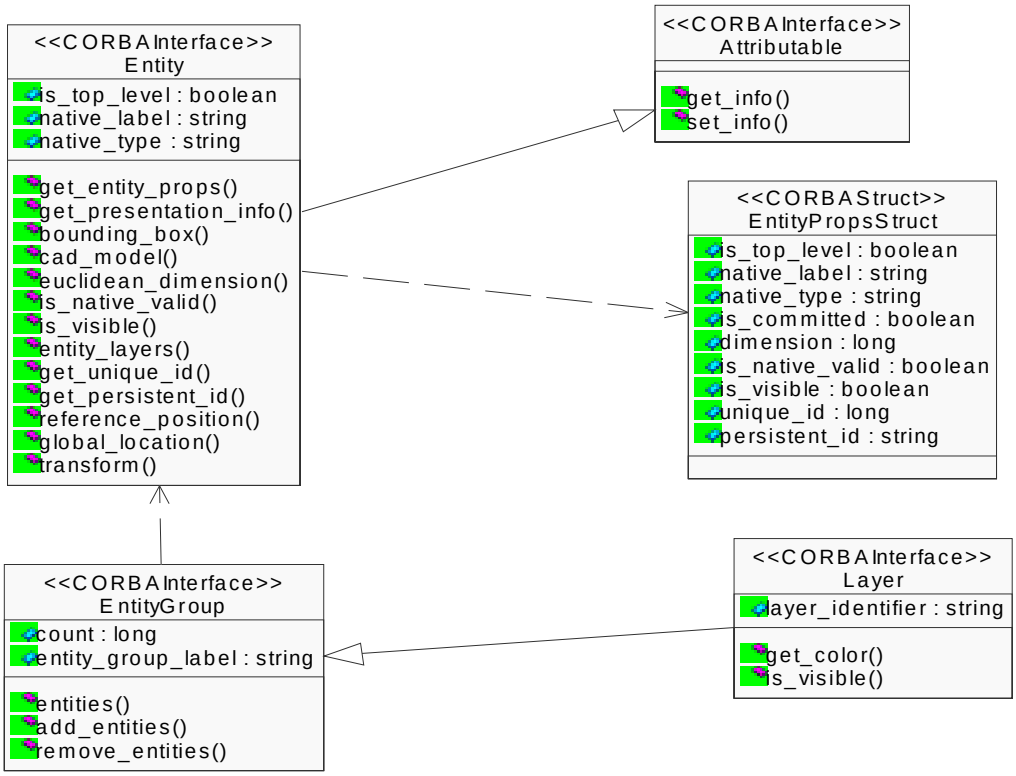


Figure 2-3 UML Diagram of CadFoundation Module

2.3.2 Entity Interface

The **Entity** Interface is a base CAD object that provides common behavior and properties that are inherited by specialized CAD objects. Most operations of this interface provide a CadUtility::CadError exception.

```
module CadFoundation

{

 // Encapsulates general elements and behavior that are shared by all model
entities.

interface Entity : Attributable
 {
   // Provides CAD functionality + properties to be inherited by
   // geometry objects

   EntityPropsStruct get_entity_props() raises (CadUtility::CadError);
   // operation providing grouped access to Entity properties
```

```
readonly attribute boolean is_top_level;
// Top level entity?

readonly attribute string native_label;
// Provides a brief description of the entity using system-
// specific terminology. Not guaranteed to be unique within
// a model.

readonly attribute string native_type;
// The system-specific type name of this entity.

CadUtility::PresentationStruct get_presentation_info()
    raises (CadUtility::CadError);
// Struct containing relevant presentation information.

CadUtility::BoundingBox bounding_box ()
   raises (UnBoundedEntity, CadUtility::CadError);
// Returns an approximate BoundingBox around the entity.
// Returns an error if the entity is unbounded in one or more
// directions.

Object  cad_model() raises (CadUtility::CadError);
// Returns the CadMain::Model object that contains this entity.
// Reference must be narrowed to CadMain::Model

long euclidean_dimension ()  raises (CadUtility::CadError);
// Returns the Euclidean dimension of the entity.

boolean is_native_valid () raises (CadUtility::CadError);
// Queries if the native entity is valid according to any internal
// checks provided by the CAD system.

boolean is_visible ()  raises (CadUtility::CadError);
// Queries whether the entity is visible or not (blanked, no-
// showed, hidden).

LayerSeq entity_layers ()  raises (CadUtility::CadError);
// Returns a sequence of the Layers that contain this entity

long get_unique_id() raises(UidUnsupported,CadUtility::CadError);
// Identifier that is guaranteed to be unique across all entities
// in a Model. This identifier is not persistent.

string get_persistent_id ()
   raises (PidUnsupported, CadUtility::CadError);
// Returns an identifier intended to identify this entity between
// interface sessions.

CadUtility::PointStruct reference_position ()
   raises (CadUtility::CadError);
```

```
// Returns a struct of the coordinates of a single, reference
// location on the entity that should be unique relative to
// neighboring entities.

  CadUtility::TransformationStruct global_location()
     raises (CadUtility::CadError);
  // Provides global coordinate location information

void transform (
    in CadUtility::TransformationStruct  transformation)
    raises (NotIndependent, ReadOnlyEntity, CadUtility::CadError);
// Applies the specified transformation (rotations and
// translation) to the entity. Throws an exception if the entity
// cannot be transformed.


};
```

2.3.2.1 Entity Attributes

is_top_level	Boolean flag indicating whether this entity is a top level entity in the geometric hierarchy. Convenient starting point for downward traversals.
native_label	String that provides a brief description of the entity using system-specific terminology that is not guaranteed to be unique within a model.
native_type	String containing the native CAD system specific type name of this CAD entity.

2.3.2.2 Entity Operations

get_entity_props	Operation that returns a data struct encapsulating all Entity properties. Minimizes the number of distributed calls needed to obtain basic Entity properties. The returned **EntityPropsStruct** data struct is listed in Section 2.3.6, “Exceptions and Data Structure,” on page 2-31.
get_presentation_info	Returns a **CadUtility::PresentationStruct** containing all presentation information on this Entity.
bounding_box	Returns a bounding box that describes the spatial limits of this entity. Should throw an UnboundedEntity exception for unbounded geometric entities.
cad_model	Returns a **CORBA::Object** object reference that contains a reference to the Model associated with this entity. Clients will need to narrow this object reference to the **CadMain::Model** scope.
euclidean_dimension	Returns the Euclidean dimension of this entity.
is_native_valid	Queries if the native CAD entity is valid according to any internal checks provided by the CAD system. True = valid.

is_visible	Queries whether the entity is visible or not. True = visible.
entity_layers	Returns a sequence of the **CadMain::Layers** that contain this entity.
get_unique_id	Returns an identifier that is guaranteed to be unique across all entities in a Model. This identifier is not persistent; that is, valid only during **CadServer** Session. Throws a UidUnsupported exception if a unique identifier is not supported.
get_persistent_id	Returns an identifier that can be used to identify this entity between sessions. An implementation that does not support the PersistentIdentifiers compliance point shall throw PidUnsupported when this operation is invoked..
reference_position	Returns a struct (**CadUtility::PointStruct**) of the coordinates of a single, reference location on the entity that is unique relative to neighboring entities.
global_location	Returns a struct (**CadUtility::TransformationStruct**) that provides global coordinate location information.
transform	Applies the specified transformation (rotations and translation) to the entity. Transformation is specified in a **CadUtility::TransformationStruct**. Throws an exception (NotIndependent or ReadOnlyEntity) if the entity cannot be transformed.

2.3.3 Attributable Interface

The **Attributable** interface provides functionality through inheritance to the Entity (**CadFoundation** Module) interface. The purpose is to allow very flexible attribute tagging for all geometric entities. The use of an “any” basic data structure is intended for DynAny data transfer. This flexible data structure provides a mechanism for Engineering Applications (as well as others) to “tag” or label the geometric entity with application specific information (for example, cost, maximum load, other data).

```
interface Attributable
 {
   // General interface allowing geometry tagging
   // The following operations should use DynAnys to extract
   // attribute information

   any  get_info() raises (CadUtility::CadError);
   void set_info(in any dyn_value) raises (CadUtility::CadError);
 };
```

get_info	Returns an *any* that contains a **DynAny**. **DynAny** is a locality-based data structure that enables run-time extensions (OMG document **formal/01-02-45**). The **DynAny** is converted to a **CORBA::any** when passed between client and server.
set_info	Input *any* parameter sets the value for this interface.

2.3.4 EntityGroup Interface

This interface provides a mechanism to group entities within a model (not associated with layering). Provides a mechanism for grouping whose semantics lie outside the standard (for example, application-specific collections of geometry and text).

```
interface EntityGroup
 {
   // A generalized grouping of entities within a model that is not
   // related to layering. Provides a mechanism for grouping whose
   // semantics lie outside the standard, e.g. application-specific

   readonly attribute long count;
   // Number of entities in group.

readonly attribute string entity_group_label;
// a label for this entity group

CadFoundation::EntitySeq entities ()
      raises (CadUtility::CadError);
   // Returns a sequence of entities defined in this group.

   void add_entities (in CadFoundation::EntitySeq entities)
       raises (CadUtility::CadError);
   // Adds the specified entities to this group.

   void remove_entities (in CadFoundation::EntitySeq entities)
      raises (CadUtility::CadError);
   // Removes the specified entities from this group.
   // Does not delete the entity objects.
 };
```

2.3.4.1 EntityGroup Attributes

count	Readonly attribute that provides a count of the entities in this group.
entity_group_label	Readonly attribute that provides a label for the entity group.

2.3.4.2 EntityGroup Operations

entities	Returns a sequence of entities defined in this group.
add_entities	Adds an input sequence of entities (**CadFoundation::EntitySeq**) to this group.
remove_entities	Removes the input sequence of entities (**CadFoundation::EntitySeq**) from this group, but does not delete the entities.

2.3.5 Layer Interface

Layers are used to organize and group various CAD entities. They share common presentation properties.

```
interface Layer : EntityGroup
 {
  // An collection of entities that corresponds to the layers.

  readonly attribute string layer_identifier;
  // string identifier of layer.

  CadUtility::ColorStruct get_color()
      raises (CadUtility::CadError);

  boolean is_visible() raises (CadUtility::CadError);
 };
```

layer_identifier	A readonly attribute indicating the string identifier of the layer.
get_color	Returns a struct (**CadUtility::ColorStruct**, Section 2.7, “CadUtility Module,” on page 2-66) providing color information.
is_visible	Boolean flag indicating visibility of this layer. TRUE = visible.

2.3.6 Exceptions and Data Structure

The following exceptions are defined within **CadFoundation** module:

```
exception UidUnsupported {};

exception PidUnsupported {};

exception GuiUnsupported {};

exception UnBoundedEntity {
     string unbounded_name;
};

exception NotIndependent {
     string dependency;
};
exception ReadOnlyEntity{};
```

UidUnsupported	Exception indicates that a unique identifier is not supported by this implementation.
PidUnsupported	Exception thrown when the client invokes an operation that requires the PersistentIdentifiers compliance point on an implementation that does not support it.
GuiUnsupported	Exception thrown when the client invokes an operation that requires the UserInterface compliance point on an implementation that does not support it.
UnBoundedEntity	Exception indicates that the entity is unbounded in at least one dimension.
NotIndependent	Exception indicates that this entity is dependent on other entities. Its geometry characteristics must be derived from other entities. The string *dependency* defines the dependency on other entities.
ReadOnlyEntity	Exception indicates that this entity can not be modified.

The following data struct (**EntityPropsStruct**) supports access to many of a CAD entity's properties through a single operation. The various data members correspond to the access operations reviewed in Section 2.3.2, "Entity Interface," on page 2-25.

```
struct EntityPropsStruct
 {
   // Properties of Entity in a struct

   boolean is_top_level;
   string native_label;
   string native_type;
   boolean is_committed;
   CadUtility::PresentationStruct presentation_info;
   long euclidean_dimension;
   boolean is_native_valid;
   boolean is_visible;
   long unique_id;
   string persistent_id;
   CadUtility::PointStruct position;
 };
```

2.4 CadGeometry Module

This module contains basic geometric data structures and interfaces that are used throughout the CAD Services interfaces. The two primary interfaces in this module are Surface and Curve. They inherit common functionality through the **CadFoundation::Entity** interface. A common use for either of these interfaces is to establish the exact three-dimensional location of the surface or curve through a point

query projection. These operations receive a sequence of three-dimensional point locations and return a sequence of points closest to the particular curve or surface. Most operations of this interface provide a CadUtility::CadError exception.

2.4.1 UML Diagram

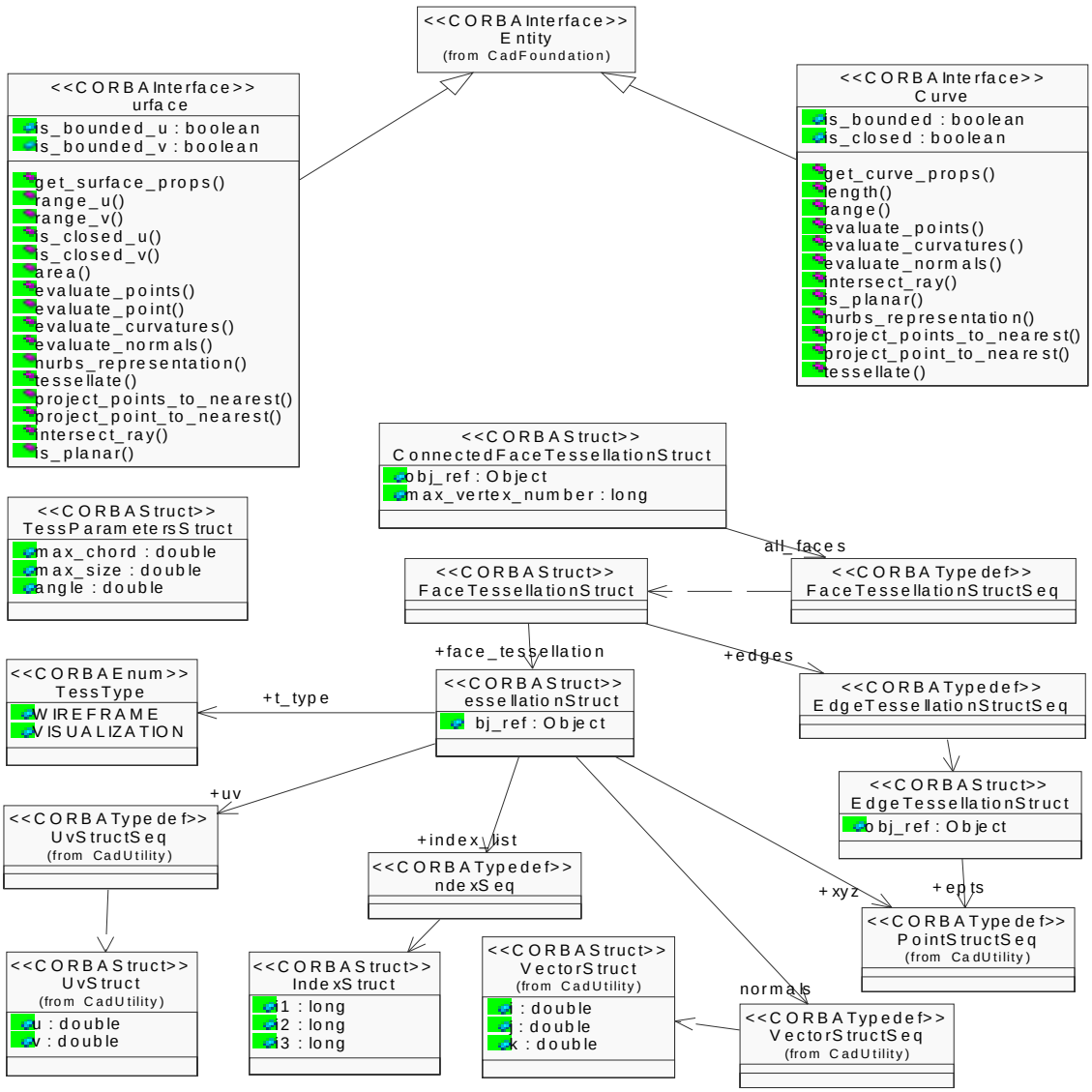


Figure 2-4 UML Diagram of CadGeometry Interfaces and Data Structures

2.4.2 Tessellation Data Structures

Tessellation data structures are commonly used for visualization of the CAD geometry, but these structures become significantly more useful if developed using specified parameters that can tailor the data structure to the application. These data structures capture some information about the underlying geometry and allow clients to easily access common groupings of information.

Please see Appendix A for an example of the Tessellation indexing.

```
module CadGeometry
{
 // Fundamental Geometry definitions

 //forward references
 interface Curve;
 interface Surface;

typedef sequence<Curve>       CurveSeq;
typedef sequence<Surface>     SurfaceSeq;

enum TessType
 {
  // an enumeration of possible types of tessellations

  WIREFRAME,
  VISUALIZATION
 };

 struct TessParametersStruct
 {
  // parameters used with the Tessellation creation

  double max_chord;
  // maximum deviation between triangle center and surface

  double max_size;

  double angle;
  // deviation between normals of facets - in degrees
 };

 struct IndexStruct
 {
  // struct supporting triangle specification
  // i1 connects to i2, i2 to i3, and i3 to i1

  long i1;
  long i2;
```

```
  long i3;
};
typedef sequence<indexStruct> IndexStructSeq;

struct EdgeTessellationStruct
{
  // edge tessellation

  Object obj_ref;
  // Object reference to underlying topology

  CadUtility::PointStructSeq epts;
  // sequence of pts defining edge tessellation (first struct is
  // the starting pt)

  CadUtility::LongSeq vertex_number;
  // index numbering for all points - relating to epts above

  CadUtility::DoubleSeq t_values;
  //sequence of doubles for t parameters
};
typedef sequence<EdgeTessellationStruct> EdgeTessellationStructSeq;

struct TessellationStruct
{
  // basic tessellation structure, please see Appendix A for indexing

  Object obj_ref;
  // Object reference to underlying topology

  TessType t_type;
  // Application specific type for this tessellation

  CadUtility::PointStructSeq xyz;
  // sequence of 3D pts defining triangles on the Face(length=npts)

  CadUtility::LongSeq face_pts;
  // index numbering for all points - relating to xyz above

  CadUtility::VectorStructSeq normals;
  // sequence of normals at vertices

  CadUtility::UvStructSeq uv;
  // uv parameters associated with the pts (length = npts)

  IndexStructSeq index_list;
  // Index list is a set of 3 values (i1,i2,i3) as pointers into the
  // points/normals/uv values to define a triangle. To allow pt
  // sharing across faces the vertex_number sequence is
  // consistent with face_pts. Please see Appendix A.
```

```
};

struct FaceTessellationStruct
{
  EdgeTessellationStructSeq edges;
  // sequence of edge tessellations

  TessellationStruct face_tessellation;
  // Face - specific tessellation data
};

typedef sequence<FaceTessellationStruct> FaceTessellationStructSeq;

struct ConnectedFaceTessellationStruct
{

  Object obj_ref;
  // Object reference to underlying topology

  long max_vertex_number;
  // total vertices used for tessellation (all faces)

  FaceTessellationStructSeq all_faces;
  // all face tessellations supporting this body
};
```

TessType	An enumeration of possible types of tessellations including WIREFRAME and VISUALIZATION.
TessParametersStruct	A structure of parameters commonly used for the creation of the tessellation especially for various applications. • **max_chord** - is the maximum deviation between triangle center and the surface. • **max-size** - is the maximum allowable length of a segment of a triangular facet for face tessellation and maximum allowable length of a segment for curve and edge tessellation. • **angle** - is the deviation between normals of facets. Angle units are degrees.
IndexStruct	A structure supporting the triangle specification (i1 connects to i2, i2 to i3, and i3 to i1). See Appendix A.

EdgeTessellationStruct	A tessellation of an edge containing: • **obj_ref** is a **CORBA::Object** reference to the underlying edge. • *epts* is the sequence of points defining the edge tessellation. First **PointStruct** is the starting point. **EdgeTessellationStruct** coordinate data within **FaceTessellationStruct** shall be guaranteed to be exactly coincident with corresponding **face_tessellation** coordinates. • **vertex_number** is a sequence of longs that define the numbering (indexing) for ept. • **t_values** is a sequence of doubles providing access to underlying edge **t_parameters**.

TessellationStruct	Basic tessellation data structure used with **FaceTessellationStruct** and **ConnectedFaceTessellationStruct**, also supports tessellation of **Surface** interfaces. • **obj_ref** provides a **CORBA::Object** reference to underlying geometry. • **t_type** is an application specific tag indicating the end-use of the tessellation. • **PointStructSeq** *xyz* is a sequence of 3D pts defining triangles • **CadUtility::LongSeq face_pts** provides index numbering for all points - relating to *xyz* above. • **VectorStructSeq** *normals* is a sequence of normals at vertices. • **CadUtility::UvStructSeq** *uv* is a sequence of uv parameters associated with the *pts*. • **IndexStructSeq index_list** provides index numbering for each triangle. Please see Appendix A, "Tessellation Indexing" for a complete example of indexing.
FaceTessellationStruct	Tessellation data structure for Faces that encapsulates all **CadBrep::Face** and **CadBrep::Edge** tessellation data. • **EdgeTessellationStructSeq edges** provides a sequence of edge tessellations. • **TessellationStruct face_tessellation** provides Face specific tessellation information.
ConnectedFaceTessellationStruct	Tessellation data structure that encapsulates the series of Faces and Edges that comprise the Body. **ConnectedFaceTessellationStructs** shall be "water-tight" (that is, joined across faces). • **Object obj_ref** provides an object reference to the underlying Body. • **long max_vertex_number** provides a count of the total vertices used for tessellation (all faces). • **FaceTessellationStructSeq all_faces** provides all face tessellations supporting this body.

2.4.3 Surface Interface

The **Surface** interface inherits functionality from the **Entity** interface (in the **CadFoundation** module). Most "properties" of this interface can be accessed either individually or through a single "**get_surface_props**" call that returns a **SurfacePropsStruct** data structure.

```
struct SurfacePropsStruct{
  boolean is_bounded_u;
  boolean is_bounded_v;
```

```
    CadUtility::RangeStruct range_u;
    CadUtility::RangeStruct range_v;
    boolean is_closed_u;
    boolean is_closed_v;
  };

struct SurfaceCurvatureStruct
  {
   double min_curvature;
   double max_curvature;
   CadUtility::VectorStruct min_princ_direction;
   CadUtility::VectorStruct max_princ_direction;
  };
  typedef sequence<SurfaceCurvatureStruct> SurfaceCurvatureStructSeq;


  interface Surface : CadFoundation::Entity
  {
   SurfacePropsStruct get_surface_props()
      raises (CadUtility::CadError);
   // recommended access operation for surface properties

   readonly attribute boolean is_bounded_u;
   readonly attribute boolean is_bounded_v;

   CadUtility::RangeStruct range_u()  raises (CadUtility::CadError);
   CadUtility::RangeStruct range_v()  raises (CadUtility::CadError);

   boolean is_closed_u() raises (CadUtility::CadError);
   boolean is_closed_v() raises (CadUtility::CadError);

   double area (inout double accuracy) raises (CadUtility::CadError);
   // Evaluates the area to a specified accuracy.

   CadUtility::PointStructSeq evaluate_points (
      in CadUtility::UvStructSeq uv_parameters,
      in boolean direction_sense_u,in boolean direction_sense_v,
      in long derivative_count,
      out CadUtility::VectorStructSeqSeqSeq derivatives )
       raises (CadUtility::CadError);
   // Evaluates a surface at the specified parameters.

   CadUtility::PointStruct evaluate_point (
      in CadUtility::UvStruct uv_point,
      in boolean direction_sense_u,in boolean direction_sense_v,
      in long derivative_count,
      out CadUtility::VectorStructSeqSeq derivatives)
       raises (CadUtility::CadError);
   // Single point operation (not recommended).

   SurfaceCurvatureStructSeq evaluate_curvatures (
```

```
    in CadUtility::UvStructSeq uv_parameters,

    in boolean direction_sense_u,in boolean direction_sense_v)
     raises (CadUtility::CadError);
// Evaluates the curvature of a surface at the specified
// parameters.

CadUtility::VectorStructSeq evaluate_normals (
    in CadUtility::DoubleSeq u_parameters,
    in CadUtility::DoubleSeq v_parameters,
    in boolean direction_sense_u,in boolean direction_sense_v)
     raises (CadUtility::CadError);
// Evaluates the normal of a surface at the specified parameters.

CadUtility::NurbsSurfaceStruct nurbs_representation (
    inout double tolerance,
    in double low_bound_u, in double high_bound_u,
    in double low_bound_v,in double high_bound_v)
     raises (CadUtility::CadError);
// Returns a NURBS surface that represents this surface.
// If NURBS representation is exact, tolerance will be returned as
// a negative (geometric=-1  and parametric=-2)

CadGeometry::Tessellation tessellate (in TessType t_type,
    inout TessParametersStruct params, out boolean flag)
     raises (CadUtility::CadError);
// Tessellates the surface to the specified TessParameters.
// If Flag is true the TessParameters were changed (original
// values could not be achieved)

boolean project_points_to_nearest (
    in CadUtility::PointStructSeq points,
    out CadUtility::UvStructSeq params,
    out CadUtility::PointStructSeq projected_points,
    out CadUtility::WarningStructSeq warnings)
     raises (CadUtility::CadError);
// Projects each specified point to the nearest point on the
// surface.

boolean project_point_to_nearest (
    in CadUtility::PointStruct point,
    out CadUtility::UvStruct param,
    out CadUtility::PointStruct projected_point,
    out string warning_string)
     raises (CadUtility::CadError);
// Projects a single point - NOT recommended

boolean intersect_ray (in CadUtility::RayStruct i_ray,
    in double tolerance,
    out CadUtility::PointStructSeq intersection_points,
    out CadUtility::UvStructSeq intersection_parameters)
```

```
        raises (CadUtility::CadError);
    // Evaluates the intersections between the specified ray and
    // the surface.

    boolean is_planar (out CadUtility::RayStruct ray)
        raises (CadUtility::CadError);
    // Queries if the surface is planar.
    // If so, the returned ray defines a point and direction for
    // this plane.

};
```

2.4.3.1 Surface Attributes

is_bounded_u	Boolean attribute indicating if the Surface is bounded in u parameter space. True equals bounded.
is_bounded_v	Boolean attribute indicating if the Surface is bounded in v parameter space. True equals bounded.

2.4.3.2 Surface Operations

get_surface_props	Returns a **SurfacePropsStruct** data structure providing range, closure and bounding information in a single operation. This is the recommended approach to obtain basic information about the Surface instance. Property errors will be identified by throwing a CadUtility::CadError.
range_u	Returns a **CadUtility:: RangeStruct** containing minimum and maximum values in u parameter space.
range_v	Returns a **CadUtility:: RangeStruct** containing minimum and maximum values in v parameter space.
is_closed_u	Returns a Boolean that indicates if the Surface is closed in u parameter space.
is_closed_v	Returns a Boolean that indicates if the Surface is closed in v parameter space.
area	Operation determining the area of the Surface to a specified accuracy. Accuracy is an inout parameter that requests a specific level of accuracy. Many CAD systems may not support this accuracy feature and shall return a negative value to indicate no support for this accuracy request. If supported, the input *accuracy* value indicates the requested accuracy and the output value indicates the achieved level. The interpretation of the accuracy parameter is implementation defined.

evaluate_points	Operation that takes a sequence of u, v parameters (*UvStructSeq*) and returns the three-dimensional coordinates through a sequence of *PointStructs*. Also returned is a **CadUtility::VectorStructSeqSeqSeq** containing surface partial derivatives. The **derivative_count** in parameter specifies the maximum partial derivative desired for each *UvStruct*. For each parameter pair, derivatives, Sij, are returned where i corresponds to the number of times with respect to U and j corresponds to the number of times with respect to V. All partial derivative combinations, Sij are returned for 0 <= (i + j) <= *numDeriv*s. If *numDerivs* = 0 (the default), no derivative information is returned.
evaluate_point	Not recommended. This operation is a single point evaluation version of the above operation. This operation is provided as a convenience to developers, but should be used sparingly. For performance reasons, grouping requests through **evaluate_points** is recommended.
evaluate_curvatures	Evaluates the curvature of a surface at the specified u, v parameters and returns the curvature information in a **SurfaceCurvatureStructSeq** This sequence contains minimum and maximum curvature information as well as principle directions. The **direction_sense** arguments indicate the direction this surface should be evaluated from if a discontinuity exists at the specified parameter. If TRUE, the evaluation is from the lower parametric side; if FALSE, from the higher parametric side.
evaluate_normals	Evaluates the normal of a surface at the specified parameters. The direction_sense arguments indicate the direction this surface should be evaluated from if a discontinuity exists at the specified parameter. If TRUE, the evaluation is from the lower parametric side; if FALSE, from the higher parametric side.
nurbs_representation	Returns a **NurbsSurfaceStruct** that represents this surface within the specified tolerance between the specified bounds. The parameterization of the returned surface will likely be different than the original surface. If NURBS representation is exact, tolerance will be returned as a negative (geometric=-1 and parametric=-2).
tessellate	Tessellates the surface to the specified **TessParametersStruct** and **TessType** input structure. If *flag* is true the **TessParametersStruct** were changed (original values could not be achieved). Tessellation indexing is covered in Appendix A.
project_points_to_nearest	Projects a sequence of points (**PointStruct**) to the nearest point on the surface. Returns a sequence of parameter values (**UvStructSeq**) corresponding to the specified points. Also returns a sequence of the projected points. The returned boolean indicates if any warnings have been thrown with the warnings captured in the **WarningSeq**.

project_point_to_nearest	A single point projection operation similar to **project_points_to_nearest**. For performance reasons, this operation is not recommended for applications where groups of points are evaluated.
intersect_ray	Evaluates the intersections between the specified ray (**CadUtility::RayStruct**) and the surface. The *tolerance* defines how close the ray must come to the surface to be considered an intersection. Returns TRUE if any intersections were found, FALSE if not. Any intersections are returned in two sequences: one of 3D points (**CadUtility::PointStructSeq**) and one of corresponding parameters on the surface (**CadUtiltiy::UvStructSeq**).
is_planar	Queries if the surface is planar. If so, the returned ray (**CadUtility::RayStruct**) defines a point and direction for this plane.

2.4.4 Data Structures Supporting Surface

SurfacePropsStruct	This structure provides all of the basic properties of a surface in a single data structure: • **is_bounded_u** = Boolean value indicates whether the surface is bounded in u parameter space. • **is_bounded_v** = Boolean value indicates whether the surface is bounded in v parameter space. • **range_u** = **CadUtility::RangeStruct** that indicates range of u values. • **range_v** = **CadUtility::RangeStruct** that indicates range of v values. • **is_closed_u** = Boolean value indicates whether the surface is closed in u parameter space. • **is_closed_v** = Boolean value indicates whether the surface is closed in v parameter space.
SurfaceCurvatureStruct	This structure provides surface curvature value and direction information: • **min_curvature** = double indicates minimum curvature • **max_curvature** = double indicates maximum curvature • **min_princ_direction** = **CadUtility::VectorStruct** indicating minimum principle direction of the surface curvature. • **max_princ_direction** = **CadUtility::VectorStruct** indicating maximum principle direction of the surface curvature.

2.4.5 Curve Interface

This interface contains basic operations on Curves. Various data structures support this interface and are described in this section.

```
struct CurvePropsStruct
{
  // Properties of a Curve

  boolean is_bounded;
  boolean is_closed;
  CadUtility::RangeStruct range;
};

interface Curve : CadFoundation::Entity
{
  CurvePropsStruct get_curve_props()
      raises (CadUtility::CadError);
  // recommended access operation for curve properties

  readonly attribute boolean is_bounded;
  readonly attribute boolean is_closed;

  double length(inout double accuracy)
     raises (CadUtility::CadError);
  // Calculated length

  CadUtility::RangeStruct range()  raises (CadUtility::CadError);

  CadUtility::PointStructSeq evaluate_points (
     in CadUtility::DoubleSeq parameters,
     in boolean direction_sense,
     in long derivative_count,
     out CadUtility::VectorStructSeqSeq derivatives)
      raises (CadUtility::CadError);
  // Evaluates a curve at the specified parameters.

  CadUtility::DoubleSeq evaluate_curvatures (
     in CadUtility::DoubleSeq parameters,
     in boolean direction_sense) raises (CadUtility::CadError);
  // Evaluates the curvature of a curve at the specified parameters.


  CadUtility::VectorStructSeq evaluate_normals (
     in CadUtility::DoubleSeq parameters,
     in boolean direction_sense) raises (CadUtility::CadError);
  // Evaluates the normal of a curve at the specified parameters.

  boolean intersect_ray (in CadUtility::RayStruct i_ray,
     in double tolerance,
     out CadUtility::PointStructSeq intersection_points,
     out CadUtility::DoubleSeq intersection_parameters)
      raises (CadUtility::CadError);
  // Evaluates the intersections between the specified ray and the
```

```
// curve.

boolean is_planar (out CadUtility::RayStruct ray)
    raises (CadUtility::CadError);
// Queries if the curve is planar.
// If so, the returned ray defines a point and direction for this
// plane.

CadUtility::NurbsCurveStruct nurbs_representation (
   inout double tolerance,
   in double t_min,in double t_max)
   raises (CadUtility::CadError);
// Returns a NURBS curve that represents this curve within the
// specified tolerance. If the representation is exact tolerance
// will be returned as a negative value (geometric=-1  and parametric=-2)

boolean project_points_to_nearest (
   in CadUtility::PointStructSeq points,
   out CadUtility::UvStructSeq params,
   out CadUtility::PointStructSeq projected_points,
   out CadUtility::WarningStructSeq warnings)
    raises (CadUtility::CadError);
// Projects each specified point to the nearest point on the
// curve.

boolean project_point_to_nearest (
   in CadUtility::PointStruct point,
   out CadUtility::UvStruct param,
   out CadUtility::PointStruct projected_point,
   out string warning_string)
    raises (CadUtility::CadError);
// Projects a single point (not recommended for points)

CadGeometry::EdgeTessellationStruct tessellate ( in double tolerance)
    raises (CadUtility::CadError);
// Tessellates the curve to a specified chordal deviation
// tolerance.
};
```

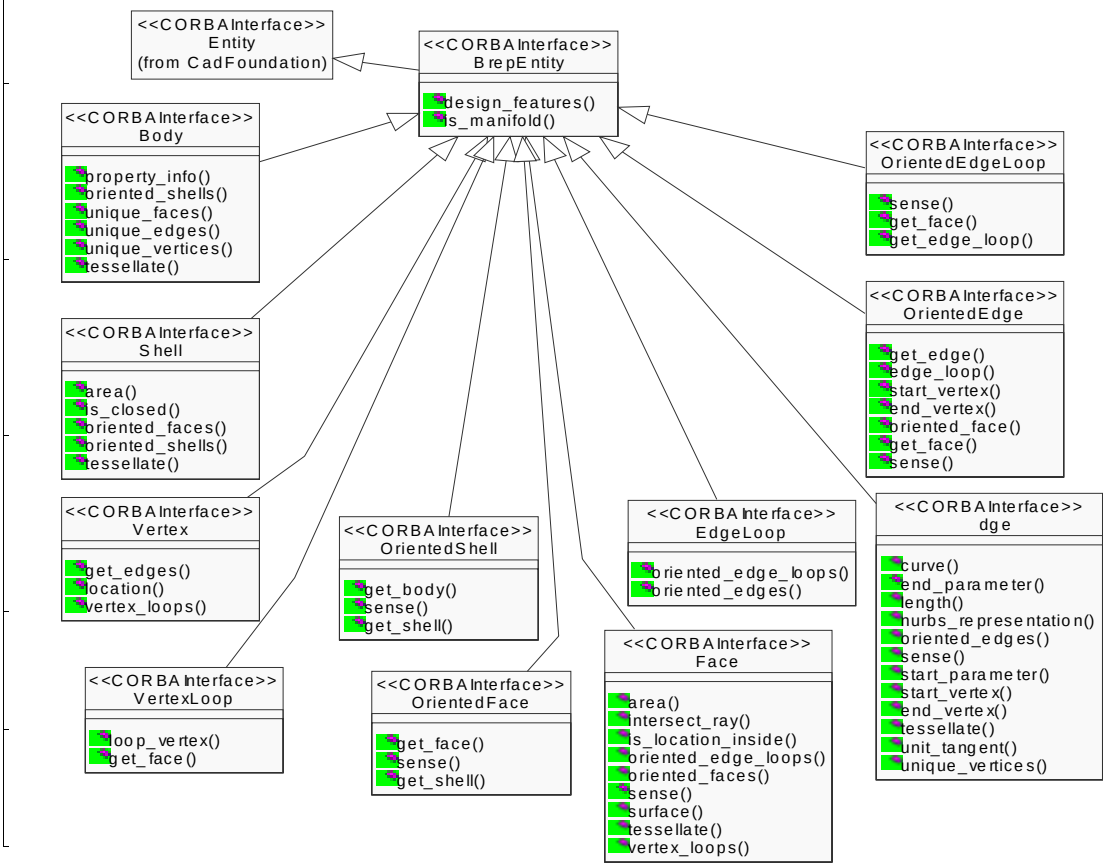
2.4.5.1 Curve Attributes

is_bounded	Readonly boolean attribute that is TRUE if the Curve is bounded.
is_closed	Readonly boolean attribute that is TRUE if the Curve is closed.

2.4.5.2 Curve Operations

get_curve_props	Operation supporting access to most common Curve properties in a single call. Property errors will be identified by throwing a CadUtility::CadError. Returns **CurvePropsStruct** that provides the following information: • boolean **is_bounded** – indicates if the curve is bounded (True = bounded) • boolean **is_closed** – indicates if the curve is closed (True = closed) • **CadUtility::RangeStruct range** provides coordinate limits for the Curve.
range	Operation returning *a* **CadUtility::RangeStruct** providing coordinate limits for the Curve.
length	Operation returning a calculated length of the Curve. Accuracy is an inout parameter that requests a specific level of accuracy. Many CAD systems may not support this accuracy feature and shall return a negative value to indicate no support for this accuracy request. If supported, the input *accuracy* value indicates the requested accuracy and the output value indicates the achieved level. The interpretation of the accuracy parameter is implementation defined.
evaluate_points	Operation that evaluates a curve at the specified parameters. Returns a sequence of points at each corresponding parameter. The **direction_sense** parameter indicates the direction this curve should be evaluated from if a discontinuity exists at any of the corresponding parameters. If TRUE, the evaluation is from the lower parametric side; if FALSE, from the higher parametric side. If the **derivative_count** is greater than 0, the number of requested derivatives will be returned for each parameter.
evaluate_curvatures	Operation that evaluates the curvature of a curve at the specified parameters. The **direction_sense** argument indicates the direction this curve should be evaluated from if a discontinuity exists at the specified parameter. If TRUE, the evaluation is from the lower parametric side; if FALSE, from the higher parametric side.
evaluate_normals	Operation that evaluates the normal of a curve at the specified parameters. The **direction_sense** argument indicates the direction this curve should be evaluated from if a discontinuity exists at the specified parameter. If TRUE, the evaluation is from the lower parametric side; if FALSE, from the higher parametric side.

intersect_ray	Operation that evaluates the intersections between the specified ray and the curve. The *tolerance* defines how close the ray must come to the curve to be considered an intersection. Returns TRUE if any intersections were



2.5 CadBrep Module

The module contains Boundary REPresentations (BREPs). BREPs are solid models such as Bodies, Faces, Edges, and others. These solid models may expose parametric features that allow shape regeneration through interfaces in the **CadFeature** module.

2.5.1 UML Diagram

Figure 2-5 UML Diagram of CadBrep Module Interfaces

2.5.2 BrepEntity Interface

This interface provides common, inherited behavior to the Boundary REPresentations in this module. It contains the design features (or various parameters) that were used to create or derived the solid module.

```
module CadBrep

{

interface BrepEntity : CadFoundation::Entity
 {

   CadFeature::DesignFeatureSeq design_features ()
     raises (CadUtility::CadError);
```

```
// Sequence of the design features directly involved with the
// creation of this entity.

boolean is_manifold() raises (CadUtility::CadError);
};
 typedef sequence<BrepEntity> BrepEntitySeq;
```

design_features	Operation returns the sequence of design features (**CadFeature::DesignFeatureSeq**) that define this BREP. **CadFeature::DesignFeatureSeq** is described in Section 2.6.1, “UML Diagram,” on page 2-64).
is_manifold	Returns a boolean flag indicating if the BrepEntity is manifold. TRUE = manifold. A Vertex always returns TRUE.

2.5.3 Body Interface

The **Body** interface represents a collection of CAD BREP entities defining a closed volume or solid. It inherits design features from the **BrepEntity** interface (Section 2.5.2, “BrepEntity Interface,” on page 2-48).

```
interface Body : BrepEntity
 {
  // A collection of Brep entities defining a closed volume

  PropertyStruct property_info( inout double accuracy)
     raises (CadUtility::CadError);
  // Returns a structure with property info

  OrientedShellSeq oriented_shells () raises (CadUtility::CadError);
  // Returns a sequence of the associated oriented shells. The first
  // oriented shell in the list defines the external or outside
  // boundary of the body.

  FaceSeq unique_faces() raises (CadUtility::CadError);
  // Returns a sequence of the unique faces composing this body

  EdgeSeq unique_edges() raises (CadUtility::CadError);
  // returns a sequence of the unique edges in this body

  VertexSeq unique_vertices() raises (CadUtility::CadError);
  // returns a sequence of unique vertices in this body

  CadGeometry::ConnectedFaceTessellationStruct tessellate (
     in CadGeometry::TessType t_type,
     inout CadGeometry::TessParametersStruct params,
     out boolean flag)
      raises (CadUtility::CadError);
```

```
  // Tessellates the surface to the specified TessParameters
  // If Flag is true the TessParameters were changed
};
```

property_info	For the input *accuracy* (units can be read from the **CadMain::Model** interface for length, Section 2.2.2, “Model Interface,” on page 2-10), this operation returns a **PropertyStruct** (described in Section 2.5.13, “Edge Interface,” on page 2-59). Accuracy is an inout parameter that requests a specific level of accuracy. Many CAD systems may not support this accuracy feature and shall return a negative value to indicate no support for this accuracy request. If supported, the input *accuracy* value indicates the requested accuracy and the output value indicates the achieved level. The interpretation of the accuracy parameter is implementation defined.
oriented_shells	Returns a sequence of the associated oriented shells. The first oriented shell in the sequence defines the external or outside boundary of the body.
tessellate	Tessellates the body to the specified **TessParametersStruct** and **TessType** input values. If *flag* is true, the **TessParametersStruct** were changed (original values could not be achieved). Returns a **CadGeometry::ConnectedFaceTessellationStruct**.

2.5.4 Interface OrientedShell

An oriented shell must always be used by at least one body and therefore is never independent. This interface inherits design features from the **BrepEntity** interface (Section 2.5.2, “BrepEntity Interface,” on page 2-48).

```
interface OrientedShell : BrepEntity
 {
  // An oriented use of a shell.
  // An oriented shell must always be used by at least one body

  Body get_body () raises (CadUtility::CadError);
  // Returns the body that uses this oriented shell.

  boolean sense () raises (CadUtility::CadError);
  // Queries whether the direction of the oriented shell agrees with
  // the direction of the underlying shell.

  Shell get_shell () raises (CadUtility::CadError);
  // Returns the shell associated with this oriented entity.
 };
```

get_body	Returns the **Body** interface that uses this oriented shell.
sense	Queries whether the direction of the oriented shell agrees with the direction of the underlying shell.
get_shell	Returns the Shell interface associated with this OrientedShell entity.

2.5.5 Shell Interface

A **Shell** interface is a collection of oriented faces. An independent, open **Shell** can represent a skin or quilt. A closed **Shell** must always be used by a body. This interface inherits design features from the **BrepEntity** interface (Section 2.5.2, "BrepEntity Interface," on page 2-48).

```
interface Shell : BrepEntity
 {
   // An collection of oriented faces.
   // An independent, open shell can represent a skin or quilt.

   double area( inout double accuracy)
      raises (CadUtility::CadError);
   // calculated shell area

   boolean is_closed() raises (CadUtility::CadError);

   OrientedFaceSeq oriented_faces () raises (CadUtility::CadError);
   // Returns a sequence of the oriented faces in this shell.
   // The ordering of the oriented faces in this sequence has no
   // significance.

   OrientedShellSeq oriented_shells () raises (CadUtility::CadError);
   // Returns a sequence of the oriented shells that use this shell.
   //Returns an empty sequence if this shell is independent.

   CadGeometry::FaceTessellationStructSeq tessellate (
      in CadGeometry::TessType t_type,
      inout CadGeometry::TessParametersStruct params,
      out boolean flag)
       raises (CadUtility::CadError);
   // Tessellates the surface to the specified TessParametersStruct
   // If Flag is true the TessParameters were changed
 };
```

area	For the input *accuracy* (units can be read from the **CadMain::Model** interface for length, Section 2.2.2, “Model Interface,” on page 2-10), this operation returns an area for the **Shell (double)**. Accuracy is an inout parameter that requests a specific level of accuracy. Many CAD systems may not support this accuracy feature and shall return a negative value to indicate no support for this accuracy request. If supported, the input *accuracy* value indicates the requested accuracy and the output value indicates the achieved level. The interpretation of the accuracy parameter is implementation defined.
is_closed	Returns a boolean flag indicating if the **Shell** is closed. TRUE = closed.
oriented_faces	Returns a sequence of the **OrientedFaces** in this **Shell**. The ordering of the **OrientedFaces** in this sequence has no significance.
oriented_shells	Returns a sequence of the **OrientedShell** interfaces that use this **Shell**. Returns an empty sequence if this **Shell** is independent.
tessellate	Tessellates the shell to the specified **TessParametersStruct** and **TessType** input values. If *flag* is true the **TessParametersStruct** were changed (original values could not be achieved). Returns a sequence of **CadGeometry::FaceTessellationStruct**.

2.5.6 Vertex Interface

An independent topological point that represents a point in three-dimensional space. This interface inherits design features from the **BrepEntity** interface (Section 2.5.2, “BrepEntity Interface,” on page 2-48).

```
interface Vertex : BrepEntity
 {
   // A topological point.

   EdgeSeq get_edges () raises (CadUtility::CadError);
   // Returns a sequence of the edges that use this vertex.
   // Returns an empty sequence if this vertex is independent.

   CadUtility::PointStruct location() raises (CadUtility::CadError);
   // Returns the 3D coordinates.

   VertexLoopSeq vertex_loops() raises (CadUtility::CadError);
   // Returns a sequence of the vertex loops that use this vertex.
 };
```

get_edges	Returns a sequence of the edges that use this Vertex. Returns an empty sequence if this Vertex is independent (for example, a 3D point).
location	Returns the 3D coordinates of this Vertex.
vertex_loops	Returns a sequence of the **VertexLoops** (Section 2.5.7, "VertexLoop Interface," on page 2-53) that use this Vertex.

2.5.7 VertexLoop Interface

A topological pole or point location used to define the boundary of a face. Examples include the pole of a sphere or a cone. A vertex loop must always be used by a face and therefore is never independent. This interface inherits design features from the **BrepEntity** interface (Section 2.5.2, "BrepEntity Interface," on page 2-48).

```
interface VertexLoop : BrepEntity
 {
   // A topological pole or point location used to define the
   // boundary of a face. Examples include the pole of a sphere or a
   // cone. A vertex loop must always be used by a face (never
   // independent).

   Vertex loop_vertex () raises (CadUtility::CadError);
   // Returns the vertex that defines the 3D location of this vertex
   // loop.

   Face get_face () raises (CadUtility::CadError);
   // Returns the face that uses this vertex loop.
   // Since vertex loops cannot be independent, this object must be
   // used to construct an edge loop before it is considered valid.
 };
```

loop_vertex	Returns the *Vertex* that defines the 3D location of this **VertexLoop**.
get_face	Returns the *Face* that uses this vertex loop. Since vertex loops cannot be independent, this object must be used to construct an edge loop before it is considered valid.

2.5.8 EdgeLoop Interface

An interface to a region defined by a Edge Loop. This interface inherits design features from the **BrepEntity** interface (Section 2.5.2, "BrepEntity Interface," on page 2-48).

```
interface EdgeLoop : BrepEntity
 {
   OrientedEdgeLoopSeq oriented_edge_loops()
```

```
raises (CadUtility::CadError);
   // oriented edge loops that reference this edge loop

   OrientedEdgeSeq oriented_edges() raises (CadUtility::CadError);
   // oriented edges that compose the edge loop
};
```

oriented_edge_loops	Returns the sequence of **OrientedEdgeLoops** that reference this edge loop.
oriented_edges	Returns the sequence of **OrientedEdges** that reference this edge loop.

2.5.9 OrientedEdgeLoop Interface

An interface to an oriented region defined by a Edge Loop. This interface inherits design features from the **Entity** interface (Section 2.5.2, “BrepEntity Interface,” on page 2-48).

```
interface OrientedEdgeLoop : BrepEntity
 {
   boolean     sense() raises (CadUtility::CadError);
   // true indicates agreement with the underlying edge loop

   Face     get_face() raises (CadUtility::CadError);
   EdgeLoop get_edge_loop() raises (CadUtility::CadError);
 };
```

sense	Boolean return that indicates agreement with the underlying edge loop. TRUE = agreement.
get_face	Returns the **Face** interface associated with this **OrientedEdgeLoop**.
get_edge_loop	Returns the **EdgeLoop** interface associated with this **OrientedEdgeLoop**.

2.5.10 OrientedFace Interface

An oriented use of a face. An oriented face must always be used by at least one shell and therefore is never independent. This interface inherits design features from the **BrepEntity** interface (Section 2.5.2, “BrepEntity Interface,” on page 2-48).

```
interface OrientedFace : BrepEntity
 {
   // An oriented use of a face.

   Face get_face () raises (CadUtility::CadError);
   // Returns the face associated with this oriented entity.
```

```
  boolean sense () raises (CadUtility::CadError);
  // Queries whether the direction of the oriented face agrees with
  // the direction of the underlying face.

  Shell get_shell () raises (CadUtility::CadError);
  // Returns the shell that uses this oriented face.
};
```

get_face	Returns the face associated with this oriented entity.
sense	Queries whether the direction of the oriented face agrees with the direction of the underlying face.
get_shell	Returns the shell that uses this oriented face.

2.5.11 Face Interface

Interface to a region bounded by edges (triangular or quadrilateral). This interface inherits design features from the **Entity** interface (Section 2.5.2, “BrepEntity Interface,” on page 2-48).

```
interface Face : BrepEntity
 {
  readonly attribute CadUtility::RangeStruct range_u;
  readonly attribute CadUtility::RangeStruct range_v;
  // bounds of the active region of the face as defined by the inner
  // and outer loops.

  double area( inout double accuracy)
     raises (CadUtility::CadError);
  // Evaluates face area to a specified accuracy.


  boolean intersect_ray ( in CadUtility::RayStruct ray,
     in double tolerance,
     out CadUtility::PointStructSeq intersection_points,
     out CadUtility::UvStructSeq intersection_parameters)
      raises (CadUtility::CadError);
  // Evaluates the intersections between the specified ray and the
  // face. The tolerance defines how close the ray must come to the
  // face to be considered an intersection. Returns TRUE if any
  // intersections were found, FALSE if not. Any intersections are
  // returned in two sequences: one of 3D points and one of
  // corresponding 2D parameter values on the face's surface.

  Location is_location_inside (in CadUtility::UvStruct location)
     raises (CadUtility::CadError);
 // Queries if a location (defined by uv parameter values) is in the
```

```
//active region of the face as defined by the inner and outer
// loops.

OrientedEdgeLoopSeq oriented_edge_loops ()
    raises (CadUtility::CadError);
// Returns a list of the associated OrientedEdgeLoop entities.
// The first oriented edge loop in the list defines the outside
// boundary of the face.

OrientedFaceSeq oriented_faces () raises (CadUtility::CadError);
// Returns a list of the associated OrientedFace entities.
//Returns an empty list if this face is independent.

boolean sense () raises (CadUtility::CadError);
// Queries whether the direction of the face agrees with the
// parametric (normal) direction of the underlying surface.


CadGeometry::Surface surface () raises (CadUtility::CadError);
// Returns the CadGeometry::Surface entity that defines the shape
// of this face.

CadGeometry::FaceTessellationStruct tessellate (
    in CadGeometry::TessType t_type,
    inout CadGeometry::TessParametersStruct params,
    out boolean flag)
     raises (CadUtility::CadError);
// Tessellates the surface to the specified TessParameters
// If Flag is true the TessParameters were changed

VertexLoopSeq vertex_loops () raises (CadUtility::CadError);
// Returns a sequence of any vertex loops defined on this face.
};
```

2.5.11.1 Face Attributes

range_u	Readonly attribute that returns a **CadUtility::RangeStruct** indicating bounds of the active region of the face for the u parameters.
range_v	Readonly attribute that returns a **CadUtility::RangeStruct** indicating bounds of the active region of the face for the v parameters.

2.5.11.2 Face Operations

area	For the input *accuracy* (units can be read from the **CadMain::Model** interface for length, Section 2.2.2, “Model Interface,” on page 2-10), this operation returns an area for the **Face**. Accuracy is an inout parameter that requests a specific level of accuracy. Many CAD systems may not support this accuracy feature and shall return a negative value to indicate no support for this accuracy request. If supported, the input *accuracy* value indicates the requested accuracy and the output value indicates the achieved level. The interpretation of the accuracy parameter is implementation defined.
intersect_ray	Evaluates the intersections between the specified ray and the face. The *tolerance* defines how close the ray must come to the face to be considered an intersection. Returns TRUE if any intersections were found, FALSE if not. Any intersections are returned in two sequences: one of 3D points (**CadUtility::PointStructSeq intersection_points**) and one of corresponding 2D parameter values (**CadUtility::UvStructSeq intersection_parameters**) on the face’s surface
is_location_inside	Queries if a location (defined by uv parameter values- **CadUtility::UvStruct**) is in the active region of the face as defined by the inner and outer loops. The Location enumeration will indicate whether the parameter values fall: INSIDE, ON_BOUNDARY, or OUTSIDE.
oriented_edge_loops	Returns a list of the associated **OrientedEdgeLoop** entities. The first oriented edge loop in the list defines the outside boundary of the face, if applicable.
oriented_faces	Returns a list of the associated **CadBrep::OrientedFace** entities. Returns an empty list if this face is independent, for example, a trimmed surface.
sense	Queries whether the direction of the face agrees with the parametric (normal) direction of the underlying surface. Critical for determining the “outside” of a face in a body, for example.
surface	Returns the **CadGeometry::Surface** entity that defines the shape of this face in model space.
vertex_loops	Returns a sequence of any vertex loops defined on this face.
tessellate	Tessellates the Face to the specified **TessParametersStruct** and **TessType** input values. If *flag* is true, the **TessParametersStruct** were changed (original values could not be achieved). Returns a **CadGeometry::FaceTessellationStruct**.

2.5.12 OrientedEdge Interface

Interface to an oriented use of an edge. An oriented edge must always be used by at least one edge loop and therefore is never independent.

```
interface OrientedEdge : BrepEntity
 {
   // An oriented use of an edge.

   Edge get_edge () raises (CadUtility::CadError);
   // Returns the edge associated with this oriented entity.

   EdgeLoop edge_loop () raises (CadUtility::CadError);
   // Returns the edge loop that uses this oriented edge.

Vertex start_vertex () raises (CadUtility::CadError);
// Returns the vertex that defines the start of this oriented edge.

Vertex end_vertex () raises (CadUtility::CadError);
   // Returns the vertex that defines the end of this oriented edge.
   // Takes into account any sense differences.

   OrientedFace oriented_face ()
      raises (MultipleFaces, CadUtility::CadError);
   // Returns the oriented face that uses this oriented edge.
   // Returns NULL if the oriented edge is in an independent edge
   // loop or bounds an independent face. Raises an exception
   // if more than one oriented face uses this oriented edge.

   Face get_face() raises (MultipleFaces, CadUtility::CadError);

   boolean sense () raises (CadUtility::CadError);
  // Queries whether the direction of the oriented edge (from start
  //to end vertices) agrees with the direction of the underlying
  // edge.
 };
```

get_edge	Returns the edge associated with this oriented entity.
edge_loop	Returns the edge loop that uses this oriented edge.
start_vertex	Returns the vertex that defines the start of this oriented edge.
end_vertex	Returns the vertex that defines the end of this oriented edge. Takes into account any sense differences.

oriented_face	Returns the oriented face that uses this oriented edge. Returns NULL if the oriented edge is in an independent edge loop or bounds an independent face. Raises an exception (MultipleFaces) if more than one oriented face uses this oriented edge.
get_face	Returns the associated face that uses this oriented edge. Returns NULL if the oriented edge is in an independent edge loop or bounds an independent face. Raises an exception (MultipleFaces) if more than one oriented face uses this oriented edge.
sense	Queries whether the direction of the oriented edge (from start to end vertices) agrees with the direction of the underlying edge.

2.5.13 Edge Interface

An edge entity represents a trimmed portion of a curve. An edge that uses the same vertex for both start and end vertices must be defined as a closed edge on a closed curve starting and ending at this vertex. An independent edge can be used to represent a trimmed curve.

```
interface Edge : BrepEntity
 {
   // A trimmed portion of a curve.An edge that uses the same vertex
   // for both start and end vertices must be defined as a closed
   // edge on a closed curve starting and ending at this vertex. An
   // independent edge can be used to represent a trimmed curve.

   CadGeometry::Curve curve()  raises (CadUtility::CadError);
   // Returns the curve that defines the shape of this edge in model
   // space.

   double end_parameter ()  raises (CadUtility::CadError);
   // Returns the curve parameter corresponding to the end vertex.

   double length ( inout double accuracy )
      raises (CadUtility::CadError);
   // Evaluates the length of the edge to a specified accuracy.


   CadUtility::NurbsCurveStruct nurbs_representation (
      inout double tolerance ) raises (CadUtility::CadError);
   // Returns a NURBS curve that approximates this edge within the
   // specified tolerance.

   OrientedEdgeSeq oriented_edges () raises (CadUtility::CadError);
   // Returns a sequence of the oriented edges that use this edge
   // Returns an empty sequence if this edge is independent.

   boolean sense () raises (CadUtility::CadError);
   // Queries whether the direction of the edge (from start to end
```

```
    // vertices)agrees with the parametric direction of the underlying
    // curve.

    double start_parameter () raises (CadUtility::CadError);
    // Returns the curve parameter corresponding to the start vertex.

    Vertex start_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the start of this edge.

  Vertex end_vertex () raises (CadUtility::CadError);
    // Returns the vertex that defines the end of this edge.

    CadGeometry::EdgeTessellationStruct tessellate (
       in double tolerance)
        raises (CadUtility::CadError);
    // Tessellates the edge to a specified chordal deviation
    // tolerance.

    CadUtility::VectorStruct unit_tangent (in double parameter,
        in boolean sense) raises (CadUtility::CadError);
    // Evaluates the unit tangent vector of the edge at the specified
    // parameter and sense. If the sense is TRUE, the tangent vector
    // is oriented with the edge. If the sense is FALSE, the tangent
    // vector is oriented in the opposite direction.

    VertexSeq unique_vertices () raises (CadUtility::CadError);
    // Returns a sequence of unique vertices used by this edge.
  };
```

curve	Returns the **CadGeometry::Curve** that defines the shape of this edge in model space.
end_parameter	Returns the curve parameter corresponding to the end vertex.
length	Evaluates the length of the edge to a specified *accuracy.* Accuracy is an inout parameter that requests a specific level of accuracy. Many CAD systems may not support this accuracy feature and shall return a negative value to indicate no support for this accuracy request. If supported, the input *accuracy* value indicates the requested accuracy and the output value indicates the achieved level. The interpretation of the accuracy parameter is implementation defined.
nurbs_representation	Returns a NURBS curve (**CadUtility::NurbsCurveStruct**) that approximates this edge within the specified tolerance. The parameterization of the returned curve will likely be different than the original edge. If the representation is exact - *tolerance* will be returned as a negative value (geometrically exact = -1 and parametrically exact = -2).
oriented_edges	Returns a sequence of the oriented edges that use this edge. Returns an empty sequence if this edge is independent.

sense	Queries whether the direction of the edge (from start to end vertices) agrees with the parametric direction of the underlying curve.
start_parameter	Returns the curve parameter corresponding to the starting vertex.
start_vertex	Returns the vertex that defines the start of this edge.
end_vertex	Returns the vertex that defines the end of this edge
tessellate	Tessellates the edge to a specified chordal deviation tolerance. Returns an **CADGeometry::EdgeTessellationStruct**.
unit_tangent	Evaluates the unit tangent vector of the edge at the specified parameter and sense. If the sense is TRUE, the tangent vector is oriented with the edge. If the sense is FALSE, the tangent vector is oriented in the opposite direction. Returns the unit tangent vector in a **CadUtility::VectorStruct**.
unique_vertices	Returns a sequence of unique vertices used by this edge.

2.5.14 Structures and Exceptions

2.5.14.1 PropertyStruct

A **PropertyStruct** is used to pass basic information on CAD system BREP entities. Properties are clearly identifiable from the IDL naming with units set or determined at the **CadMain::Model** interface. Error information from the native CAD system is provided on derived properties.

```
struct PropertyStruct
 {
   double  surface_area;
   double   volume;
   double   mass;
   double   solid_density;
   // Solid density is provide as a reference value

   CadUtility::VectorStruct  centroid;
   CadUtility::VectorStruct  intertial_moments;
   CadUtility::VectorStruct  inertial_products;
   CadUtility::VectorStruct  principle_x_axis;
   CadUtility::VectorStruct  principle_y_axis;
   CadUtility::VectorStruct  principle_z_axis;
   CadUtility::VectorStruct  gyration_radii;
   // Items relative to the frame

   CadUtility::VectorStruct inertial_moments_centroidal;
   CadUtility::VectorStruct inertial_products_centroidal;
   CadUtility::VectorStruct principle_moments_centroidal;
   CadUtility::VectorStruct gyration_radii_centroidal;
   // Items relative to the centroid
```

```
    double surface_area_error;
    double volume_error;
    double mass_error;
     // Error Values

};
```

surface_area	Value indicates surface area of geometric body.
volume	Volume of geometric body.
mass	Mass of geometric body.
solid_density	Solid density of body which is provided for reference.
centroid	Centroid or center-of-mass for geometric body.
inertial_moments	Moments of inertia relative to the frame.
inertial_products	Inertial products relative to the frame.
principle_x_axis	Principal x axis relative to the frame.
principle_y_axis	Principal y axis relative to the frame
principle_z_axis	Principal zaxis relative to the frame
gyration_radii	Radius of gyration along each axis
inertial_moments_centroidal	Moments of inertia relative to the centroid.
inertial_products_centroidal	Inertial products relative to the centroid.
principle_moments_centroidal	Moments of inertia along the principle axes through the center of gravity of a geometric body.
gyration_radii_centroidal	Radius of gyration through the centroid.
surface_area_error	Error associated with surface area calculation.
volume_error	Error associated with volume calculation.
mass_error	Error associated with mass calculation.

2.5.14.2 MultipleFaces Exception

This exception supports the **OrientedEdge** interface by identifying instances referencing more than one **Face** interface.

```
exception MultipleFaces
{
  FaceSeq multiples;
};
```

2.6 CadFeature Module

The **CadFeature** Module provides interfaces (through inheritance with **CadFoundation::Entity**) that enable modification of native CAD entities. These interfaces enable suppression of various design features and a parameter set of expressions or values that define the geometry of the CAD entities. For example, a solid model of a box might have an associated parameter set that uniquely defines the width, length and height of the box. A client application might alter any of these parameters to regenerate the geometry, but would be unable to specify new parameters.

2.6.1 UML Diagram

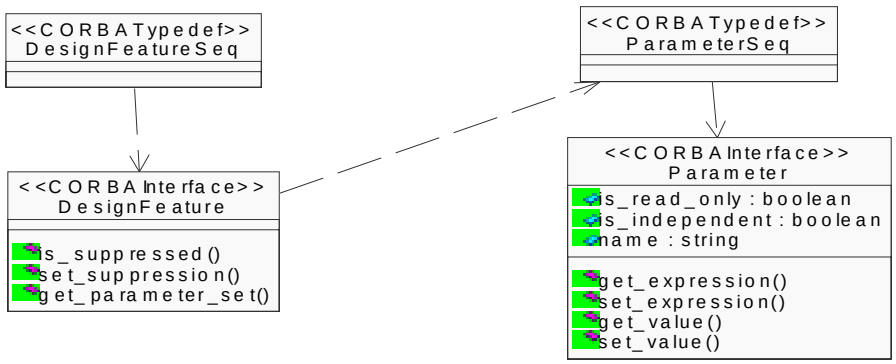


Figure 2-6 UML Diagram of CadFeature Module

2.6.2 DesignFeature Interface

A distinct step or node in the parametric definition of a model. It drives the creation of a set of geometric entities in the fully evaluated form of the model.

```
module CadFeature
{

interface DesignFeature : CadFoundation::Entity
 {
   // A distinct step or node in the parametric definition of a model.
   // It drives the creation of a set of Brep entities in the fully-
   // evaluated form of the model.

   boolean       is_suppressed() raises (CadUtility::CadError);
   void          set_suppression() raises (CadUtility::CadError);
   ParameterSeq get_parameter_set() raises (CadUtility::CadError);
 };
```

is_suppressed	Operation that indicates whether a design feature is suppressed. TRUE = suppressed; FALSE = unsuppressed.
set_suppression	Operation that toggles the suppression of a design feature. If the feature is suppressed, this operation unsuppresses the feature.
get_parameter_set	Returns a sequence of **Parameter** interfaces (Section 2.6.3, “Parameter Interface,” on page 2-65) that define this design feature. If the implementation does not support the Parametrics compliance point; that is, does not allow parametric regeneration of CAD entities, **get_parameter_set** shall return a zero-length sequence.

2.6.3 Parameter Interface

An interface to capture the parametric features of native CAD entities.

```
interface Parameter
 {
  // data structures that capture the (parametric) features

  readonly attribute boolean        is_read_only;
  readonly attribute boolean        is_independent;
  readonly attribute string         name;

  string   get_expression();
  void set_expression(in string e_value)
     raises (CadUtility::CadError);
  // operations to allow an expression that may drive geometry

  CadUtility::EntityAttrib  get_value()
     raises (CadUtility::CadError);
  void set_value(in CadUtility::EntityAttrib value)
     raises (CadUtility::CadError);
  // operations providing access to parameter value
 };
```

2.6.3.1 Parameter Attributes

is_read_only	Readonly attribute that indicates if the parameter is read only; that is, cannot be modified. TRUE = Read Only.
is_independent	Readonly attribute that indicates if this parameter depends on other parameters. TRUE = independent.
name	Readonly string providing a user-understandable name for this parameter. This string is associated with the **CadUtility::EntityAttrib** value (determined in the **get_value** or **set_value** operations, Section 2.6.3.2, “Parameter Operations,” on page 2-65).

2.6.3.2 Parameter Operations

get_expression	Returns a string that provides an expression that defines the geometry of the native CAD entity.

set_expression	Takes as a string an input argument that provides a geometry modifiying expression.
get_value	Operation that returns a value (type defined by **CadUtility::EntityAttrib**). The name attribute provides a string to define this value (“length,” “width,” or “height” in the above example).
set_value	Set the value (type defined by **CadUtility::EntityAttrib**) of this parameter.

2.7 CadUtility Module

The **CadUtilty** module is a collection of general-purpose data definitions in the form of *typedef* and *struct* used throughout the specification. No interface definitions are in this module.

The **CadUtility** starts by defining some very basic geometric data structures:

```
module CadUtility
{
// basic geometric structures

 struct PointStruct
 {
  // three dimensional location

  double x;
  double y;
  double z;
 };

 typedef sequence<PointStruct>     PointStructSeq;
 typedef sequence<PointStructSeq>  PointStructSeqSeq;

 struct BoundingBox
 {
  PointStruct   point_min;
  PointStruct   point_max;
 };

  struct VectorStruct
 {
  // Direction in 3D

  double i;
  double j;
  double k;
 };

 typedef sequence<VectorStruct> VectorStructSeq;
```

```
typedef sequence<VectorStructSeq> VectorStructSeqSeq;
typedef sequence<VectorStructSeqSeq> VectorStructSeqSeqSeq;

struct TransformationStruct
{
  PointStruct   offset;
  VectorStruct    i_ref_dir;
  VectorStruct    k_dir;
};

struct RayStruct
{
  PointStruct     origin;
  VectorStruct    direction;
};
typedef sequence<RayStruct> RayStructSeq;

struct UvStruct
{
  double u;
  double v;
};

typedef sequence<UvStruct> UvStructSeq;
```

PointStruct	The **PointStruct** structure is used as the basic three-dimensional coordinate data structure with locations along the three coordinate directions (x,y,z). The units associated with these locations are defined in the **CadUtility:: LengthUnit** Enumeration.
BoundingBox	The **BoundingBox** structure refers to the geometric size limits of a particular entity (or group of entities). It is frequently used to graphically limit the viewing window of a CAD system entity.
VectorStruct	This structure describes a basic vector, from the origin to the specified distances along each axis (in three-dimensions).
TransformationStruct	**TransformationStruct** locates any entity by providing a three-dimensional transformation. The unit vectors in the **i, j**, and **k** direction can be determined from: • **i** would be determined as unit vector corresponding to: **i_ref_dir – ((i_ref_dir) · k_dir)) k_dir** • **j** would then be unit vector of: **k_dir x i** • **k** would be unit vector of **k_dir**. The resulting **i**, **j**, and **k** are unit vectors that are mutually perpendicular and form a right hand coordinate system.

```
typedef sequence<boolean> BooleanSeq;
 typedef sequence<long>    LongSeq;
 typedef sequence<double>  DoubleSeq;
 typedef sequence<string>  StringSeq;
 typedef sequence<DoubleSeq> DoubleSeqSeq;
 typedef sequence<any>     AnySeq;
 typedef sequence<CORBA::TypeCode> TypeCodeSeq;


 enum MassUnit
 {
  // mass unit options

  POUNDS,
  GRAMS,
  KILOGRAMS,
  UNKNOWN_MASS
 };

 enum LengthUnit
 {
  // length unit options

  INCH,
  FEET,
  M,
  CM,
  MM,
  UNKNOWN_LENGTH
 };

 // enum + union supporting parameters

 enum AttribTypes
 {
  LONG_TYPE,
  DOUBLE_TYPE,
  STRING_TYPE,
  BOOLEAN_TYPE
 };

 union EntityAttrib switch(AttribTypes)
 {
  case LONG_TYPE: long     l_value;
  case DOUBLE_TYPE: double d_value;
  case STRING_TYPE: string s_value;
  case BOOLEAN_TYPE: boolean b_value;
    };

 struct ColorStruct
 {
```

```
    // basic color information in RGB form
    // Valid values range from 0.0 to 1.0

    double red;
    double green;
    double blue;
  };
  typedef sequence<ColorStruct> ColorStructSeq;

  struct PresentationStruct
  {
    // CAD system presentation data
    // Unsupported features will return a negative value
    // Valid values range from 0.0 to 1.0

ColorStruct object_color;
ColorStruct   specular_color; // light source color
    double diffuse_factor;
    double specular_factor;
    double ambient_factor;
    double roughness;
    double transparency; //  0. is opaque and 1. is transparent
  };

  struct RangeStruct
  {
    // basic range information

    double high;
    double low;
  };

  struct WarningStruct
  {
    // struct for warning messages

    long   index;
    string message;
  };
  typedef sequence<WarningStruct>WarningStructSeq;

// NURBS data structures

  struct NurbsCurveStruct
  {
    boolean is_rational;
    // rational or polynomial?

    CadUtility::DoubleSeq knots;
    // A sequence of knot values.
```

```
  CadUtility::DoubleSeq weights;
  // A sequence of weight values.

  CadUtility::PointStructSeq control_points;
  // A sequence of control points in 3D

  CadUtility::LongSeq multiplicity;
  long       degree;
};
typedef sequence<NurbsCurveStruct> NurbsCurveStructSeq;

struct NurbsSurfaceStruct
{
  boolean is_rational;
  // rational or polynomial?

  CadUtility::DoubleSeq knots_u;
  CadUtility::DoubleSeq knots_v;
  // Sequence of knot values.

  CadUtility::DoubleSeqSeq weights;
  // A sequence of weight values.

  CadUtility::PointStructSeqSeq control_points;
  // A sequence of control points.
  // Each point is a sequence of a sequence of 3D points.

  CadUtility::LongSeq multiplicity_u;
  CadUtility::LongSeq multiplicity_v;
  long   degree_u;
  long   degree_v;
};
typedef sequence<NurbsSurfaceStruct> NurbsSurfaceStructSeq;
```

Sequences of Basic Types	The **CadUtility** Module provides a series of sequences of basic CORBA types: boolean, long, double, string, any, and **CORBA::TypeCode**.
Unit Types	A series of data types used to establish mass and length units. These data structures are used by the **CadMain::Model** interface (Section 2.2.2, “Model Interface,” on page 2-10) to set or get unit properties. • **MassUnit** – an enumeration of available mass units. The UNKNOWN_MASS value can be used for normalized or non-dimensional mass units. • **LengthUnit** – an enumeration of available length units. The UNKNOWN_LENGTH value can be used for normalized or non-dimensional length units.

Enum AttribTypes and Union EntityAttrib	Data structures used to establish permitted types for the **CadFeature::Parameter** interface. Through the **EntityAttrib** switch the following basic types are supported: long, double, string, and boolean.
ColorStruct	Data struct capturing RGB values defining the color of CAD System entities. Valid values range from 0.0 to 1.0. Negative values indicate no support for color.
PresentationStruct	CAD systems support a variety of presentation information that is captured in a single data struct. Valid values range from 0.0 to 1.0 and unsupported variables will return a negative value. • *object_color* - The color of the CAD entity. • *specular_color* - Determines the color of the highlight which is a key to visualization parameter to differentiate between metals and plastics. For plastic and painted materials the specular color is generally white or the color of the light source. For metallic objects the highlight color is the color of the material. This is a RGB color value in the range of 0.0 to 1.0. (Light reflection in plastic is a surface function, but in metals the photon is absorbed and then re-emitted which is why the incoming color changes the material color.) • *diffuse_factor* - Diffuse light reflection from an object where direct incoming light is generally scattered in multiple directions. This value ranges from 0.0 to 1.0 where low values indicate smoother surfaces and high values are very coarse surfaces (tennis ball). • *specular_factor* - Determines the intensity of the highlight on the surface. This value ranges from 0.0 to 1.0 and a rule of thumb would be specular_factor = 1.0 - diffuse_factor, where smoother surfaces generally have brighter highlights. • *ambient_factor* - Determines a level of overall global lighting. Ambient light uniformly increases the brightness of all objects in a scene. This value is normally in the range of 0.1 to 0.2. Higher values cause the rendering to look “washed out.” • *roughness* - Determines how much the highlight spreads out Low values in the 0.0 to 1.0 range yield very small and intense highlights. Higher values cause the highlight to spread over more of the surface. Roughness is used to model the “microfacet roughness” of the surface and diffuse and specular values model more of the visual level of roughness. • *transparency* – Light transmission of the body. A low value of 0 is opaque and 1 is transparent.

RangeStruct	Basic high / low range information, usually used to establish limits for geometric bounds and / or parameters.
WarningStruct	Data structure used to capture warning messages in “grouped” queries, for example, the point projection operations on the **CadGeometry::Surface** interface.
NURBS Data Structures	Non-Uniform Rational B-Splines (NURBS) are commonly used to represent (or approximate) curves and surfaces. In this standard, all NURBS data shall be presented as being clamped; that is, knot vectors that have orderful multiplicity at start and/or end knots. If native CAD data is not clamped, the precise conversion to clamped form shall be performed. Periodic NURBS data structures; that is, “unclamped” are not supported. • **NurbsCurveStruct** – A NURBS curve defined by the following properties: •-*boolean rational* determines whether the NURBS is rational or polynomial. TRUE = rational. •-*CadUtility::DoubleSeq knots* provides a sequence of distinct NURBS knot values. •-*CadUtility::DoubleSeq weights* provides a sequence of weight values. When rational is FALSE weights will be ignored. •-*PointStructSeq control_points* provides a sequence of control points. •-*CadUtility::LongSeq multiplicity* - the number of times each distinct knot is repeated. •-*long degree* specifies the degree of the NURBS representation. • **NurbsSurfaceStruct** - A NURBS surface that provides a description along the surface (e.g., in u and v). Similar to the **NurbsCurveStruct**, but with data sequences in both u and v parameters.

2.8 CadGeometryExtens Module

Interfaces and data structures in **CadGeometryExtens** supplement the interfaces within the **CadGeometry** Module. Three interfaces related to Points are in the **CadGeometryExtens** Module and two new modules are structured under **CadGeometryExtens**. These new modules are: **CadCurve** and **CadSurface**. The interfaces within each module inherit functionality through the either the **CadGeometry:Curve** or **CadGeometry:Surface** interfaces.

These interfaces and data structures provide additional details on various types of geometry entities. Most of the functionality is provided through inheritance, but each interface has an operation (frequently **xx_info()**) that returns a data struct with the basic defining information for the geometric entity. An example is the **CadGeometryExtens::CadCurve::Circle** interface. This interface has one

operation: **circle_info()** that returns a **CircleStruct** containing a **CORBA::double** indicating the radius and a **CadUtility::TransformationStruct** indicating location. These two pieces of data clearly define the circle.

2.8.1 CadGeometryExtens Module Interfaces and Data Structures

Three new interfaces are introduced that provide functionality for points.

2.8.1.1 UML Diagram

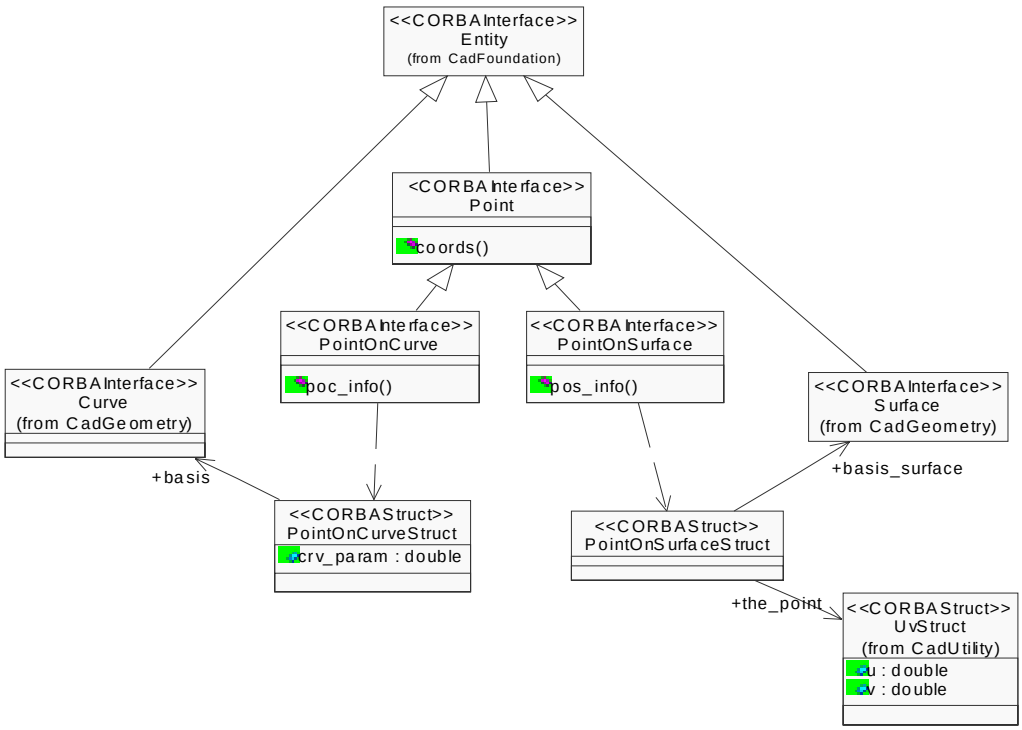


Figure 2-7 UML Diagram of CadGeometryExtens Interfaces and Data Structures

2.8.1.2 Point, PointOnCurve, and PointOnSurface Interfaces

These interfaces inherit most of their functionality through the **CadFoundation::Entity** interface. The **Point** interface supports one operation that locates the **Point** and sub-typed interfaces define the type of **Point** (located on a surface or a curve).

```
module CadGeometry{

  interface Point : CadFoundation :: Entity{
```

```
  CadUtility::PointStruct coords()raises (CadUtility::CadError);
};

struct PointOnSurfaceStruct{
  CadGeometry::Surface basis_surface;
  CadUtility::UvStruct the_point;
};

interface PointOnSurface : Point{
  PointOnSurfaceStruct pos_info()raises (CadUtility::CadError);
};

struct PointOnCurveStruct{
  CadGeometry::Curve basis;
  double crv_param;
};

interface PointOnCurve : Point {
    PointOnCurveStruct poc_info()raises (CadUtility::CadError);
};
```

2.8.2 CadGeometryExtens::CadSurface Module

The interfaces within the **CadGeometryExtens::CadSurface** module inherit most of their functionality from the **CadGeometryExtens::Surface** interface. They provide a range of geometric entities that are different types of surfaces.

2.8.2.1 UML Diagram

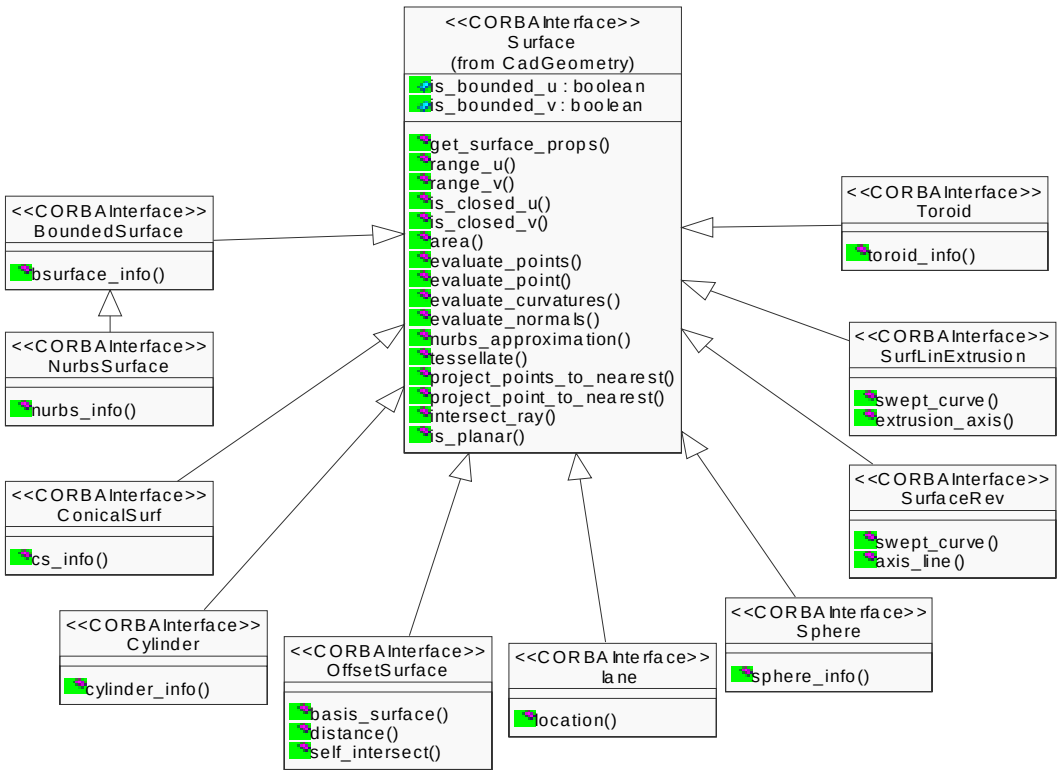


Figure 2-8 UML Diagram of CadGeometryExtens::CadSurface Interfaces

2.8.2.2 CadGeometryExtens::CadSurface Interfaces and Data Structures

```
module CadSurface{

 interface BoundedSurface;
 enum TransitionCode{
      DISCONTINUOUS, CONTINUOUS, CONT_SAME_GRAD,
      CONT_SAME_GRAD_SAME_CURVATURE};

 struct SurfacePatchStruct{
     BoundedSurface parent_surface;
     TransitionCode u_transition;
     TransitionCode v_transition;
     boolean u_sense;
     boolean v_sense;
 };
```

```
interface BoundedSurface : CadGeometry::Surface {
    SurfacePatchStruct bsurface_info()
     raises (CadUtility::CadError);
};

struct ConicalSurfStruct{
    CadUtility::TransformationStruct location;
    double radius;
    double semi_angle;
};

interface ConicalSurf : CadGeometry::Surface{
    ConicalSurfStruct cs_info()raises (CadUtility::CadError);
};
struct CylinderStruct{
    CadUtility::TransformationStruct location;
    double radius;
};

interface Cylinder: CadGeometry::Surface {
    CylinderStruct cylinder_info()raises (CadUtility::CadError);
};


struct HyperbolaStruct{
    CadUtility::TransformationStruct location;
    double semi_axis;
    double semi_imag_axis;
};

interface NurbsSurface : BoundedSurface{
    CadUtility::NurbsSurfaceStruct nurbs_info()
     raises (CadUtility::CadError);
};

interface OffsetSurface : CadGeometry::Surface {
    CadGeometry::Surface basis_surface()
     raises (CadUtility::CadError);
    double distance()raises (CadUtility::CadError);
    boolean self_intersect()raises (CadUtility::CadError);
};

interface SurfaceRev : CadGeometry::Surface {
    CadGeometry::Curve swept_curve()raises (CadUtility::CadError);
    CadUtility::RayStruct axis_line()
     raises (CadUtility::CadError);
};

struct SphereStruct{
    double radius;
    CadUtility::TransformationStruct location;
```

```
};

interface Sphere: CadGeometry:: Surface{
   SphereStruct sphere_info()raises (CadUtility::CadError);
};

struct ToroidStruct{
   CadUtility::TransformationStruct location;
   double major_radius;
   double minor_radius;
};

interface Toroid : CadGeometry::Surface{
   ToroidStruct toroid_info()raises (CadUtility::CadError);
};

interface Plane : CadGeometry::Surface{
   CadUtility::TransformationStruct location()
    raises (CadUtility::CadError);
};

interface SurfLinExtrusion : CadGeometry:: Surface{
   CadGeometry::Curve swept_curve()raises (CadUtility::CadError);
   CadUtility::VectorStruct extrusion_axis()
    raises (CadUtility::CadError);
};
};
```

The various types of surfaces represented by these interfaces and data structures follow the convention of providing one or two operations that define the surface type. An enumeration (**TransitionCode**) supports different types of transitions between surfaces for the **BoundedSurface** interface. These transitions include: DISCONTINUOUS, CONTINUOUS, CONT_SAME_GRAD, CONT_SAME_GRAD_SAME_CURVATURE.

2.8.3 CadGeometryExtens::CadCurve Module

The interfaces within the **CadGeometryExtens::CadCurve** module inherit most of their functionality from the **CadGeometryExtens::Curve** interface. They provide a range of geometric entities that are different types of curves. The various types of curve represented by these interfaces and data structures follow the convention of providing one or two operations that define the curve type.

2.8.3.1 UML Diagram

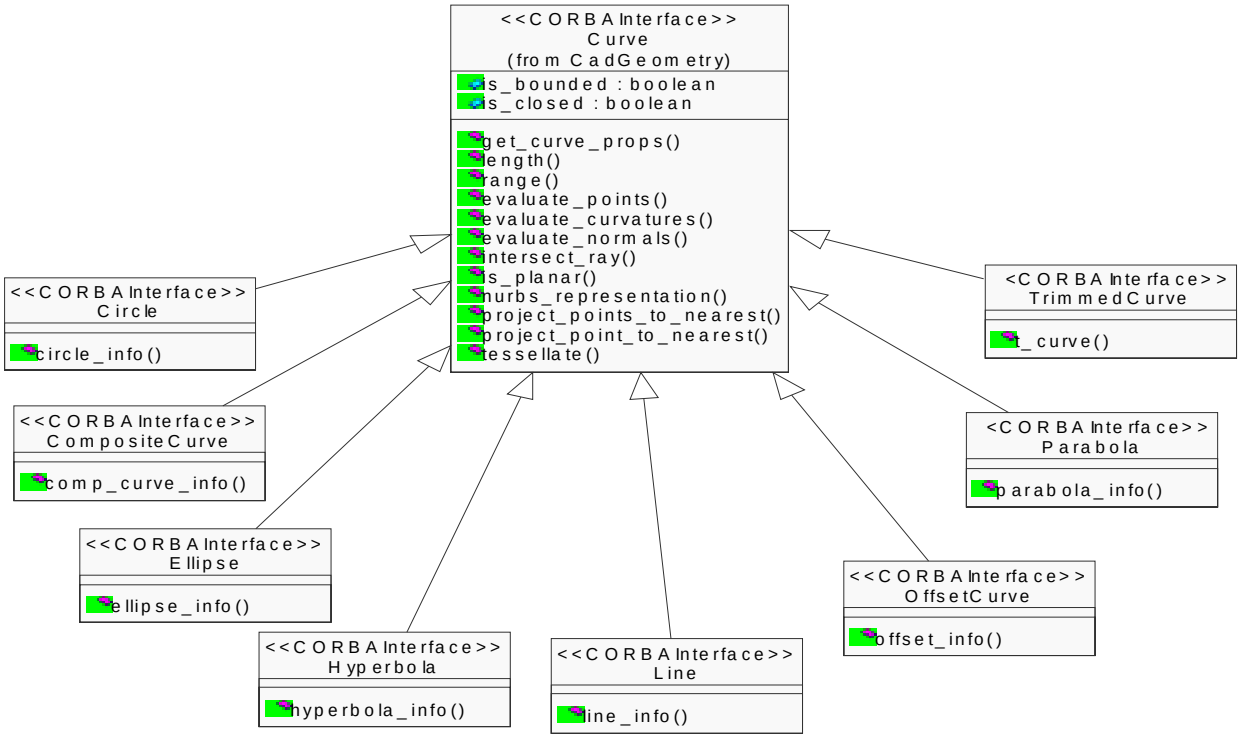


Figure 2-9 UML Diagram of CadGeometryExtens::CadCurve Interfaces

2.8.3.2 CadGeometryExtens::CadCurve Interfaces and Data Structures

```
module CadCurve{
  struct CircleStruct{
    CadUtility::TransformationStruct location;
    double radius;
  };

  interface Circle : CadGeometry::Curve{
    CircleStruct circle_info()raises (CadUtility::CadError);
  };

  struct CompositeCurveStruct{
    long count;
    CadGeometry::CurveSeq segments;
    CadUtility::BooleanSeq senses;
  };

  interface CompositeCurve: CadGeometry::Curve{
```

```
  CompositeCurveStruct comp_curve_info()
    raises (CadUtility::CadError);
};

struct ParabolaStruct{
  CadUtility::TransformationStruct location;
  double focal_distance;
};

interface Parabola : CadGeometry::Curve {
  ParabolaStruct parabola_info()raises (CadUtility::CadError);
};

interface Hyperbola : CadGeometry::Curve {
  CadGeometryExtens::CadSurface::HyperbolaStruct hyperbola_info()
     raises (CadUtility::CadError);
};

struct LineStruct{
  CadUtility::PointStruct the_point;
  CadUtility::VectorStruct direction;
};

interface Line : CadGeometry::Curve {
  LineStruct line_info()raises (CadUtility::CadError);
};

struct OffsetCurveStruct{
  double distance;
  CadUtility::VectorStruct ref_direction;
  boolean self_intersect;
};

interface OffsetCurve : CadGeometry::Curve{
  OffsetCurveStruct offset_info()raises (CadUtility::CadError);
};

struct EllipseStruct{
  CadUtility::TransformationStruct location;
  double semi_axis_1;
  double semi_axis_2;
};

interface Ellipse : CadGeometry::Curve {
  EllipseStruct ellipse_info()raises (CadUtility::CadError);
};

struct PolyLineStruct{
  CadUtility::PointStructSeq the_points;
};
```

```
  struct TrimmedCurveStruct{
    CadGeometryExtens::PointOnCurveStruct trim_1;
    CadGeometryExtens::PointOnCurveStruct trim_2;
    boolean sense_agreement;
  };

  interface TrimmedCurve : CadGeometry::Curve{
    TrimmedCurveStruct t_curve()raises (CadUtility::CadError);
  };

  struct SurfaceCurveStruct{
    CadGeometry::Curve curve_3d;
    long basis_count;
    CadGeometry::SurfaceSeq basis_surfaces;
  };
};
```

Optional vs. Mandatory Interfaces 3

Contents

This chapter contains the following sections.

3.1 Summary of optional versus mandatory interfaces

Optional interfaces are identified for various compliance points as discussed in Section 1.2, “Compliance Discussion,” on page 1-2.

Information on supported compliance points shall be provided by the **OptionsStruct** struct that is contained in the **NativeCadAttributeStruct** struct (Section 2.1.2.1, “CadServer Attributes,” on page 2-3). All other interfaces, data structures, and operations in this standard are mandatory.

3.2 Compatibility With PDM Enablers

A joint submission team is developing a response to the PDM Enablers V2.0 RFP (OMG Document mfg/2000-01-02). The PDM Enablers V2.0 submission proposes an interface that is compatible with this CAD Services specification. This section describes the proposed approach toward integration. While the integration with PDM Enablers V2.0 involves no changes to the CAD Services IDL, significant implementation extensions will be required. It should be noted that this section is provided as a convenience to the

implementers; however, this information may not be current. Implementers are encouraged to review latest document available from the OMG web site (http://www.omg.org).

The PDM Enablers V2.0 submission proposes an **ActiveModel** interface that permits engineering applications to access the CAD native data stored in the PDM system. The **ActiveModel** object represents the PDM system knowledge of a specific instantiation of a product model in an engineering software system. It represents any persistent constructs (files, directories, and other linkages) that the PDM system creates in order to instantiate the product model. When the **ActiveModel** object is destroyed, the external linkage constructs created by the PDM for that instantiation may be deleted. In many cases, the implementation of such an object may be a private interface between the PDM system and the engineering tool, or a parameterized script for the activation of the engineering tool, etc.

An **ActiveModel** is created from an **ActiveModelFactory** interface and supports various types of access through an ActivationMode enumeration. The various modes of activation include: ACTIVE_READONLY, ACTIVE_CHECKOUT, and ACTIVE_DETAILED. When the mode is ACTIVE_CHECKOUT, it is the intent of the client to be able to invoke a subsequent **checkin()** operation on the **ActiveModel** after modifying some elements of the representation. When the mode is ACTIVE_READONLY, the client will not invoke **checkin()**. The value ACTIVE_DETAILED means that other (implementation-defined) parameters affect the interpretation of the mode parameter.

A CAD Services client would access native CAD system file(s) or open the **CadMain::Model** interface through a two-step process. First, the client would log-in to a PDM Enablers V2.0 server following the process for all PDM clients. It would then instantiate (or connect to) an **ActiveRepresentationFactory** and create an **ActiveModel** from either a **create_from_files**, **create_from_context**, or **find** operation. Second, the **ActiveRepresentation** provides two possible mechanisms to open the CAD model. The **eng_sys_reference** operation returns a string that can be passed into the **open_model** operation of the **CadConnection::CadSystem** interface. The second mechanism is to use the **get_external_object** operation that returns an object reference that can be narrowed to a **CadMain::Model** interface.

Using two mechanisms to open the **CadMain::Model** interface is provided, due to the expected delay in opening the **CadMain::Model** interface. A **get_external_object** operation may involve a large computational load to open a complex CAD model. For these uses, the **eng_sys_reference** is recommended. This operation should return a string to the client in a reasonably short time-period. The CAD Services client can then use this string to open the CAD model, without the complication of possible delays in multiple systems.

CAD Services clients of the PDM Enablers V2.0 should invoke *destroy* on their **ActiveModel** reference at the end of the session to allow the PDM system to clean-up any supporting system resources.

An example of the use of PDM Enablers V 2.0 and CAD Services v 1.0 for an ACTIVE_CHECKOUT mode is displayed in figure 3-1. This sequence diagram illustrates a typical client interaction where requests are made to create an ActiveModel

from a context using relevant project descriptions. The ActiveModel is then accessed for a string (“XYZ123”) that allows the CadSystem to open a Model. Certain modifications of the Model are performed and this Model is then saved and checked-in to the PdmSystem.

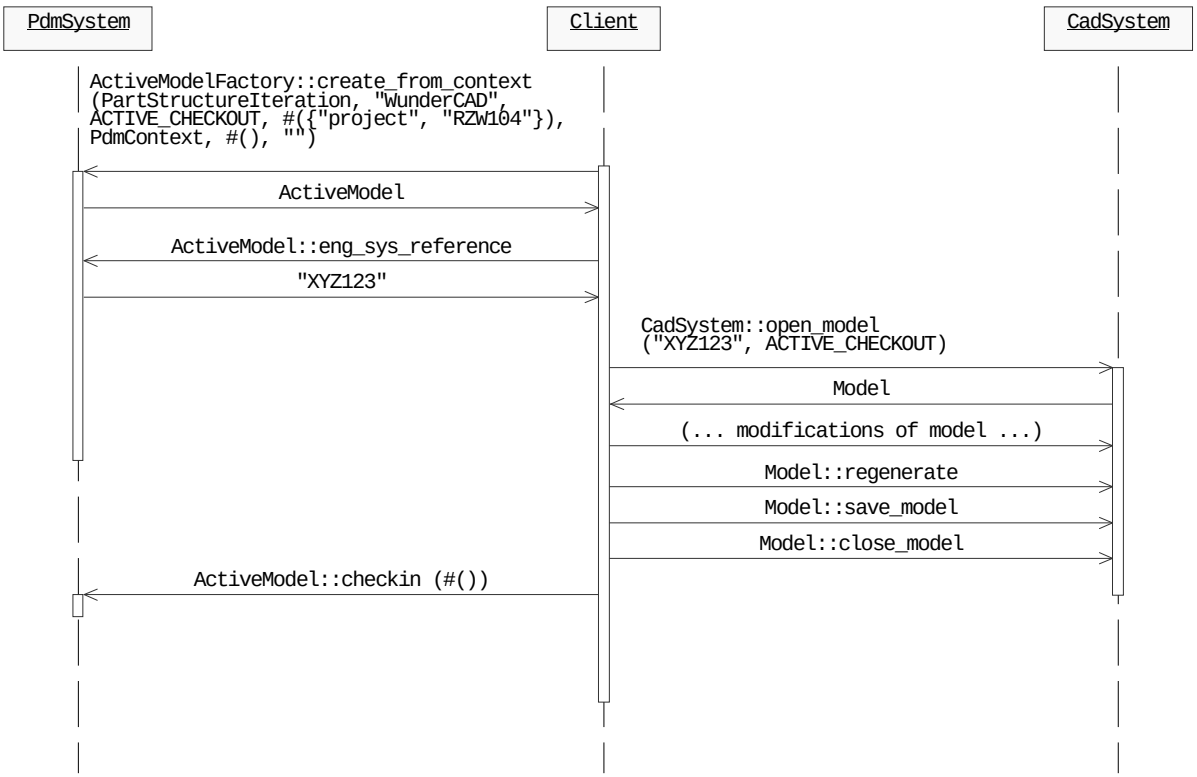


Figure 3-1 Sequence diagram illustrating ACTIVE_CHECKOUT operations between a Client and both the PdmSystem and CadSystem.

An example of the use of PDM Enablers V 2.0 and CAD Services v 1.0 for an ACTIVE_READONLY mode of interaction is displayed in Figure 3-2. This sequence diagram illustrates a typical client interaction where requests are made to create an ActiveModel from a file description. This Model is opened by the CadSystem for Read-Only operations.

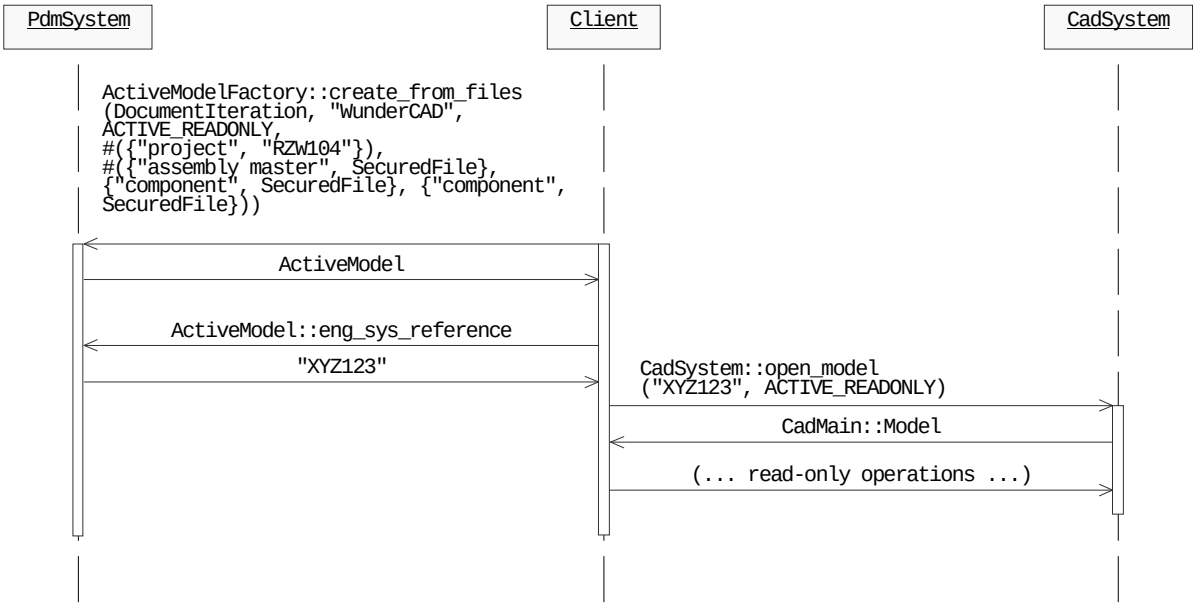


Figure 3-2 Sequence diagram illustrating CadSystem and PdmSystem interactions for Read-Only operations.

An final example of the use of PDM Enablers V 2.0 and CAD Services V1.0 displays a client accessing the Model using a find operation as shown in Figure 3-3. This sequence diagram is similar to the process illustrated in Figure 3-1, but employs a find operation to access the ActiveModel and the ActiveModel returns an object reference. The object reference is a valid **CadMain::Model** reference, which provides complete CAD Services functionality. (This object reference requires narrowing from a **CORBA::Object** scope.)

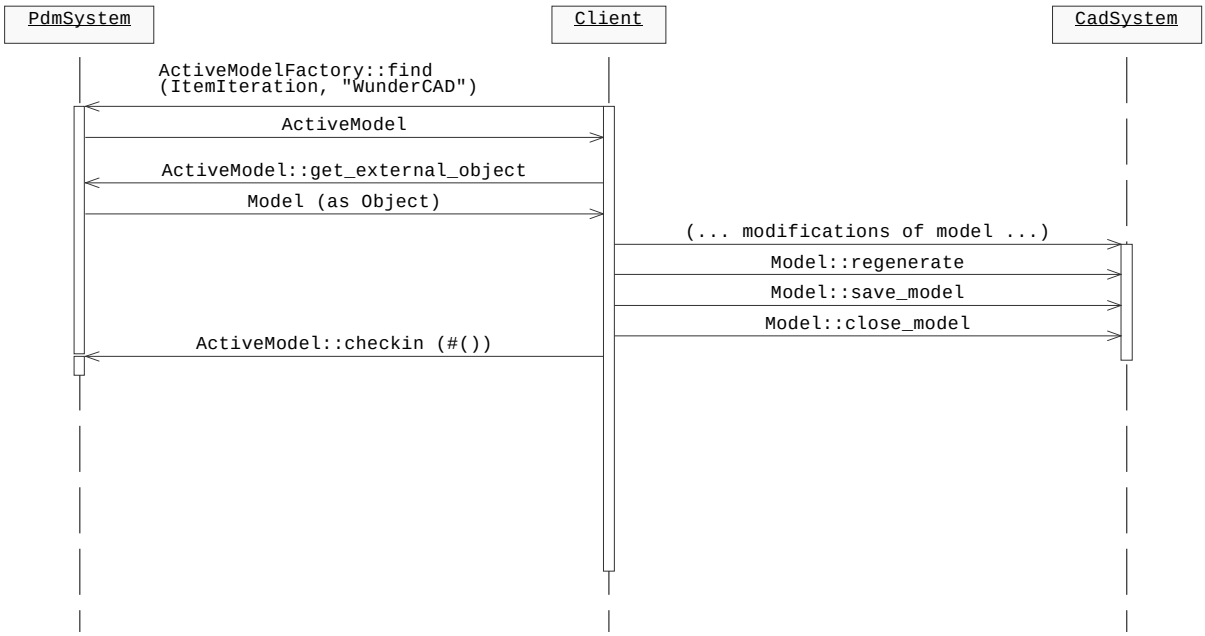


Figure 3-3 Sequence diagram illustrating CadSystem and PdmSystem interactions for a use case where the PdmSystem provides a valid CadMain::Model object reference.

3.2.1 Proposed IDL from the PDM Enablers V2.0 proposal

```
interface ActiveModel : PdmFoundation::Manageable,
   PdmFramework::Navigable
{
     readonly attribute string activation_type;
     readonly attribute IdentifierSeq eng_sys_reference;
     readonly attribute ActivationMode mode;
     readonly attribute ItemIteration product_model;
     readonly attribute ActiveFiles associated_files;

     Object get_external_object()
       raises (UnsupportedInstance, PDM_EXCEPTIONS);

     void checkin(in CosPropertyService::Properties args) raises
        (UnsupportedInstance, RELATIONSHIP_CREATE_EXCEPTIONS);

     void destroy();
};
```

3.2.1.1 Attributes

activation_type	Identifies the type of engineering system in which this **ActiveModel** has been instantiated.
eng_sys_reference	An external identifier that can be used by the client to find the model in the engineering system using the interfaces to the engineering system.
mode	The mode in which this ActiveModel was created. If the mode is **ACTIVE_READONLY**, the **checkin** operation is not permitted.
product_model	The **ItemIteration** from which the **ActiveModel** was constructed.
associated_files	The File objects that were used to construct the **ActiveModel**, along with their associated Roles.

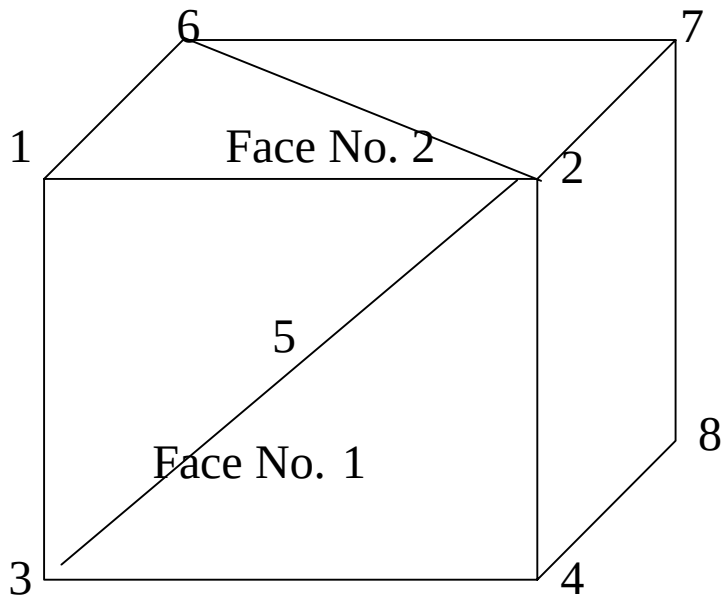
3.2.1.2 Operations

get_external_object	Returns the Object that provides interface to the activated model in the engineering software system. This returned object should be narrowed to a **CadMain::Model** interface. • UnsupportedInstance is thrown if the PDM system cannot perform the **get_external_object()** operation on this **ActiveModel** object. For example, because the PDM is unaware that the engineering system in question provides such object services, or because accessing them requires capabilities or information that the PDM does not have. • PermissionDenied is thrown if the client does not have the required permissions (and possibly other properties) for the PDM to perform this operation on its behalf. (This determination may be made by the PDM system, or made by the engineering system and reported as such to the PDM.) • PdmError is thrown if the PDM attempts the activation of engineering object services and it fails for some other reason.

checkin	Causes the updated engineering model objects to become the next Iteration of the corresponding objects in the PDM system. This may entail creation of ItemIteration and File objects and corresponding relationships. The entire interpretation of this operation is implementation-defined, and it may be restricted to a small class of external models. UnsupportedInstance is thrown if the PDM system cannot perform the **checkin()** operation on this **ActiveModel** object. Other exceptions are implementation-defined, although they will be generally as described for the item and relationship creation exceptions for the related object factories.
destroy	Causes this object to be destroyed, and causes the structures created in the PDM system to manage this instantiation of the product model to be destroyed and related resources to be released. The effect on the model in the engineering system itself is implementation-defined.

Tessellation Indexing *A*

A.1 Tessellation Indexing



The numbering for various Edge, Face, and Body Tessellations is illustrated using this unit cube. The body tessellation is comprised of a sequence of face tessellations that are joined along shared edges. The FaceTessellation for Face 1 would contain the following data:

EdgeTessellationSeq edges

A sequence of Edge Tessellations that contain:

1. Object obj_ref – An object reference to the underlying Edge.
2. PointStructSeq epts – A sequence of point structures (x, y, z) that identify the 3D spatial location of the points along the edge. EdgeTessellation coordinate data within FaceTessellation is guaranteed to be exactly coincident with corresponding face_tessellation coordinates.
3. LongSeq vertex_number – A sequence of integers that index the PointStructSeq (epts).
4. DoubleSeq t_values – A sequence of doubles that are the underlying parameters for this edge.

Tessellation face_tessellation

Basic tessellation containing:

1. Object obj_ref - An object reference that permits remote CORBA operations on Face 1 after narrowing the reference.
2. TessType t_type – An enumerated data structure indicating the use for this tessellation (WIREFRAME, VISUALIZATION).
3. PointStructSeq xyz – A sequence of point structures (x, y, z) that lie on the face.
4. LongSeq face_pts– A sequence of integers that index the PointStructSeq (xyz).
5. VectorStructSeq normals – A sequence of normals at the location defined by xyz above.
6. CadUtility::UvStructSeq uv – A sequence of uv values at the location defined by xyz above.
7. IndexSeq index_list – A sequence of index values defining connectivity of triangles. Index values use those defined in the LongSeq integers (both epts and face_pts). Index values are global at the underlying geometric representation.

In the example above, Face No. 1 is comprised of:

Edge 1 –

PointStructSeq epts = [0,1,1] and [0,0,1]

LongSeq vertex_number = 1 and 3

Edge 2 –

PointStructSeq epts = [0,1,1] and [1,1,1]

LongSeq vertex_number = 1 and 2

Edge 3 –

PointStructSeq epts = [1,1,1] and [1,0,1]

LongSeq vertex_number = 2 and 4

Edge 4 –

PointStructSeq epts = [0,0,1] and [1,0,1]

LongSeq vertex_number = 3 and 4

Face 1 -

PointStructSeq xyz = [0,1,1],[1,1,1],[0,0,3],[1,0,1],[0.5,0.5,1]

LongSeq face_pts = 1,2,3,4,5

PointStructSeq normals = [0,0,1],[0,0,1],[0,0,1],[0,0,1],[0,0,1]

IndexSeq contains:

Triangle 1 – points 1, 2 and 5

Triangle 2 – points 2, 4 and 5

Triangle 3 – points 3, 4 and 5

Triangle 4 – points 1, 3 and 5

At the body level, Face No.1 would remain as shown above and Face No 2. would be described by:

Edge 1 –

PointStructSeq epts = [0,1,1] and [0,1,0]

LongSeq vertex_number = 1 and 6

Edge 2 –

PointStructSeq epts = [0,1,1] and [1,1,1]

LongSeq vertex_number = 1 and 2

Edge 3 –

PointStructSeq epts = [1,1,1] and [1,1,0]

LongSeq vertex_number = 2 and 7

Edge 4 –

PointStructSeq epts = [0,1,0] and [1,1,0]

LongSeq vertex_number = 6 and 7

Face 2-

PointStructSeq xyz = [0,1,1],[1,1,1],[0,1,0],[1,1,0]

LongSeq face_pts = 1,2,6,7

PointStructSeq normals = [0,1,0],[0,1,0],[0,1,0],[0,1,0]

IndexSeq contains:

Triangle 1 – points 1, 2 and 6

Triangle 2 – points 2, 6 and 7

Use Case Scenarios and Examples *B*

There are a number of different approaches for using the CAD Services. The scenarios described below demonstrate how, for example, a CAE application will use the CAD interfaces being proposed in this RFP. These scenarios intend to assist implementers and users in understanding the motivation behind some of the requirements. This list represents just a few of the many possible scenarios.

B.1 Stand Alone

In this scenario, the native CAD application is integrated with an application on a single computer via CAD services interfaces and no external (distributed) requests are needed. The integration process is eased and the developer is shielded from the complete CAD API through the use of CAD Services interfaces. In addition, the use of CAD independent (neutral) interfaces enables same application to be built against different CAD system with very little changes to code or not at all.

B.2 CAD Services as Geometry Server (Non-Interactive Mode)

In this scenario, an application plays the role of a client and the CAD system plays the role of a geometry server. The client and the server interact without visual presentation of these requests and could run in a distributed and heterogeneous environment or on a single host in a co-located mode.

For example, a CAE client application uses geometry and attributes from a part or assembly to conduct analysis. The user interface is managed by the client application or by some other visualization tool. It is also possible that the client application will store some analysis result or modified attributes back to the CAD file.

It is important to note that the CAD system's user interface was not available or not used. In a case where the CAD system architecture couples the geometric modeler and GUI, this scenario is still feasible if the CAD System runs on a host machine with a display device.

B.3 Interactive Mode

Application plays the role of a client and the CAD system plays the role of a server, however, the CAD system is part of the user interface component of the application..

Many applications use the CAD system as the primary display mechanism in addition to just a source for part geometry. They need the ability to select geometry and to display the modified geometry. While the scenario suggests a stand-alone mode (client application and server reside on same host machine), many new applications are de-coupled from the CAD system and require remote access to a CAD session.

B.4 Multi-CAD Interoperability

CORBA support for heterogeneous and distributed programming could also enable the CAD Services as an environment for implementing client application that operates on different CAD systems simultaneously. Since the client side binding is system independent, client application does not need to be compiled with multiple vendors' libraries. For example, consider a client application that uses vehicle platform data for analysis, and uses the result to design or validate other vehicle components. If vehicle platform and components are designed in different CAD Systems, CAD Services could enable the interoperability without an intermediate step such as writing the data to a file or data exchange of the complete data-set.