

---

# ORACLE DESIGNER API: MULTILINE TEXT AND OTHER SECRETS

Douglas Scherer  
*Core Paradigm*

Peter Koletzke  
*Millennia Vision Corporation*

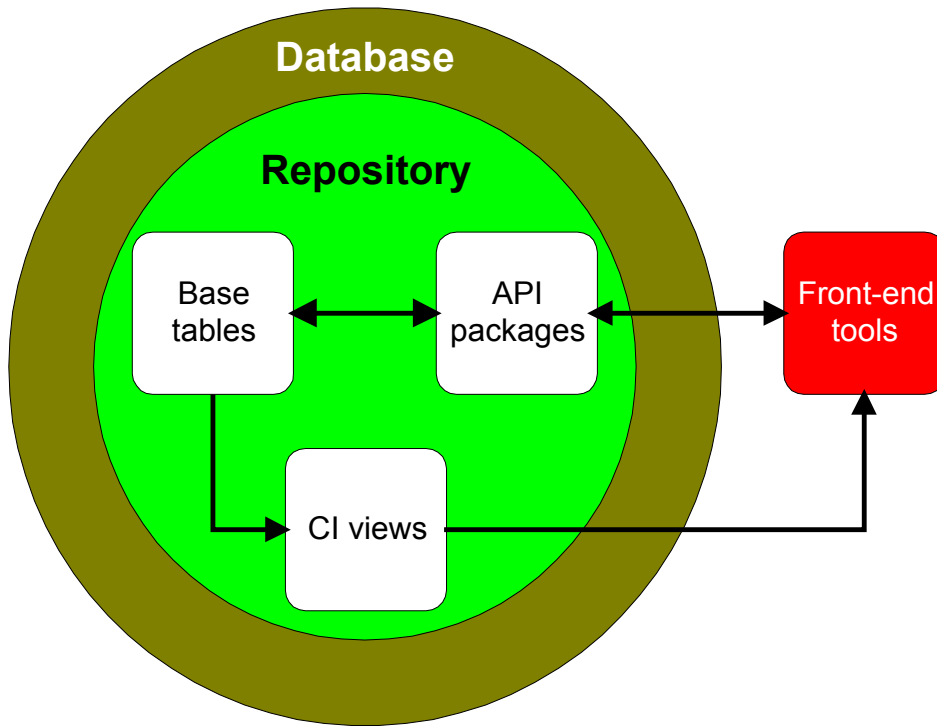
At some point in your Oracle Designer work, you will find something that you want to do that the tools will not do for you. The good news is that you have access to the same application programmatic interface (API) calls that the Oracle Designer tools use. With this API, you can build your own front-end utilities to perform repetitive tasks, write customized reports or create your own front-end GUI tools. Knowledge of the API is not optional if you want to do any serious work with the tool.

This paper provides the reader with an overview of the Oracle Designer API and provides some sample real-life uses. It presents additional features that are not well documented, such as the routines that allow you to manipulate text-like notes and descriptions. To help you get started with this powerful feature, the paper will provide some sample code. This code is also available from the authors' web sites.

## Overview of the API

The API is a set of database views and PL/SQL packages in the repository owner's schema that allow safe access to the repository data (or meta-data). The repository consists of a relatively small number of tables that store the actual data. These tables have complex (undocumented) relationships, and Oracle does not support direct access to them using standard DML SQL statements. There are, however, many views of these tables that represent actual repository objects, such as entities and attributes. These views are an important part of the API because they allow you to examine the definitions you create in your application systems.

The API also consists of the PL/SQL packages that allow you to change the contents of the tables safely outside of the Oracle Designer front-end. These packages allow you to supplement the Oracle Designer diagrammers and utilities with your own front-end programs or code. The Oracle Designer tools also use the API to insert, update, delete, and select data from the repository. Therefore, when you use the API, you are using the same method Oracle Designer uses to manipulate the repository. Figure 1 shows how the API works with front-end tools to access and manipulate the repository data. Note that API packages provide the only method for writing to the repository, and all front-end tools use them, whether they are Oracle Designer tools or your own.



**Figure 1: API access to the repository**

### Getting Started with the API

In addition to an understanding of the SQL and PL/SQL languages, you need a good understanding of the elements and properties that you want to modify or manipulate. The steps you take when using the API are typically the following:

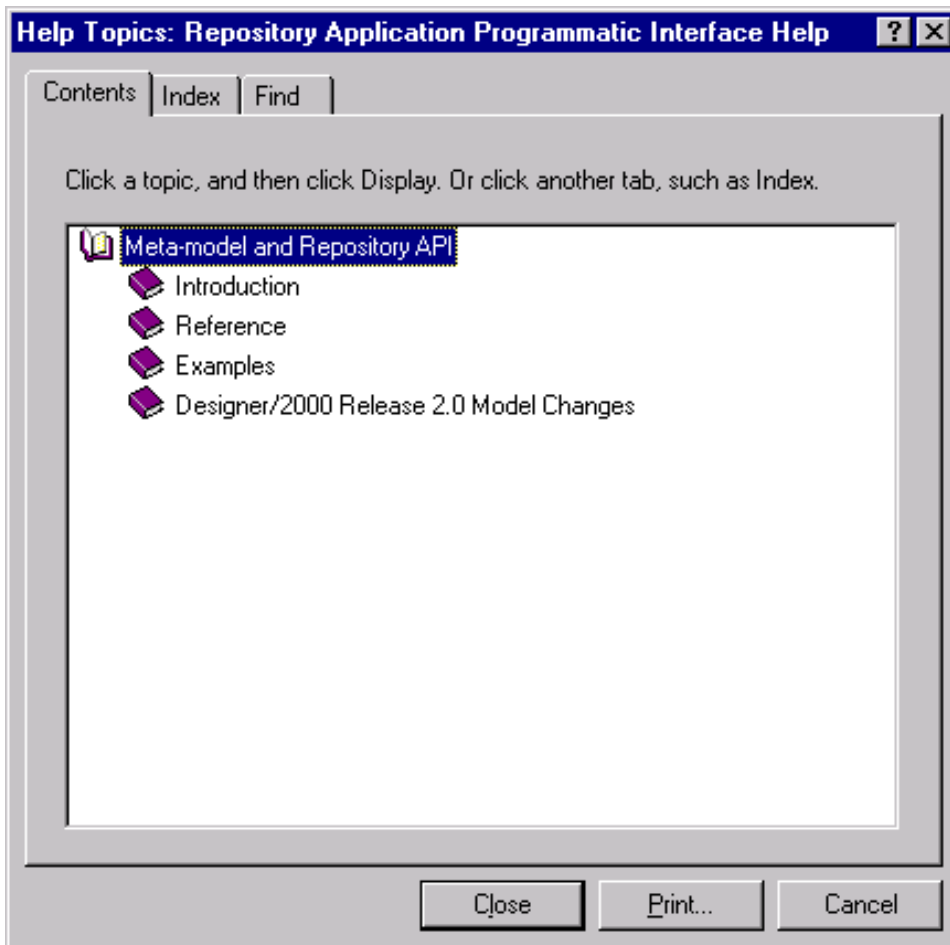
- Identify the views and packages you need.
- Obtain detailed information on those views and packages.
- Create the API code.

After you get some experience with the API, the first two steps may take no effort at all if you know the details of the views and packages. With or without experience, the first step should be straightforward. If you have suitably analyzed your needs, you should already know what elements are involved in the activity you wish to perform, or you should at least be able to look in RON to find their names.

Once you know the names of the element or association types, you can consult the documentation for more information. The main documentation for the API is in the online help system and in diagrams of the meta-model views that are shipped with the product.

### API Help (Designer/2000 from Start Menu)

Repository Application Programmatic Interface Help, shown in Figure 2, is available from the Programs group of the Windows Start menu by selecting **Designer 2.1→Designer 2000 API**.



**Figure 2: API Help contents**

From this topic, you can load other topics that explain the meta-model in general and provide reference information as well as a sample program. The reference information contains topics on the API views (View/Element Type Definitions) and packages (API Call Information). Once you know the elements you are using, you can use these two topics to obtain detailed information on the calling syntax, view columns, and related views.

**Note**

You will also find interesting and useful information in the Repository API Software Release Bulletin (SRB) that you can access to by clicking Designer 2.1 Bulletins→Repository API SRB. For example, items 2 and 3 in the SRB will inform you that even though the style and general manner in which you use the API has not changed from version 1.3 of Oracle Designer, some of the package interfaces have changed because of additional and removed elements.

### **The Meta-Model Diagram**

Oracle Designer ships with a set of diagrams, in a set of hard-copy documentation called "Repository Model," that shows the API views and their relationships. These are standard Server Model Diagrams, with foreign key relationships between views and a notation of the view name and primary key. The views are grouped into the following subsets, with each subset taking up one page:

- Business Requirement Model
- Business Planning Model
- Database Administrator Model
- Database Schema Model
- Index Storage and Partitioning Model
- Detailed Module Design Model (General)

- Detailed Module Design Model (PL/SQL)
- Module Design Model
- Server Model
- Object Database Designer Model

Which model you need to look at depends on the type of element you are manipulating. If you want to examine Design phase elements such as tables and columns, the Database Schema Model is appropriate. If you want to view the relationships between meta-model views for the application modules, you consult the Module Design Model.

### Using the Meta-Model Application System

Oracle Designer ships with an application system that contains the definitions of all meta-model views with their columns and relationships. You can load this application system into the repository from a .DAT load file or a .DMP archive file. The files are called MODEL\_20.DAT and MODEL\_20.DMP, respectively, and are located in the ORACLE\_HOME/des2\_70/model directory. You load the .DAT file into an empty application system using **Utilities→Load** in RON. Alternatively, you can import the .DMP file using **Application→Restore** in RON to create the application system. Either method will load the view definitions into the repository.

Once you have the view definitions loaded in an application system, you can create report modules based on them and use the Report Generator to generate Oracle Reports code for the module. One benefit of creating the reports this way is that, if you are using Oracle Designer to generate code, you already know how to generate report modules and do not have to be concerned with the details of the Reports tool. In addition, the repository reports you write will have a standard appearance, and you can generate all supported report types, including matrix and drill-down reports.

### API Naming Convention

The repository base tables have names with prefixes such as SDD\_ and CDI\_ --for example, SDD\_ELEMENTS, SDD\_STRUCTURE\_ELEMENTS, and CDI\_TEXT. These tables are highly normalized, and a handful of tables serves as the basis for many different element types. For example, table definitions are stored in the SDD\_ELEMENTS table with a value of 'TAB' for the column EL\_TYPE\_OF and a value of 'TABLE' for the EL\_OCCUR\_TYPE. You can always look at the view definitions (in the USER\_VIEWS data dictionary view) to see how the definitions for a particular element are stored in the SDD\_ELEMENTS table, but all you really need to know is the name of the view.

Most API views have names starting with CI\_ and ending with the plural name of the element. For example, the view that shows entity definitions is called CI\_ENTITIES, and the view that shows table definitions is called CI\_TABLE\_DEFINITIONS. Once you have determined the names of the views by looking in the help system topics as mentioned before, you can access the names of the columns in those views in the same help area. The column information consists of the datatype, size, and in most cases a brief description of the column and its valid values.

#### Tip

There are columns in all repository views called CREATED\_BY and CHANGED\_BY. You can use these to determine when the definition of a particular element was inserted or updated, and you can, for example, issue a query that lists all table definitions that have been created or changed in the last week.

### API Code

Once you have identified the views and packages and know the details of each, you can create the SQL or PL/SQL script to do the work. If you are just querying the views, a SQL SELECT script will work, although you can also use the API packages. If you are performing an insert, update, or delete operation on a repository view, you need to write a PL/SQL script to do it. This can be stored as a procedure or package in the database or run as an anonymous block from any SQL front-end tool such as SQL\*Plus. The SQL SELECT syntax is no different from the code you use to perform DML. However, special considerations and routines apply to the PL/SQL syntax, as described later.

## API Querying

View columns correspond roughly on a one-to-one basis with the properties of the element that the view represents. Each view has an ID column containing a number that identifies the element uniquely within the repository. Most element views also have a NAME column that holds the name you see in RON. In addition, there are other columns that correspond with the properties of the element. For example, the CI\_ATTRIBUTES view has such columns as FORMAT (for the datatype), OPTIONAL\_FLAG (which indicates whether the attribute is nullable), MAXIMUM\_LENGTH, and DEFAULT\_VALUE. If you know the properties of an attribute, you will be able to interpret what these view columns contain. In addition, the help system description of the columns can assist.

The CI\_ATTRIBUTES view also contains a column called ENTITY\_REFERENCE that contains a number acting as a foreign key that references the ID column of the CI\_ENTITIES view. With this information, you can construct the following query to show details on all attributes in the EMPLOYEE entity:

```
SELECT a.name, a.format, a.maximum_length, a.optional_flag
FROM ci_entities e, ci_attributes a
WHERE e.id = a.entity_reference
AND e.name = 'EMPLOYEE' ;
```

If you have an EMPLOYEE entity in more than one application system, you will need to work the CI\_APPLICATION\_SYSTEMS view into the query. This view displays information on the properties of the application systems. The unique identifier is ID, as usual, but the foreign key column in CI\_ENTITIES that refers to the application system is APPLICATION\_SYSTEM\_OWNED\_BY. Therefore, the query to find attributes of an EMPLOYEE entity owned by an application system called CTA, version 1, would become

```
SELECT a.name, a.format, a.maximum_length, a.optional_flag
FROM ci_entities e,
     ci_attributes a,
     ci_application_systems s
WHERE e.id = a.entity_reference
AND s.id = e.application_system_owned_by
AND s.name = 'CTA'
AND s.version = 1
AND e.name = 'EMPLOYEE' ;
```

If you do not know the version number and want to ensure that you are accessing the current application system, remove the s.version = 1 and replace it with s.latest\_version\_flag = 'Y'.

You can use this same strategy—finding the element names and properties in the help system, checking for foreign key relationships, and constructing the SELECT statement with standard SQL syntax—for most API queries.

## Getting API Information from within RON

### Secret 1

You cannot see the ID number for elements in RON or the Design Editor (or Object Database Designer). To find an element's ID, select the element in the hierarchy, click a property in the property palette, and press F5. A Property Details window will appear that contains the ID of the element along with the object type, base table name, and details of the property. This feature is extremely handy if you want to query an element using the view and need the ID.

A sample Property Details window is shown in Figure 3. This window is based on the *Latest Version ?* property found in the Property Palette of an application system.

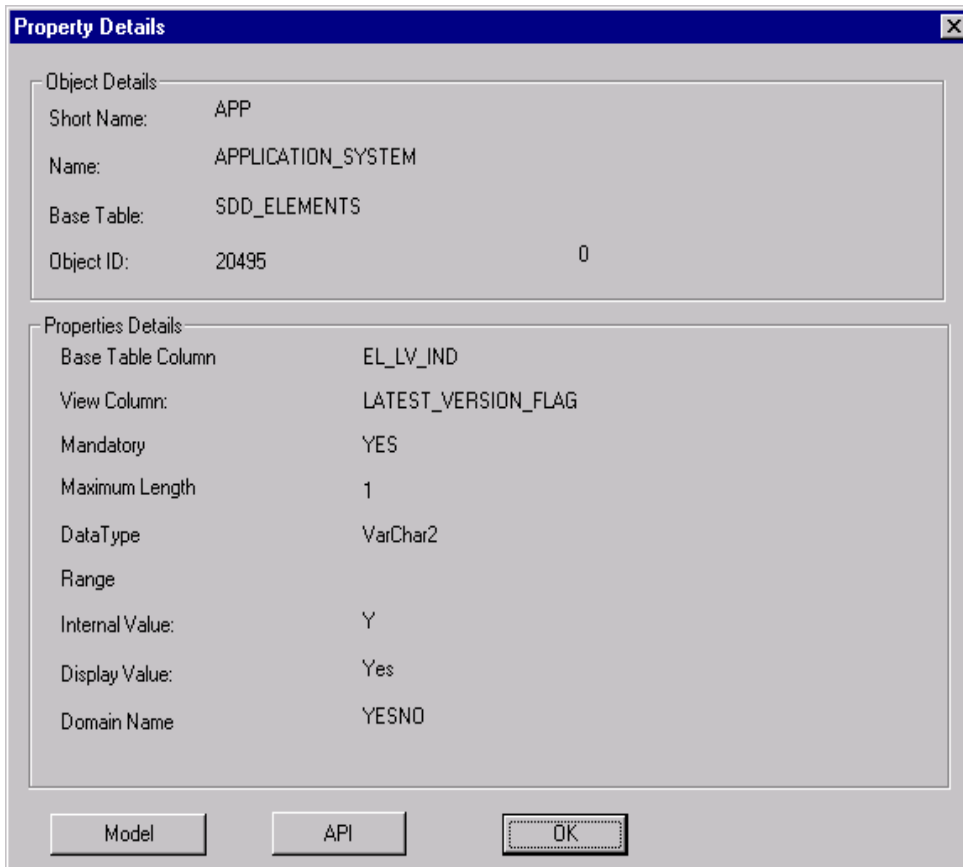


Figure 3: A Property Details window

The Property Details window is divided into two sections. The top section (labeled Object Details) provides base table and view (that is, element) information while the bottom section (labeled Properties Details) shows detailed information about what is stored in the base table (that is, property information). If you review Figure 3, you will find the following:

- *Short Name* A short name for the type of element. It is taken from CI\_APPLICATION\_SYSTEMS.ELEMENT\_TYPE\_NAME.
- *Name* The name of the view that is used to retrieve the property data. In this example the name is APPLICATION\_SYSTEM. Since repository view names have the prefix CI and are plural, the name of the view is CI\_APPLICATION\_SYSTEMS.
- *Base Table* The name of the actual table from which the view specified in Name gets its data. The view CI\_APPLICATION\_SYSTEMS is based on the table SDD\_ELEMENTS.
- *Object ID* Almost every CI view has an ID column that uniquely identifies an element instance. In the example, this value is drawn from CI\_APPLICATION\_SYSTEMS.ID.
- *Base Table Column* The name of the column in the base table that contains the Latest Version? (SDD\_ELEMENTS.EL\_LV\_IND).
- *View Column* The name of column in the view that contains the Latest Version? (CI\_APPLICATION\_SYSTEMS.LATEST\_VERSION\_FLAG).
- *Mandatory* States whether this property is mandatory or optional.
- *Maximum Length* States the maximum length of the Internal Value (described below) of this property.
- *DataType* The datatype of the Internal Value.
- *Range* For a property that stores a numeric value there may be a low and high numeric domain. *Latest Version ?* does not contain a numeric value so there is no range. If you look at the Property Details window for the *Version* property you will see that the range is from 1 through 999.
- *Internal Value* This is what actually is stored in the SDD\_ELEMENTS table. Maximum Length and DataType both refer to the storage of the value in the base table.
- *Display Value* Represents how the Internal Value is represented in the Property Palette. The value for *Latest Version ?* is stored as “Y,” but is displayed as “Yes.”
- *Domain Name* An internal domain Oracle Designer uses as a valid values list for the property. This list is maintained in the REF\_VALUES table. So, if you want to find all the possible values for this domain you could issue the query:

```
SELECT *  
  
FROM ref_values  
  
WHERE ref_domain = 'YESNO' ;
```

In addition to the values shown, the Property Details window includes these buttons:

- **Model** Opens an HTML page displaying the Property Detail information for each of the properties on the current Property Palette.
- **API** Opens an HTML page displaying information about the API package that is used to access the information in the current Property Palette. You can also retrieve a list of each API package specification in the speclist.html file in the ORACLE\_HOME/CDOC70/api/specs directory.

## API Transaction Model

### Note

To be most effective when using the API, you need to be relatively fluent with the SQL and PL/SQL languages. If you are using only the view component of the API, the SQL language SELECT statement will serve you well. If you intend to modify or store repository data with the API packages, you will also need a good understanding of PL/SQL control structures, packages, and record variables.

### Overview

More than 400 PL/SQL packages make up the API. These are organized like the views, so there is one package (prefixed with CIO) for each of the repository element types--for example, CIOCOLUMN, CIOTABLE\_DEFINITION, CIO\_VIEW\_DEFINITION, CIOAPPLICATION\_SYSTEM. You can think of the views as base tables that have a one-to-one relationship with the object types in RON, and the packages as the code you use to perform DML on those tables. Each package has procedures with the following names:

- **INS** for insert operations
- **UPD** for update operations
- **DEL** for delete operations
- **SEL** for select operations

For example, the CIOCOLUMN.INS procedure creates a new record in the repository tables as displayed through the CI\_COLUMNS view. These API procedures perform validation checking, so if a value you are updating relates to or affects another object, the repository state will not be corrupted. For example, if you try to remove a primary key column that is referenced by a foreign key constraint, the API will stop you because the constraint would then be invalid. Although you can select directly from the view as shown above, the package allows you to select from it as well (using the SEL procedure).

As mentioned, the help system for the API contains calling information for these packages.

### Package Contents and Sample Code

The procedures in the API package use as parameters the ID number of the element you want to work on (for UPD, DEL, and SEL) and a record variable that contains the data (for INS, UPD, and SEL). Each package has a record variable called data (actually a record of records) that you can use to type a variable in your PL/SQL block. For example, if you design capture a table, its *Display Title* property will be null. If you want to fill the *Display Title* property of all the design captured tables with a value similar to what the Database Design Transformer creates, you can declare a variable, in a PL/SQL block of your own, as follows:

```
DECLARE
```

```
    r_tab  ciotable_definition.data;
```

You then load a member of this record variable with the value you want to use to update the property of the element:

```
    r_tab.v.display_title := 'Employees';
```

The "v" member indicates that this is a value. The "i" member acts as an indicator to the API that you have changed this property:

```
    r_tab.i.display_title := TRUE;
```

The last step is to issue the update statement, passing as parameters the ID number of the column definition and the PL/SQL record that contains the updated data and indicators.

```
    ciotable_definition.upd(v_tabid, r_tab);
```

Other API package calls are needed to create a transaction and close it correctly, but the essence of the operation is in loading the record variable and passing it to the API procedure. The next section discusses the transaction model and the code you need to write to implement it. For a complete example of a PL/SQL block that calls all necessary routines to access the API, click the "PL/SQL Program" topic link on the main contents page of the API help system.

## Performing API Transactions

### Caution

Remember that before you try any technique, outside of the Oracle Designer front-end tools, that alters your repository you should either backup your database or export your repository information. In this way, you will have a way to recover in case an error is made.

In addition to the API calls already mentioned, there are API calls that implement a transaction model that handles the logical beginning and end of a unit of work. This is somewhat like the standard SQL transaction model that uses COMMIT and ROLLBACK statements to mark and reverse transactions. However, the API transaction model uses a few more steps and methods to handle errors.

The intention of the transaction model is to provide a way to validate statements as a set rather than as individuals. During an API transaction, or activity, Oracle Designer constraints and rules defined for elements can be temporarily violated without aborting the operation. For example, when you enter the definition for a relationship between two entities, you have to make an entry for both ends of the relationship. However, if there is no concept of an activity and you enter the first relationship end without the second, an error state will occur, and the action will be rejected. The transaction model temporarily disables rule and validation checking until you state that the transaction is complete. This allows you to establish complex associations and dependencies as the transaction is taking place, but it still enforces the rules and validations at the end of the transaction.

The transaction model is handled mostly by the CDAPI package and consists of the steps listed in Table 1 below.

Step	Sample Call	Notes
Initialize	<code>cdapi.initialize('CTA', 1);</code>	Declares which application system and version you are using. You need to perform this step only once each session for all statements in a particular application system.
Open	<code>cdapi.open_activity;</code>	Starts the activity (transaction).
Load record variable values	<code>r_tab.v.display_title := 'Emps'</code>	Populates a value in the record variable, representing a potential change to repository property.
Load record variable indicators	<code>r_tab.i.display_title := TRUE</code>	Signifies that a particular property value is changing.
Perform the "DML"	<code>ciotable_definition.upd(3712, r_tab)</code>	Updates a repository definition.
Validate	<code>cdapi.validate_activity(v_status, v_warning)</code>	Checks whether the action succeeded (a <code>v_status</code> of 'Y' is returned if the transaction succeeded).
Report Errors	<code>cdapi.instantiate_message</code>	Returns an error message and takes as parameters values from the <code>CI_VIOLATIONS</code> view that is loaded automatically when an error occurs.
Close	<code>cdapi.close_activity(v_status)</code>	Validates the data and state at the end of the transaction.
Abort upon failure	<code>cdapi.abort_activity;</code>	Rolls back the transaction if an error occurs in the close process.

**Table 1: Steps in the API Transaction**

The following sample uses some of the calls shown in Table 1 to update the *Display Title* property of a table.

### Note

The user running the code must have rights to update the effected application system.

```
DECLARE
    r_tab    ciotable_definition.data;
    v_tabid  ci_table_definitions.id%TYPE;
    v_status cdapi.activity_status%TYPE;
BEGIN
    /*
    || Get information for Repository transaction
    */
    -- Find the ID of the table to be affected
    SELECT t.id
        INTO v_tabid
        FROM ci_table_definitions t,
             ci_application_systems s
        WHERE s.id = t.application_system_owned_by
             AND s.name = 'CTA'
             AND s.version = 1
             AND t.name = 'EMP' ;
    /*
    || Begin repository transaction
    */
    -- Initialize
    cdapi.initialize('CTA',1);
    -- Open
    cdapi.open_activity;
    -- Load record variable values
    r_tab.v.display_title := 'Employees';
    -- Load record variable indicator
    r_tab.i.display_title := TRUE;
    -- Perform the "DML"
    ciotable_definition.upd(v_tabid, r_tab);
    -- Close
    cdapi.close_activity(v_status);
    IF v_status != 'Y'
    THEN
        -- Abort upon failure
        cdapi.abort_activity;
    END IF;
END;
```

The two steps that load record variables may consist of multiple statements if you are inserting an element definition or updating more than one property at a time. In addition, you can choose to validate (close) the activity whenever you want. For example, you can loop through a number of records and update each before closing the activity. The validation will be deferred until the close, so if any of the records violate the constraints, the entire set will be rolled back. This behavior is similar to that of the SQL transaction model and should be familiar territory.

Having seen this simple example of an API script, you will find it easy to add the loop that would fulfill the post-design capture needs described above.

```

DECLARE

    r_tab    ciotable_definition.data;

    v_status cdapi.activity_status%TYPE;

CURSOR c_display_title

    IS

    SELECT t.id, t.name

        FROM ci_table_definitions t,

             ci_application_systems s

    WHERE s.id = t.application_system_owned_by

           AND s.name = 'CTA'

           AND s.version = 1

           AND t.display_title IS NULL ;

BEGIN

    /*

    || Begin repository transaction

    */

    -- Initialize

    cdapi.initialize('CTA',1);

    -- Open

    cdapi.open_activity;

    FOR r_display_title IN c_display_title

    LOOP

        -- Load record variable values

        r_tab.v.display_title :=

            initcap(replace(r_display_title.name, '_', ' '));

        -- Load record variable indicator

        r_tab.i.display_title := TRUE;

        -- Perform the "DML"

        ciotable_definition.upd(r_display_title.id, r_tab);

    END LOOP;

    -- Close

    cdapi.close_activity(v_status);

    IF v_status != 'Y'

    THEN

```

```

-- Abort upon failure
cdapi.abort_activity;

END IF;

END;
```

### Secret 2

You can call `cdapi.close_activity` without first calling `cdapi.validate_activity`. This will provide an increase in performance while maintaining the same level of data integrity checking, but will not display any warnings.

## Manipulating Multiline Text

### Secret 3

Despite the flexibility and support for all major element and association types in the repository, the API has some limitations. One of these is that you cannot manipulate text types such as Notes, Descriptions, Derivation Expressions, Select Text, PL/SQL Blocks, and Where clauses. The application of DML statements to this table, other than queries, is completely unsupported and discouraged by Oracle Corporation. You can, however, safely query the `CDI_TEXT` table.

The structure of the `CDI_TEXT` table is as follows:

Name	Null?	Type
-----	-----	----
TXT_REF	NOT NULL	NUMBER(38)
TXT_SEQ	NOT NULL	NUMBER(6)
TXT_TYPE	NOT NULL	VARCHAR2(10)
TXT_NOTM		NUMBER(38)
TXT_TEXT		VARCHAR2(2000)

The `TXT_REF` column is the ID number of the element that this text is used by. The `TXT_TYPE` is a code that designates the type of text. For example, a value of `CDINOT` means that this text type contains notes for the element. `CDIPLS` is the text for the PL/SQL block. A block of text can consist of more than one record, and the `TXT_SEQ` column is the line number for this record. `TXT_TEXT` is the actual text. `TXT_NOTM` is an, as yet, unused column that is intended to store the number of times that this record was modified. Using this information, you can construct a SQL statement to query the `CDI_TEXT` table and extract any text type for any element. For example, to extract the user help text for the `STUDENTS` table in the version 1 CTA application system, you would issue the following SQL statement:

```

SELECT txt_text
  FROM cdi_text
 WHERE txt_type = 'CDHELP'
    AND txt_ref =
      (SELECT t.id
         FROM ci_table_definitions t,
              ci_application_systems a
        WHERE t.application_system_owned_by = a.id
              AND a.name = 'CTA'
              AND a.version = 1
              AND t.name = 'STUDENTS')
 ORDER by txt_ref, txt_seq;
```

**Tip**

You can view a list of the text types and descriptions of those types by querying the RM\_TEXT\_TYPES view.

**A Multiline Text API Sample Script**

After performing a design capture on tables, the *User/Help Text and Description* table properties remain unpopulated. Using the API, you can write a routine to put information into these properties for all of an application system's tables based on the information in the *Comment* property. This is a bit trickier than the *Display Title* example above because *User/Help Text* and *Description* are multiline text types and are designed to store more than one row in the repository for each table instance.

**Tip**

When you work in the TextPad to edit multiline text, such as Description, you can include a carriage return in the text to indicate a new line. The carriage return "character" is saved in the CDI\_TEXT table, so if you use a select statement similar to the one shown above to retrieve the multiline text data directly from the CDI\_TEXT table, you will see the Description stored as multiple lines, with a carriage return at the end of each. When you use the API to load text, remember that you can use the carriage return (CHR(10)) to create a new line.

To perform this write into the multiline text property, you may use the undocumented package RMOTEXT. Through the use of RMOTEXT.READALL and RMOTEXT.WRITEALL, you can issue transactions against the repository with streams rather than with records of scalar types. Below is an example using RMOTEXT to populate User/Help Text from Comment after design capturing tables in a schema. You could add in *Description* property population if needed. As mentioned, this is undocumented, and, although it uses a standard API-like interface, you should be careful to back up your repository before using this (or any other) utility that you create outside the Oracle Designer front-end interface.

```

DECLARE

    v_status          cdapi.activity_status%TYPE;
    v_bufwrt          integer;
    v_stream_handle  rm.stream;

    -- Declare cursor for User/Help Texts
    -- The result set of this cursor will contain
    -- the ID and COMMENT from a repository
    -- table definition for which the User/Help Text
    -- does not exist. If the comment property is
    -- null the User/Help Text will not be populated.
    CURSOR c_uht -- uht = user_help_text
    IS
        SELECT t.id, t.remark, t.name
           FROM ci_table_definitions t,
                ci_application_systems s
          WHERE NOT EXISTS
                (SELECT NULL
                 FROM cdi_text
                WHERE txt_type = 'CDHELP'
                   AND txt_ref = t.id
                )

```

```

        AND s.id = t.application_system_owned_by
        AND s.name = 'CTA'
        AND s.version = 1
        AND t.remark IS NOT NULL;
BEGIN
    /*
    || Begin repository transaction
    */
    -- Initialize
    cdapi.initialize('CTA',1);
    -- Open
    cdapi.open_activity;
    -- Update User Help Text
    FOR r_uht IN c_uht
    LOOP
        -- Load stream value
        rmotext.open(r_uht.id,'CDHELP','w',v_stream_handle);
        rmotext.writeall(r_uht.id,'CDHELP',r_uht.remark,
            length(r_uht.remark),v_bufwrt);
        rmotext.close(r_uht.id,v_stream_handle);
    END LOOP;
    -- Close
    cdapi.close_activity(v_status);
    IF v_status != 'Y'
    THEN
        -- Abort upon failure
        cdapi.abort_activity;
    END IF;
END;
```

## API Sample Uses

### Note

Working with the API is straightforward once you understand the principles. However, no matter what your level of understanding is, the API routines will take time to write, which you should weigh against the time savings that the API potentially offers. Keep in mind also, though, that if the routine you are writing has a general-purpose use, you will be able to reuse it in future projects, so any extra time required creating it might be worthwhile in the long run.

While you can perform a large number of tasks with the Oracle Designer front-end tools, some things are not easy or possible with those tools. Also, even if a task is easy or possible, it may be repetitive or tedious if a large number of repository objects are affected. In addition, you may need to support access to and manipulation of a user-extended property or element. The API is the perfect facility to use in all these cases. The following sections list some possible uses for the API.

- **Loading Legacy Report Definitions** using your own front-end tool to input module definitions for legacy reports.
- **Creating Domains from Column Definitions** by querying all attributes with null domain properties and creating domains from them and assigning the new domain back to the source column. **Making Global Changes to Column Names** using a procedure that reads the column definitions and changes names to the abbreviations stored in another table.
- **Mapping Requirements to Modules** by copying the requirements associations you mapped to functions with the modules that were created from those functions.
- **Tracking Problems** using an API routine that manages the Problems element. This can help you in tracking users' comments and issues.
- **Changing Free-Format View Definitions into Declarations** using an API routine that reads the Select Text and Where Clause properties, parses out the table and column names and creates references for them.
- **Reporting on Table Usages** with an API report that shows modules, module component usages, and base tables with the insert, update, delete, select capabilities. This is useful in cross-checking the completeness of your design.
- **Reporting on Role Access to Tables** by writing a report on the user roles (definitions) that have access to the tables you have defined in the repository. You can include other database objects in the report, as well.
- **Loading Definitions from a File or Table** using an external source for data you load into the repository. The INS procedure does the work, but the external source (loaded from another tool or process) provides the data.
- **Supplementing User Documentation** using an API report of help text and other descriptive properties via RMOTEXT.READALL and the CDI\_TEXT table (not really an API view).
- **Supplementing the Repository Reports** with queries to the API views. It is highly likely that you will need a report that is not available in the Repository Reports tool and the API views allow you to get what you need.

## Developing a Forms Front-End Program for the API

You may have a need to access the repository that is not filled in the Oracle Designer front-end. You could develop a front-end program that interactively accesses the repository in the same way the Oracle Designer tools do. The API allows you to use a development tool you are comfortable with, such as Oracle Forms, to manipulate repository data. The following paragraphs outline the issues you need to consider when creating a Forms front-end program to access the repository. While this discussion is specific to Oracle Forms, you can translate the techniques into any other development tool that can connect to an Oracle database.

### The Requirement

When you write code to perform work using the repository, it is important that you limit the scope. Completely generic programming can be time-consuming, and if you never use the generalized part, it can be wasted effort. Suppose you run the Reconcile Report to cross-reference the repository to the Oracle data dictionary, and the resulting report has a number of differences in the Not Null properties of the database and repository definitions for some views. The database view definition may be based on tables that have the wrong Not Null constraint. What you really want is a utility that lets you see the data dictionary Not Null values side by side with the repository Not Null values. This utility can be a form you develop to show these properties side by side which will help you determine which values are wrong. At that point, you can click an item and change the value in the repository by issuing API calls to update the column definition.

## The API Form

The API is as easy to call from a Developer form as it is in SQL\*Plus. Forms version 5 (Oracle Developer version 2) provides PL/SQL version 2.3, which can handle direct access to the Oracle Designer API. You can develop a form that queries the data dictionary and the repository views to show the null indicator for a particular view definition. There are a few ways to do this. One way is to have a list of values from which the user can select a view definition name. Figure 4 shows the main work screen of such a utility.

**Figure 4: Form API utility**

Column	Repository Value	Data Dict Value	Differ-ent?
DATE_REGISTERED	NULL	NOT NULL	<input type="checkbox"/>
FIRST_NAME	NULL	NOT NULL	<input checked="" type="checkbox"/>
LAST_NAME	NOT NULL	NOT NULL	<input type="checkbox"/>
PHONE	NULL	NOT NULL	<input checked="" type="checkbox"/>
TITLE	NULL	NOT NULL	<input checked="" type="checkbox"/>
ZIP	NOT NULL	NOT NULL	<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>

You can then perform a query on a block based on the API view CI\_COLUMNS using the selected view name. Naturally, you will need to have the user specify which application system this form is acting on. The columns block has a POST-QUERY trigger that looks up and displays the corresponding data dictionary property value in another non-base table item. You can have a mechanism (for example, double-clicking the null indicator) that indicates that the repository value is to be changed. If the user employs this mechanism, a value is written to a non-displayed flag item that indicates a change.

If the repository and data dictionary are in separate databases or in different user accounts, you will also need to set up database links and synonyms for those other databases.

### The Forms Package

When you click the Commit button, a package procedure checks the flag item of each record to determine if the record changed. If a record was changed, the procedure calls the API routines to make the change in the repository. The form code follows the same model as code you would execute from another tool like SQL\*Plus.

## Creating a Use Case Type Diagram

Although Oracle Designer now provides the Object Database Designer, there are some modeling tools it does not provide for object-oriented analysis. One of the missing tools is a "Use Case Diagrammer." In a Use Case diagram, there are Actors (similar to Users) and Use Cases (similar to Business Functions or processes). The diagrams can graphically portray interaction between the Actors and the Use Cases.

Secret 4

The Process Modeller allows you to map a process to the Business Unit owning that process. Using the Oracle Designer API and Oracle Reports you can use that information to create a Use Case type diagram. After building the report, you can add it to the Repository Reports List.

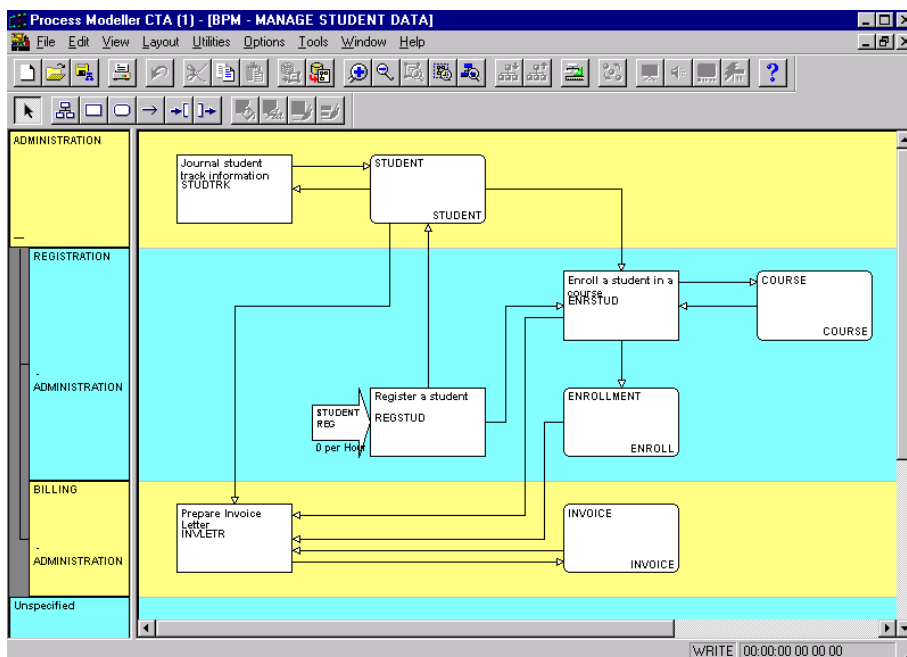
## The Report's Main Query

Given an application system and business process model, the main query of the report focuses on finding the business units and processes in that diagram. To do this, you need to write a query like the one below. Not all of the tables in the query appear in the meta-model diagrams. SDD\_ELEMENTS for example is a base table that supports many of the API views. For more information on the SDD\_ tables do a find for SDD\_ in the API help system.

Figure 5 shows the BPM - MANAGE STUDENT DATA business process model in the CTA application system which will be the basis for the main query of the Use Case type report.

### Note

The Business Process Modeller truly shows ownership of a process with a Business Unit rather than interactivity or communication with that Business Unit. So, if you choose to use this method to create Use Case diagrams you are choosing to use the Process Modeller outside of the scope of its original purpose.



**Figure 5: BPM - MANAGE STUDENT DATA**

From the information you entered into the Business Process Modeller, you can create a query like the one below to list Business Units to their Business Functions in a given diagram.

```

SELECT cif.function_label, cif.short_definition , cib.name business_unit
FROM ci_functions cif,
     ci_business_units cib,
     ci_function_business_units cifb
WHERE cif.id = cifb.function_reference
AND cib.id = cifb.business_unit_reference
AND cifb.application_system_owned_by =
    (SELECT id
     FROM ci_application_systems
     WHERE name = 'CTA'
     AND version = 1
    )
AND cib.name IN
    (SELECT el_name
     FROM sdd_elements
     WHERE el_id IN
        (SELECT cielement_reference
         FROM ci_diagram_element_usages
         WHERE diagram_reference =
            (SELECT id
             FROM ci_diagrams
             WHERE name = 'BPM - MANAGE STUDENT DATA'
             AND application_system_owned_by =
                (SELECT id
                 FROM ci_application_systems
                 WHERE name = 'CTA'
                 AND version = 1
                )
            )
        )
    )
AND el_type_of = 'BUN'
)
AND cif.function_label IN
    (SELECT el_name
     FROM sdd_elements
     WHERE el_id IN
        (SELECT cielement_reference
         FROM ci_diagram_element_usages
         WHERE diagram_reference =
            (SELECT id

```

```

FROM ci_diagrams
WHERE name = 'BPM - MANAGE STUDENT DATA'
AND application_system_owned_by =
(SELECT id
FROM ci_application_systems
WHERE name = 'CTA'
AND version = 1
)
)
)
AND el_type_of ='FUN'
)
ORDER BY 1,3 ;

```

Below is the formatted listing of the result set of this query when run in SQL\*Plus. Specifically, you are seeing a list of the Business Functions and Business Units in the BPM - MANAGE STUDENT DATA diagram in the CTA version 1 application system.

FUNCTION_LABEL	SHORT_DEFINITION	BUSINESS_UNIT
ENRSTUD	Enroll a student in a course	ADMINISTRATION REGISTRATION
INVLETR	Prepare invoice letter	BILLING REGISTRATION
MANAGESTU	Manage student data	BILLING
REGSTUD	Register a student	ADMINISTRATION REGISTRATION
STUTRK	Journal student track information	ADMINISTRATION REGISTRATION

Based on this query, you can create a report similar to the one shown in Figure 6. The report cpuct1.rdf is shown running from within Repository Reports.

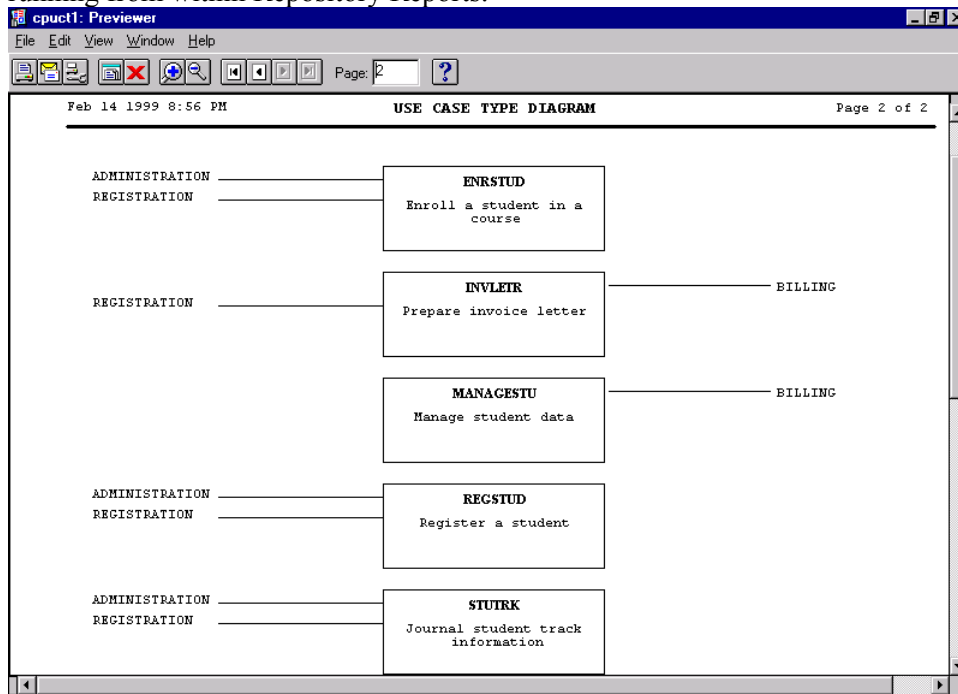


Figure 6: Use Case type report

## Summary

As you can see, the API provides a powerful way to access the repository. The main hurdle is learning about its design and how some specific objects work. The API help file, as mentioned, is the main source for documentation.

You have seen some standard repository API techniques and learned some secrets you can use to assist in your API work. These include:

- Using F5 to find API information about a property from within RON or Design Editor
- Eliminating the use of `cdapi.validate_activity` in a repository transaction to enhance performance
- Manipulating multiline text by using the `CDI_TEXT` table and the `RMO_TEXT` package
- Creating a Use Case type report by using a business process model and Oracle Reports

Lastly, you were cautioned to backup your repository before attempting the techniques discussed in this paper.

## About the Authors

**Douglas Scherer** is president of Core Paradigm, a firm providing consulting, mentoring and formal training solutions primarily for Oracle database application and management environments. He is a frequent speaker at conferences and user-group meetings internationally and has appeared in *Visions of the New Millennium*, a series seen on PBS and its affiliates. He is lead author of *Oracle 8i Tips & Techniques* and contributing editor to the *Oracle Designer Handbook*, second edition, both published by Osborne (Oracle Press). He also chairs the database track at Columbia University's Computer Technology and Applications program. He holds an M.S. from Columbia University. <http://www.coreparadigm.com>

**Peter Koletzke** is a practitioner and self-proclaimed evangelist for Oracle Designer and Developer. He is a Consulting Manager and Principal Instructor for Millennia Vision Corporation (MVC), of Redwood Shores, CA. He is also a member of the Board of Directors of the International Oracle Users Group — Americas, a frequent contributor to national and international Oracle newsletters and users group conferences, and co-author, with Dr. Paul Dorsey, of two Oracle Press books: the *Oracle Designer Handbook, Second Edition*, (1998), which was used as a source for this paper; and *Oracle Developer Advanced Forms and Reports* (1999). His paper on Developer and Designer help systems won the ODTUG 1999 Editor's Choice Award.

[http://ourworld.compuserve.com/homepages/Peter\\_Koletzke](http://ourworld.compuserve.com/homepages/Peter_Koletzke)

MVC is a new breed of business consulting firm dedicated to improving its clients' business performance and position in the expanding e-business economy. MVC's unique, integrated approach enables companies to align their people, processes, and technologies in support of their business strategy. Global leaders in a wide variety of industries rely on MVC's proven methodologies, knowledge products, professional expertise and technical depth to extend their existing business models and seize new opportunities. <http://www.mvsn.com>

## Acknowledgments

Thanks to Alice Lau, an Oracle Staff Consultant in New York City for her assistance in creating the Use Case type report.