Ferret Hacking Guide

The GNU Entity\Relationship Wished INcarnation, version 1.0.0

José E. Marchesi (jemarch@gnu.org)

Table of Contents

List of Figures

1 Introduction and generalities

Ferret (the GNU Entity/Relationship Wished Incarnation) is a CASE tool for edit data models. These data models can then be implemented on some relational data base implementation such as PostgreSQL or Mysql.

The operational model supported by the CASE tool is depicted in the Figure 1.1, where all views are fully editable:

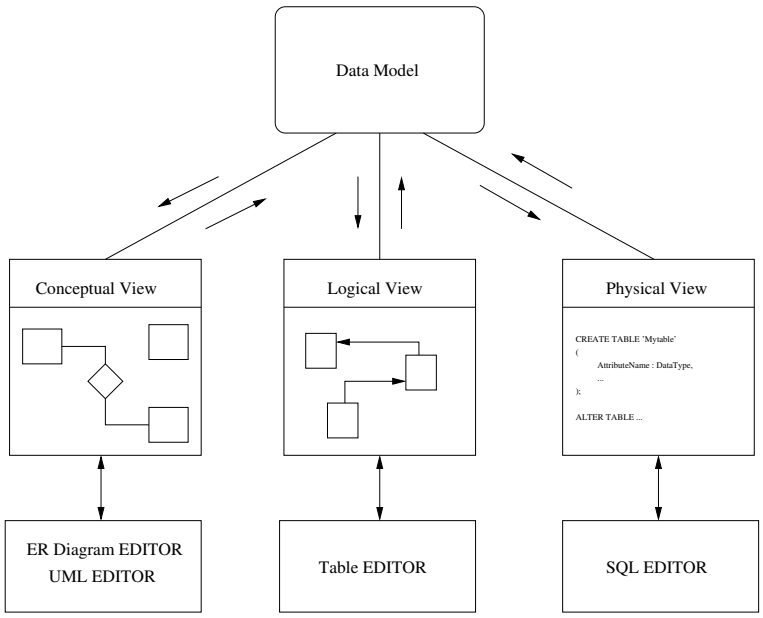

Figure 1.1: Ferret operational model

1.1 Ferret Architecture

The overall Ferret architecture is a hybrid one, involving both C and Tcl environments. See Figure 1.2.

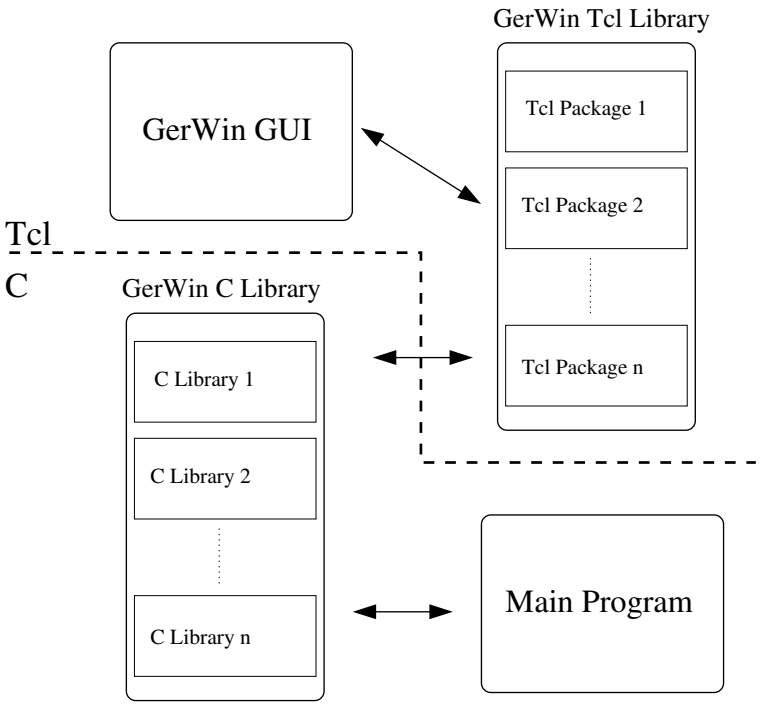

Figure 1.2: Ferret Architecture

1.2 Guidelines for writing Ferret code

1.2.1 Writing Tcl code

1.2.1.1 About Tcl Packages

- Tcl package version numbers should be of the form:

  `MAJOR_NUMBER.MINOR_NUMBER`

  These numbers should match the Ferret major and middle version numbers, respectively. So, for example, package *Diagram 1.0* is intended to run on *Ferret 1.0.x*, while *Diagram 1.1* should run on *Ferret 1.1.x*.

  Since Ferret 0.x do not use formalized packages *major_number* should never contain 0.

1.2.1.2 About Tcl Namespaces

- Follow the rule *"one namespace mean one package"*. And reverse it too!.
- Confine a namespace into one unique file, named after the namespace. Eg. *namespace Diagram*, '`diagram.tcl`'.

  If you need more than one file, then consider to split the namespace into several sub-namespace.
- Never implement namespaces in the "sparse way":

```
namespace eval foo {}

proc foo::a_procedure {...} {...}
proc foo::another_procedure {...} {...}
variable foo::a_variable
```

Instead, encapsulate the namespace contents inside it, as in the following example.

```
namespace eval foo {

  variable a_variable

  proc a_procedure {...} {...}

  proc another_procedure {...} {...}

} ;# End of namespace foo
```

In that way, it is very easy to determine the entire contents of a namespace.

1.2.2 Writing C code

When writing C code for the Ferret project, please carefully follow the guidelines of the GNU Coding Standards (GCS).

The latest release of the GCS (along with other interesting information for GNU maintainers) is always available on http://savannah.gnu.org/projects/gnustandards.

2 The data model

The data model is the heart of any running Ferret. From an abstract point of view, it is the whole object being edited by the tool. According with the general edition model, the Ferret application is the *editor*, and the data model is the *object under edition*. The user can then manipulate several aspects of the datamodel by using the appropiate view:

- Entities, relationships and class/subclass relationships are manipulated via the E/R editor, editing the conceptual view of the datamodel.
- Relations are manipulated via the relational editor, editing the logical view of the datamodel.
- Physical issues like cluster indexes and DBMS dependant data types are manipulated via the SQL editor, editing the physical view of the datamodel.

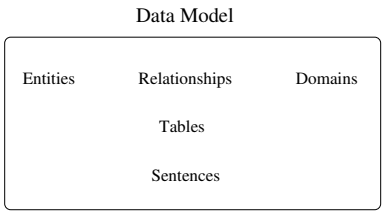

Figure 2.1: Data model

So the datamodel is an abstract object from which several views are made, at convenience. Note that each view (conceptual, logic and physical) has its own abstraction level, from higher to lesser. Because that the datamodel should maintain dispar information pertaining to different abstraction levels, such as entities, relationships, relations, cluster indexes, etc.

When the user changes some aspect of the datamodel (editing a view) the datamodel should reflect these changes in order to maintain the global coherence. So if the user adds a new entity to the datamodel, the corresponding table and SQL sentence should be also added. In this respect the datamodel views should be seen by the editors as opaque objects.

2.1 Data model implementation

The data model implementation is splitted into two distinct layers:

Layer 1

The layer 1 contain the actual in-memory structures that hold the data model contents.

Data model contents (the components of all views: entities, relationships, relations, etc) are structured as *objects*. This layer offer an uniform interface to manage data model objects.

Layer 2

The layer 2 contain the implementation of the final APIs presented to the rest of the application. Both C and Tcl interfaces are provided.

Each view of the datamodel is implemented in this layer.

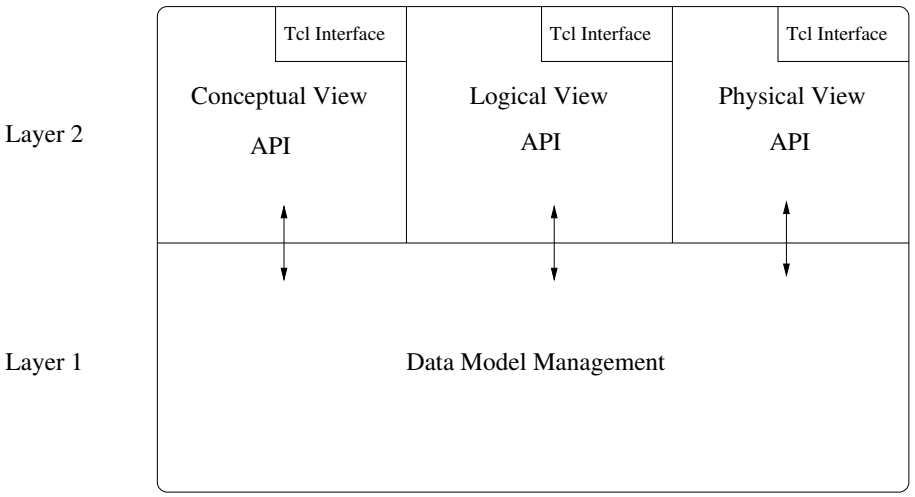

Figure 2.2: Data model layers

The rationale of the layer structuration (See Figure 2.2,) of the data model is to ease the introduction of both new data model elements ("objects") and new views. Every view can use information of any defined data model object type defined on the layer 1.

2.2 Data model objects

Any data model manages several objects pertaining to several types of objects. Entities, relationships or CSR (class/subclass relationships between entities) are examples of types for data model objects.

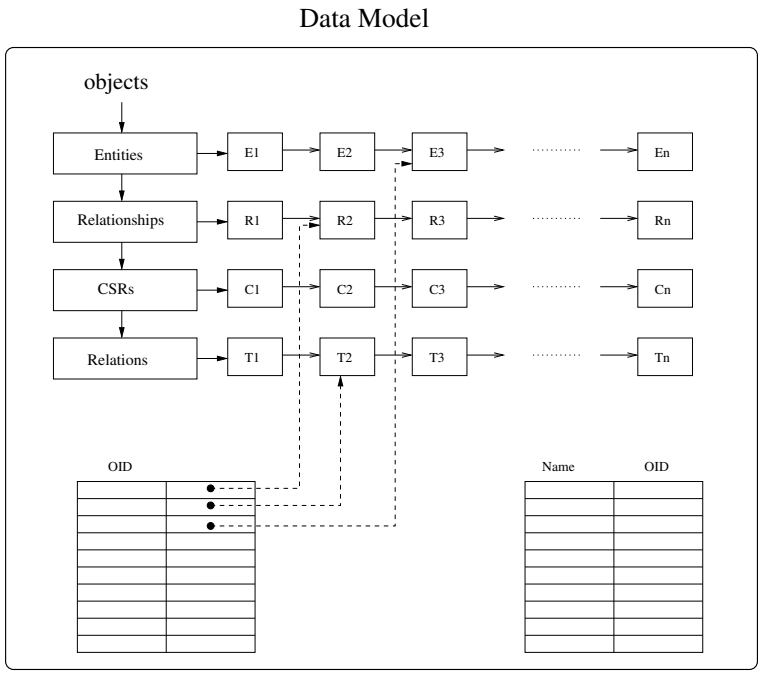

Figure 2.3: Data model and objects

The Figure 2.3 depicts the internal structure of a Ferret data model. There is a collection of objects catalogued by the corresponding type.

There is also two global tables:

The OID table
: This table stores the relationship between objects and unique integer identified called OIDs (Object IDentifier).

  The data model library should create and maintain a OID for each object stored on the data model.

The names table
: The data model library allow to assign a name to one or more objects. The user of the datamodel can define a collection of equally named objects.

2.3 Object types

Each object stored in a data model should pertain to some object type. The data model know about several object types: entities, relationships, etc. The data model implementation should be scalable enough to manage any arbitrary number of object types.

Any object type should provide the following information in order to be registered into a data model:

A data structure that describes the object type
A data structure containing the specific data for any object of that type
For example, an entity object type should manage an entity name, attributes, documentation, etc.

Callbacks to be called on:

Object creation
`void *create_object_callback ()`

Object destruction
`destroy_object_callback (void *)`

Storing object data
`store_data_callback (void *)`

Retrieving object data
`void *retrieve_data_callback ()`

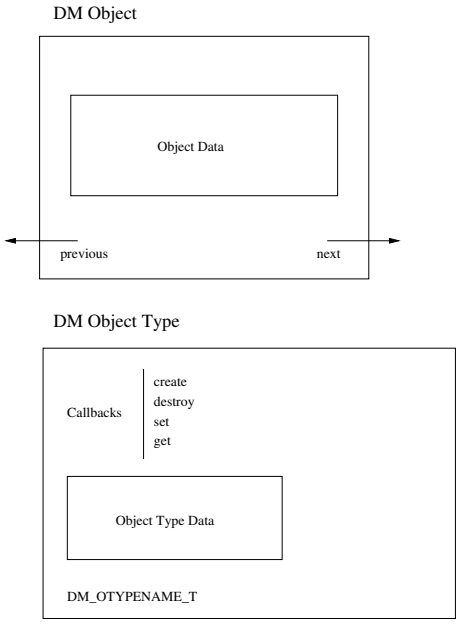

Figure 2.4: Object types

2.3.1 Data model api for objects

`ferret_string_t dm_object_get_name` (*dm_oid_t oid*) [Function]
Return the name of the object identified by OID, or NULL if it is an unnamed object.

`dm_oid_t dm_object_create` (*int object_type, void *object_data*) [Function]
Creates a new object into the data model of type OBJECT_TYPE. The object is created to contain OBJECT_DATA.

The `otype->callbacks->create` callback is called with OBJECT_DATA.

`state_t dm_object_destroy` (*dm_oid_t oid*) [Function]
Destroys the object identified with OID.

The `otype->callbacks->destroy` callback is called.

`state_t dm_object_destroy_by_name` (*ferret_string_t name*) [Function]
Destroys the collection of objects identified by NAME.

The `otype->callbacks->destroy` callback is called for each object in the collection.

`state_t dm_object_set_data` (*dm_oit_t oid, void *object_data*) [Function]
Set OBJECT_DATA as the private data of the object identified by OID.

The `otype->callbacks->set` callback is called with OBJECT_DATA.

`state_t dm_object_set_data_by_name` (*ferret_string_t name, void *object_data*) [Function]
Set OBJECT_DATA as the private data of all objects called NAME within the datamodel.

`void *dm_object_get_data` (*dm_oit_t oid*) [Function]
Get the private data of the object identified by OID.

The `otype->callbacks->get` callback is called.

2.4 The Data Model API

2.4.1 Entities

2.4.1.1 Creation and destruction

`ferret_status_t dm_create_entity` (*fdm_t fdm, ferret_string_t ename*) [Function]
Create a new entity called *ename*.

`ferret_status_t fdm_destroy_entity` (*fdm_t fdm, ferret_string_t ename*) [Function]
Destroy the entity called *ename*.

2.4.1.2 Testing

`ferret_bool_t fdm_entity_p` (*fdm_t fdm, ferret_string_t ename*) [Function]
Return *TRUE* if *ename* names an entity on the datamodel. Return *FALSE* else.

`ferret_bool_t fdm_entity_attribute_p` (*fdm_t fdm, ferret_string_t ename, ferret_string_t aname*) [Function]
Return *TRUE* if *aname* names an attribute inside the *ename* entity.

2.4.1.3 Getting and setting entity data

`int fdm_get_entity_num_attributes` (*fdm_t fdm, ferret_string_t ename*) [Function]
Return the number of attributes *ename* has.

`int fdm_get_entity_type` (*fdm_t fdm, ferret_string_t ename*) [Function]

Return the type of entity *ename*.

Legal values are:

`FDM_ENTITY_TYPE_NULL`
: The entity has not a defined type. This should always be considered as an inconsistency on the data model.

`FDM_ENTITY_TYPE_STRONG`
: The entity is a strong entity (see the user's guide for more details about strong entities).

`FDM_ENTITY_TYPE_WEAK`
: The entity is a weak entity with existence constraint (see the user's guide for more details about weak entities).

`FDM_ENTITY_TYPE_WEAKE`
: The entity is a weak entity with existence constraint and ID-dependant (see the user's guide for more details about weak by ID entities).

`ferret_status_t fdm_set_entity_type` (*fdm_t fdm, ferret_string_t ename, int etype*) [Function]

Set the type of *ename*. See the documentation for `fdm_get_entity_type` for information about legal types.

`ferret_status_t fdm_change_entity_name` (*fdm_t fdm, ferret_string_t ename, ferret_string_t nename*) [Function]

Rename the *ename* entity to *nename*.

`ferret_string_t fdm_get_entity_attribute_name` (*fdm_t fdm, ferret_string_t ename, int index*) [Function]

Return the name of the *index*-th attribute of the *ename* entity. Note that the attribute list is indexed by 0.

`ferret_status_t fdm_set_entity_attribute_name` (*fdm_t fdm, ferret_string_t ename, ferret_string_t aname, ferret_string_t naname*) [Function]

Rename the attribute *aname* to *naname* on entity *ename*.

`ferret_status_t fdm_add_entity_attribute_group` (*fdm_t fdm, ferret_string_t ename, ferret_string_t gname*) [Function]

Add a new attribute group (see the user's guide for more information about attribute groups) name *gname* to the *ename* entity.

If there is already an attribute group named *gname*, then the function return `FERRET_ERROR`.

`ferret_status_t fdm_remove_entity_attribute_group` (*fdm_t fdm, ferret_string_t ename, ferret_string_t gname*) [Function]

Remove the *gname* attribute group from the *ename* entity.

`ferret_status_t fdm_add_entity_attribute` (*fdm_t fdm, ferret_string_t ename, ferret_string_t aname*) [Function]

Add a new *aname* attribute to the *ename* entity.

If there is already an attribute named *aname*, then `FERRET_ERROR` is returned.

`ferret_status_t fdm_add_entity_attribute_to_group` (*fdm_t fdm, ferret_string_t ename, ferret_string_t aname, ferret_string_t gname*) [Function]

Add the attribute *aname* to the *gname* attribute group in the *ename* entity.

`ferret_status_t fdm_remove_entity_attribute_from_group` (*fdm_t fdm, ferret_string_t ename, ferret_string_t aname, ferret_string_t gname*) [Function]

Remove the attribute *aname* from the *gname* attribute group in the *ename* entity.

`ferret_status_t fdm_remove_entity_attribute` (*fdm_t fdm, ferret_string_t ename, ferret_string_t aname*) [Function]

Remove the *aname* attribute from the *ename* entity.

If the attribute do not exist on the entity, then `FERRET_ERROR` is returned.

`ferret_string_t fdm_get_entity_short_descr` (*fdm_t fdm, ferret_string_t ename*) [Function]

Return the short description of the *ename* entity.

`ferret_status_t fdm_set_entity_short_descr` (*fdm_t fdm, ferret_string_t ename, ferret_string_t descr*) [Function]

Set *descr* as the new short description for the *ename* entity.

`ferret_string_t fdm_get_entity_full_descr` (*fdm_t fdm, ferret_string_t ename*) [Function]

Return the full description of the *ename* entity.

`ferret_status_t fdm_set_entity_full_descr` (*fdm_t fdm, ferret_string_t ename, ferret_string_t descr*) [Function]

Set *descr* as the new full description for the *ename* entity.

`time_t fdm_get_entity_creation_ts` (*fdm_t fdm, ferret_string_t ename*) [Function]

Return the creation time-stamp for the *ename* entity.

`time_t fdm_get_entity_last_modified_ts` (*fdm_t fdm, ferret_string_t ename*) [Function]

Return the last modified time-stamp for the *ename* entity.

`ferret_string_list_t fdm_get_entity_attribute_list` (*fdm_t fdm, ferret_string_t ename*) [Function]

Return a ferret string list with all the ordered attribute names.

`int fdm_get_attribute_relative_order` (*fdm_t fdm, ferret_string_t ename, ferret_string_t aname1, ferret_string_t aname2*) [Function]

With respecto to the *ename* entity, this function return:

- `-1` if the *aname1* attribute is before the *aname2* on the entity attribute list.
- `1` if the *aname1* attribute is after the *aname2* on the entity attribute list.

2.4.2 Relationships

2.4.2.1 Creation and destruction

`ferret_statu_t fdm_create_relationship` (*fdm_t fdm*, *ferret_string_t rname*) [Function]

`fdm_destroy_relationship (fdm_t` *fdm*, *ferret_string_t rname*) [Function]

2.4.3 Class-Subclass relationships

2.4.4 Relations

3 Ferret C Libraries

3.1 Ferret Globals

Some global information about any running Ferret is stored in a global variable named `ferret_globals`, of type `ferret_globals_t`.

`ferret_globals_t` [Data Type]

Structure to store some global information about the running Ferret application. The structure is composed by the following fields:

`Tcl_Interp *master_tcl_interpreter`

The main Tcl interpreter used to build the GUI, read user commands and to execute some Tcl library functions that requires a Tcl interpreter.

– Defined on '`src/ferret_globals.h`'

`ferret_globals` [Global Variable]

Variable of type `ferret_globals_t` that contain some general information about the running Ferret application.

– Defined on '`src/ferret.c`'

3.2 Ferret Error Routines

These routines implement error control and management on Ferret.

`void ferret_panic` (*message*) [Function]

Emit an error message and quit to the operating system with an error code.

This routine never return.

– Prototyped on '`src/ferret_error.h`'

– Defined on '`src/ferret_error.c`'

3.3 Ferret Booleans

Ferret implement an abstract data type to identify boolean values. Its internal representation conform to the C boolean-integer conventions, so Ferret boolean values can (and should) be used directly on C conditional sentences.

`ferret_bool_t` [Data Type]

Can contain either `TRUE` or `FALSE` values.

– Defined on '`src/ferret_bool.h`'

3.4 Ferret Strings

String processing is a very important task in any interactive tool. Definitely, Ferret should support more than the traditional (and english-like languages oriented) ASCII character set. These multilingual functionality should also be portable. The obvious alternative is to use the ISO/IEC 10640 standard, also known as the *Unicode standard*. Altought there exist several implementations of Unicode, Ferret uses the implementation found on the Tcl

library. In that way, both Tcl and C Ferret components can share string data without many conversions.

Since the use by Ferret of the Unicode facilities implemented on the Tcl library could change some day, all other Ferret components access such facilities via an abstract data type: `ferret_string_t`.

`ferret_string_t` [Data Type]
This abstract data type implement a variable-size multilingual string, to be used on all Ferret components that manage strings.

This data type is composed by the following fields:

`Tcl_DString *tcl_string`
A Tcl string as implemented by the Tcl library.

Defined on '`src/ferret_string.h`'

`ferret_string_t ferret_string_new ()` [Function]
Create a new Ferret string and return it. The newly created string is empty.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`void ferret_string_free (`*gstring*`)` [Function]
Destroy *gstring*, that must be a correctly initialized Ferret string, freeing all the used memory.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`int ferret_string_length (`*gstring*`)` [Function]
Return the actual length of *gstring*.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`char* ferret_string_value (`*gstring*`)` [Function]
Return the actual contents of *gstring*, as a pointer to native C characters.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`void ferret_string_truncate (`*gstring, length*`)` [Function]
Truncate *gstring* to the length given by the *length* parameter. If *length* is -1, then no truncation is performed.

Note this routine never allocate memory, so any specified *length* major than the actual string length is ignored.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`void ferret_string_append` (*gstring1, gstring2, length*) [Function]
Append *gstring2* to *gstring1*. If *length* is not −1, then append *gstring2* truncated to *length*.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`void ferret_string_append_string` (*gstring, string, length*) [Function]
Append a character string to a Ferret string. LENGTH characters from STRING are appended. If LENGTH == -1 then all the string is appended.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`ferret_string_t ferret_string_dup` (*gstring*) [Function]
Make a copy of *gstring* and return it as a second Ferret string.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`void ferret_string_set_value` (*gstring, string*) [Function]
Set *string* the new content of *gstring*, destroying any previous contents.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

`ferret_bool_t ferret_string_match` (*gstring, pattern, nocase*) [Function]
Try to match the string *pattern* on *gstring*, and return a boolean value indicating the result of the matching process.

If *nocase* is `TRUE`, then the matching process is case-insensitive.

- Prototyped on '`src/ferret_string.h`'
- Defined on '`src/ferret_string.c`'

3.5 Ferret Regular Expressions

This Ferret library implement support for regular expression matching on Ferret strings.

Like Ferret strings, the actual implementation of the Ferret regexp routines make extensive use of the Tcl regexp engine. Again, this could change on the future (for example, when EDKIT integration), so the API for this routines should be as generic as possible.

The regular expression flavor supported by Ferret is the Tcl one. I consider it is a good thing because i find the Tcl regexp engine as a specially well suited one.

Please note that this routines depend on the master Tcl interpreter running on Ferret, so such interpreter should be correctly constructed.

`ferret_bool_t ferret_regexp_match` (*gstring, pattern*) [Function]
Try to match *gstring* with the regexp *pattern*, and return a boolean value indicating the result of the matching process.

- Prototyped on '`src/ferret_regexp.h`'
- Defined on '`src/ferret_regexp.c`'

3.6 Memory De/Allocation

Ferret implement generic de/allocation routines to be used instead of the plain and system-dependent `malloc` and `free` routines.

It is very encourage to the Ferret Hacker to use these routines instead of the plain ones. These routines are aware of the actual state of the Tcl interpreter and other Ferret specific state.

`void* ferret_alloc` (*size*) [Function]

Allocate *size* bytes and return a pointer to the newly allocated memory.

If there is some trouble (ie. there is not enough memory) a panic error is signaled and Ferret quit.

- Prototyped on '`src/ferret_alloc.h`'
- Defined on '`src/ferret_string.c`'

`void ferret_dealloc` (*pointer*) [Function]

Deallocate the previously allocated memory pointed by *pointer*.

If there is some trouble (ie. the memory pointed was not allocated by `ferret_alloc` a panic error is signaled and Ferret quit.

- Prototyped on '`src/ferret_alloc.h`'
- Defined on '`src/ferret_string.c`'

3.7 Ferret Project

This library manages the memory storage and manipulation of Ferret projects. It features both a C interface and a Tcl interface. The files '`ferret_project_tcl.c`' and '`ferret_project_tcl.h`' contain the Tcl stubs.

A Ferret project is composed by:

- An identifying string, unique on the Ferret environment.
- A title name.
- A file name to store the project, if any.
- A documentation string.
- Information about the author that created the project.
- Version information.
- Zero or more data models. See Section 3.8 [Ferret Datamodel], page 18.
- A domain tree. See Section 3.9 [Ferret Domain Tree], page 18.

3.7.1 Project Constants

3.7.2 Project Data Types

`ferret_author_info_t` [Data Type]

Information related to a Ferret project author.

The field contained on this structure are:

`ferret_string_t author_name`
: The full name of the author.

`ferret_string_t author_email`
: The electronic mail address of the author.

Defined on '`ferret_project.h`'

`ferret_project_id_t` [Data Type]

A string that unequivocally identifies a Ferret project.

Defined on '`src/ferret_project.h`'

`ferret_project_t` [Data Type]

This data type represents a project into the Ferret environment.

It is an structure with the following fields:

`ferret_project_id_t project_id`
: The project's unique string. It is tipically derived from the project's title name.

`ferret_string_t project_name`
: This is the name of the project, as specified by the user.

`ferret_author_info_t author_info`
: Information about the author that created the project.

`ferret_doc_t project_description`
: Documentation for the project.

`ferret_doc_t project_short_description`
: Short documentation for the project.

`ferret_version_t project_version`
: The actual version of the project.

`ferret_domtree_t project_domtree`
: The domain tree of the project.

`ferret_dm_list_t project_dm_list`
: The data models of the project.

`ferret_project_list_t` [Data Type]

This data type represents a list whose elements are Ferret projects. It is a double-linked list.

The fields contained on each list node are:

`ferret_project_t project`
: The project contained into the node.

`struct _ferret_project_list_t next`
: The next node linked on the list.

`struct _ferret_project_list_t previous`
: The previous node linked on the list.

Defined on '`ferret_project.h`'

3.7.3 Project C API

3.7.4 Project Tcl API

3.8 Ferret Datamodel

3.9 Ferret Domain Tree

4 Ferret Tcl Libraries

4.1 The AppTree

Located at '`lib/apptree.tcl`'

This package manages the Application Tree component, that is shown on the left of the Ferret frame:

It is primarily implemented via a Tree megawidget from the BWidget library.

There can be several apptrees opened on the application at a time.

4.1.1 Nodetype structures

The apptree package support several 'nodetypes', described by a structure that must be only manipulated via the apptree API. Each nodetype is displayed in a different way, and with different associated bindings. In that way, the apptree package permits to create several types of nodes, as projects, datamodels, documentation nodes, etc. I hope this to be a sufficiently scalable design in order to support future Ferret development (support more types of nodes).

Each nodetype has some associated attributes:

- A string for identify this type of node. It must begin with an upper case letter, and contain only letters.
- An image to show with nodes of that type.
- A text type indicator, that appears as a prefix on the node text surrounded by parenthesis.

  If the prefix string is empty, then no parentheis are drawn.
- The type of the parent.
- The preferred order into the parent child list.
- It can be a positive number for a concrete order number, or one of the keywords {first} and {last}.
- A primary action callback, that is invoked when the user click on the node (both text and image) with the left button of the mouse. The concrete node name ({*nodetype-instancename*}) is appended to the callback script.

  Note that code for actually selecting the node on the Tree widget is also appending.
- A secondary action callback, that is invoked when the user click on the node (both text and image) with the middle button of the mouse.

  Note that code for actually selecting the node on the Tree widget is also appending.
- A context menu structure, that defines a popup menu that is activated by clicking on the node (both text and image) with the right button of the mouse.

  Example:

```
{ {"New project" project::new}
  {"Open a new project" project::open}
  {Separator}
  {"Close project" project::close} }
```

Note that all the nodetype related information is keeping by the apptree package itself. So every client of the apptree package should use only its exported API.

4.1.2 Node instances

All nodes from the apptree pertain to one nodetype.

Any given node is identified by the string:

```
{NODETYPE-INSTANCENAME}
```

Example:

```
{Project-UnnamedProject}
```

The attributes of an apptree node are:

- A visibility flag.
- An activated flag.

4.1.3 Internal variables

`rw` The Root Widget of the apptree widgets hierarchy.

`treew` The Tree widget path.

`ats` The App Trees Structure, that contain all the logical data of the apptree.

`popts` The packing options of the root widget.

`visible_f`
: This boolean flag reflect the visible state of the apptree. It is initially set to 1.

`ctree` The current tree.

`destroy_page_text`
`next_page_text`
`previous_page_text`
`cbs`
: Associative array containing the code associated with each callback supported by the apptree. See the 'Callbacks' subsection for more information about the available callbacks.

`default_icon`
: It is an image that acts as the default icon.

4.1.4 Callbacks

`cb_destroy_page`
: This callback is called when the user press the "destroy page" button.

  If the callback return 0, then the action is interrupted and the page is not destroyed.

`cb_next_page`
: This callback is called when the user press the "next page" button.

  If the callback return 0, then the action is interrupted and the page is not changed.

`cb_previous_page`
: This callback is called when the user press the "previous page" button.

  If the callback return 0, then the action is interrupted and the page is not changed.

`cb_destroy_node`
: This callback is called when a node is destroyed. The nodename is appended to the callback script.

  If the callback return 0, then the action is interrupted and the node is not destroyed.

4.1.5 Managing the infrastructure (widgets)

`apptree::init` *WIDGET POPT* [Function]
: Put the apptree infrastructure (the several widgets) on WIDGET, with POPT packing options.

`apptree::fini` [Function]
: Destroy the apptree infrastructure, freeing all memory, and unpacking the several widgets.

`apptree::visible` *BOOLEAN* [Function]
: Show/Hide the apptree widget, depending of the value of BOOLEAN.

`apptree::toggle_visible` [Function]
: Toggle the visibility of the apptree widget.

4.1.6 Button texts management

`apptree::set_destroy_page_text` *TEXT* [Function]
: Set TEXT as the text displayed on the 'destroy page' button.

`apptree::set_next_page_text` *TEXT* [Function]
: Set TEXT as the text displayed on the 'next page' button.

`apptree::set_previous_page_text` [Function]
: Set TEXT as the text displayed on the 'previous page' button.

`apptree::set_default_icon` *IMAGE* [Function]
: Set IMAGE as the default image for nodetypes.

4.1.7 Callbacks management

`apptree::install_callback` *CBNAME SCRIPT* [Function]
: Install SCRIPT as the callback named CBNAME.

`apptree::uninstall_callback` *CBNAME* [Function]
: Remove the CBNAME callback.

4.1.8 ATS management

`apptree::ats_insert_tree` *TREENAME* [Function]
Add TREENAME to the App Tree Structure.

Initially:

- Both active and visible flag are set to 1.
- There are not nodetypes.
- There are not nodes.

`apptree::ats_remove_tree` *TREENAME* [Function]
Delete all information of TREENAME from the App Tree Structure.

`apptree::ats_get_next_tree` *TREENAME* [Function]
Return the name of the tree that follows TREENAME.

This emulates a loop list.

`apptree::ats_get_previous_tree` *TREENAME* [Function]
Return the name of the tree wich TREENAME follow.

`apptree::ats_get_tree` *TREENAME* [Function]
Return the structure of TREENAME.

`apptree::ats_set_tree` *TREENAME NEWTREE* [Function]
Set NEWTREE as the new structure for TREENAME.

`apptree::ats_get_tree_activeflag` *TREENAME* [Function]
Return the value of the active flag of TREENAME.

`apptree::ats_set_tree_activeflag` *TREENAME VALUE* [Function]
Set VALUE as the new active flag for TREENAME.

`apptree::ats_get_tree_visibleflag` *TREENAME* [Function]
Return the value of the visible flag of TREENAME.

`apptree::ats_set_tree_visibleflag` *TREENAME VALUE* [Function]
Set VALUE as the new visible flag for TREENAME.

`apptree::ats_get_tree_nodetypes` *TREENAME* [Function]
Return the nodetypes structures of TREENAME.

`apptree::ats_set_tree_nodetypes` *TREENAME NODETYPES* [Function]
Set NODETYPES as the new TREENAMEs nodetypes structure.

`apptree::ats_get_tree_nodes` *TREENAME* [Function]
Return the nodes structure of TREENAME.

`apptree::ats_set_tree_nodes` *TREENAME NODES* [Function]
Set NODES as the new TREENAMEs nodes structure.

`apptree::ats_get_tree_nodetype` *TREENAME NODETYPE* [Function]
Return the structure of NODETYPE from TREENAME.

`apptree::ats_set_tree_nodetype` *TREENAME NODETYPE NNT* [Function]
Set NNT as the struct of NODETYPE on TREENAME.

`apptree::ats_get_tree_nodetype_image` *TREENAME NODETYPE* [Function]
Return the image associated with a nodetype on the current tree.

`apptree::ats_set_tree_nodetype_image` *TREENAME NODETYPE IMAGE* [Function]
Set IMAGE as the new image for NODETYPE on TREENAME.

`apptree::ats_get_tree_nodetype_text_prefix` *TREENAME NODETYPE* [Function]
Return the text prefix associated with NODETYPE on TREENAME.

`apptree::ats_set_tree_nodetype_text_prefix` *TREENAME NODETYPE TEXT* [Function]
Set TEXT as the text prefix associated with NODETYPE on TREENAME.

`apptree::ats_get_tree_nodetype_parent` *TREENAME NODETYPE* [Function]
Return the nodetype parent associated with NODETYPE on TREENAME.

`apptree::ats_set_tree_nodetype_parent` *TREENAME NODETYPE PARENT* [Function]
Set PARENT as the nodetype parent associated with NODETYPE on TREENAME.

`apptree::ats_get_tree_nodetype_order` *TREENAME NODETYPE* [Function]
Return the preferred order associated with NODETYPE on TREENAME.

`apptree::ats_set_tree_nodetype_order` *TREENAME NODETYPE ORDER* [Function]
Set ORDER as the new order associated with NODETYPE on TREENAME.

`apptree::ats_get_tree_nodetype_actions` *TREENAME NODETYPE* [Function]
Return the actions structure associated with NODETYPE on TREENAME.

`apptree::ats_set_tree_nodetype_actions` *TREENAME NODETYPE ACTIONS* [Function]
Set ACTIONS as the associated structure with NODETYPE on TREENAME.

`apptree::ats_get_tree_nodetype_primary_action` *TREENAME NODETYPE* [Function]
Return the primary action of NODETYPE on TREENAME.

`apptree::ats_set_tree_nodetype_primary_action` *TREENAME NODETYPE ACTION* [Function]
Set ACTION as the primary action of NODETYPE on TREENAME.

`apptree::ats_get_tree_nodetype_secondary_action` *TREENAME NODETYPE* [Function]
Return the secondary action of NODETYPE on TREENAME.

`apptree::ats_set_tree_nodetype_secondary_action` *TREENAME NODETYPE ACTION* [Function]

Set ACTION as the secondary action of NODETYPE on TREENAME.

`apptree::ats_get_tree_nodetype_contextmenu` *TREENAME NODETYPE* [Function]

Return the context menu structure associated with NODETYPE on TREENAME.

`apptree::ats_set_tree_nodetype_contextmenu` *TREENAME NODETYPE MENU* [Function]

Set MENU as the menu context structure associated with NODETYPE on TREENAME.

`apptree::ats_add_tree_nodetype_contextmenu_entry` [Function]
TREENAME NODETYPE INDEX ENTRY

Set ENTRY as the INDEXth entry on the context menu associated with NODETYPE, on TREENAME.

`apptree::ats_remove_tree_nodetype_contextmenu_entry` [Function]
TREENAME NODETYPE INDEX

Remove the INDEXth entry from the context menu associated with NODETYPE, on TREENAME.

`apptree::ats_get_tree_node` *TREENAME NODENAME* [Function]

Return the structure of NODENAME on TREENAME.

`apptree::ats_set_tree_node` *TREENAME NODENAME NODE* [Function]

Set NODE as the new structure of NODENAME on TREENAME.

`apptree::ats_get_tree_node_type` *TREENAME NODENAME* [Function]

Return the type associated with NODENAME on TREENAME.

`apptree::ats_set_tree_node_type` *TREENAME NODENAME NODETYPE* [Function]

Set NODETYPE as the type associated with NODENAME on TREENAME.

`apptree::ats_get_tree_node_flags` *TREENAME NODENAME* [Function]

Return the flags structure associated with NODENAME on TREENAME.

`apptree::ats_set_tree_node_flags` *TREENAME NODENAME FLAGS* [Function]

Set FLAGS as the flags structure for NODENAME on TREENAME.

`apptree::ats_get_tree_node_activeflag` *TREENAME NODENAME* [Function]

Return the value of the active flag of NODENAME on TREENAME.

`apptree::ats_set_tree_node_activeflag` *TREENAME NODENAME VALUE* [Function]

Set VALUE as the value of the active flag of NODENAME on TREENAME.

`apptree::ats_get_tree_node_visibleflag` *TREENAME NODENAME* [Function]
Return the value of the visible flag of NODENAME on TREENAME.

`apptree::ats_set_tree_node_visibleflag` *TREENAME NODENAME VALUE* [Function]
Set VALUE as the visible flag associated with NODENAME on TREENAME.

`apptree::ats_get_tree_node_parent` *TREENAME NODENAME* [Function]
Return the nodename of the parent of NODENAME, on TREENAME.

`apptree::ats_set_tree_node_parent` *TREENAME NODENAME PARENT* [Function]
Set PARENT as the new parent for NODENAME on TREENAME.

`apptree::ats_remove_tree_node` *TREENAME NODENAME* [Function]
Remove the node NODENAME from TREENAME.

`apptree::ats_get_tree_nodetype_removableflag` *TREENAME NODETYPE* [Function]
Return the value of the removable flag from NODETYPE on TREENAME.

`apptree::ats_set_tree_nodetype_removableflag` *TREENAME NODETYPE VALUE* [Function]
Set the value of the removable flag of NODETYPE on TREENAME.

4.1.9 Managing single trees

`apptree::selection` [Function]
Return the selected node name.

`apptree::tree_exist` *TREENAME* [Function]
Return 1 if there is an apptree named TREENAME. Return 0 else.

`apptree::create_tree` *TREENAME* [Function]
Create a new apptree named TREENAME. It is appended at the end of the notebook.
Return 0 if there is already an apptree called TREENAME. Return 1 else.

`apptree::destroy_tree` *TREENAME* [Function]
Destroy the apptree named TREENAME.
Return 0 if TREENAME do not exist on the apptree. Return 1 else.

`apptree::tree_visible` *TREENAME BOOL* [Function]
Modify the 'visible' flag for TREENAME.

`apptree::tree_active` *TREENAME BOOL* [Function]
Modify the 'active' flag for TREENAME.
Note that must be at least one active tree.

`apptree::set_current_tree` *TREENAME* [Function]
Set TREENAME as the current apptree.
All the following API operates over the current apptree.

`apptree::current_tree` [Function]
Return the 'treename' of the current apptree.

4.1.10 Un/Registering node types

`apptree::nodetype_exist` *NODETYPE* [Function]
Return 1 if there is already a node type called NODETYPE on the current tree.

`apptree::register_nodetype` *ID* [Function]
Register ID as a new nodetype supported by the tree.

The defaults for nodetypes are:

- The image is 'default_icon'
- The text prefix is empty: `{}`
- The parent nodetype is 'root_type'
- The preferred order is 'last'
- There is not a primary action: `{}`
- There is not a secondary action: `{}`
- The context menu is empty: `{}`

`apptree::unregister_nodetype` *ID* [Function]
Removes the registration of the nodetype ID.

4.1.11 Managing images and text prefixes

`apptree::set_nodetype_image` *ID IMAGE* [Function]
Set IMAGE as the image to be displayed on nodes of type ID.

`apptree::get_nodetype_image` *ID* [Function]
Return the image associated with nodetype ID.

`apptree::set_nodetype_text_prefix` *ID STRING* [Function]
Set STRING as the text prefix to be displayed on nodes of type ID.

`apptree::get_nodetype_text_prefix` *ID* [Function]
Return the text prefix associated with nodetype ID.

4.1.12 Managing tree position information

`apptree::get_nodetype_parent` *NODETYPE* [Function]
Return the parent nodetype of NODETYPE.

`apptree::set_nodetype_parent` *NODETYPE PARENT* [Function]
Set PARENT as the parent nodetype of NODETYPE.

`apptree::get_nodetype_order` *NODETYPE* [Function]
Return the order of NODETYPE.

`apptree::set_nodetype_order` *NODETYPE ORDER* [Function]
Set ORDER as the new order for NODETYPE.

ORDER must be a positive integer (starting at 0), or the keywords 'first', 'last'.

4.1.13 Managing nodetypes flags

`apptree::nodetype_is_removable` *NODETYPE VALUE* [Function]
Set the removable flag for NODETYPE.

4.1.14 Managing actions (primary and secondary ones)

`apptree::set_nodetype_primary_action` *NODETYPE CALLBACK* [Function]
Set CALLBACK as the script to call when the user click on the node with <Button-1>.

The node name is appended to CALLBACK before the evaluation.

`apptree::get_nodetype_primary_action` *NODETYPE* [Function]
Return the primary action of NODETYPE.

`apptree::set_nodetype_secondary_action` *NODETYPE CALLBACK* [Function]
Set CALLBACK as the script to call when the user click on the node with <Button-2>.

The node name is appended to CALLBACK before the evaluation.

`apptree::get_nodetype_secondary_action` *NODETYPE* [Function]
Return the secondary action associated with NODETYPE.

4.1.15 Managing context menus

`apptree::add_entry_to_nodetype_contextmenu` *NODETYPE INDEX LABEL CALLBACK* [Function]
Add a new entry to the context menu for NODETYPE.

Where INDEX can be a numerical index, or the keywords {first} and {last}.

Both CALLBACK script and LABEL strings are preprocessed, with some substitutions:

- *%X* is substituted by the mouse x coordinate.
- *%Y* is substituted by the mouse y coordinate.
- *%NODETYPE* is substituted by the node type.
- *%NODENAME* is substituted by the node name.

`apptree::remove_entry_from_nodetype_contextmenu` *NODETYPE INDEX* [Function]
Delete an entry from the context menu for NODETYPE.

Where INDEX can be a numerical index, or the keywords {first} and {last}.

4.1.16 Creating and deleting node instances

`apptree::node_exis` *NODENAME* [Function]
Return 1 if NODENAME exist on the current tree. Return 0 else.

`apptree::instance_node` *NODETYPE NODENAME* [Function]
Show a new node on the current tree, of type NODETYPE, identified by the string NODENAME.

Initially, both the visibility and activation flags are set to 1.

`apptree::remove_node` *NODENAME* [Function]
Removes NODENAME from the tree.

`apptree::node_visible` *NODETYPE NODENAME BOOLEAN* [Function]
Set the visible state of NODENAME

`apptree::node_active` *NODETYPE NODENAME BOOLEAN* [Function]
Set the active state of NODENAME

4.2 The Diagram Package

The Diagram package is a pure Tcl implementation that uses the Tk canvas in order to implement interactive diagrams.

4.2.1 Diagram Overview

The Tk canvas allow to construct many kind of structured graphics. That means that almost every kind of user interface can be constructed using the Tk canvas. Ferret is mainly a graphical editor, with well defined edited objects: entities, relationships, etc. Each of these objects usually need a graphical representation with which the user interact.

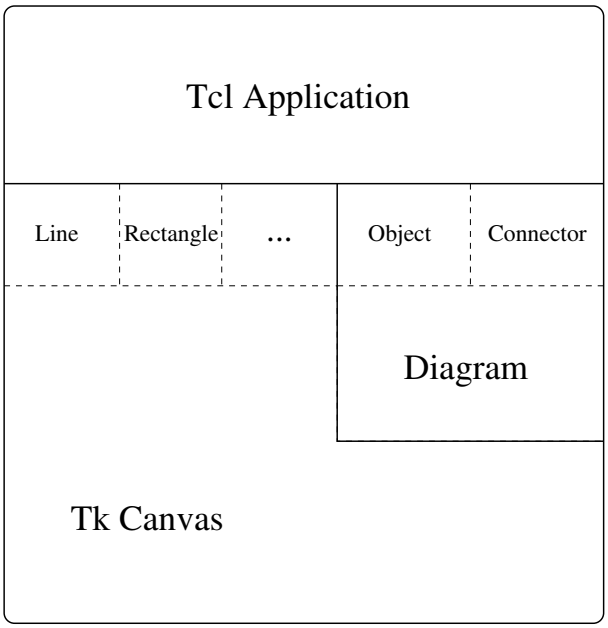

Figure 4.1: Diagram Package Architecture

A diagram is composed by two main component types:

– Objects with shape.
– Connectors that involves one or more objects.

4.2.2 Diagram manipulation

`diagram::create` *DNAME WIDGET* [Function]
Creates a new diagram named DNAME on WIDGET.

`diagram::destroy` *DNAME* [Function]
Destroy DNAME.

`diagram::pack` *DNAME* [Function]
Pack the tk widgets of DNAME.

`diagram::unpack` *DNAME* [Function]
Unpack the tk widgets of DNAME.

`diagram::get_object_list` *DNAME* [Function]
Return a list with the names of all the objects present on DNAME.

`diagram::get_connector_list` *DNAME* [Function]
Return a list with the names of all the connectors present on DNAME.

`diagram::print_ps` *DNAME FILENAME PAPERTYPE COLORMODE PAGEANCHOR* [Function]
Where,

colormode color or gray

papertype fit or a4

4.2.3 Object manipulation

`diagram::exist_object` *DNAME ONAME* [Function]
Return 1 if ONAME is an existing object into DNAME.

`diagram::set_object` *DNAME ONAME NEWOBJ* [Function]
Set NEWOBJ as the new object structure for ONAME on DNAME.

`diagram::get_object` *DNAME ONAME* [Function]
Return the object structure for ONAME on DNAME.

`diagram::create_object` *DNAME ONAME OTYPE LOCATION* [Function]
Creates a new object on the diagram named ONAME, of type OTYPE at the point LOCATION.

`diagram::remove_object` *DNAME ONAME* [Function]
Remove ONAME from the DNAME diagram.

`diagram::set_object_location` *DNAME ONAME LOCATION* [Function]
Set LOCATION as the new location point of ONAME on DNAME.

`diagram::get_object_location` *DNAME ONAME* [Function]
Return the location of ONAME on DNAME.

`diagram::set_object_window` *DNAME ONAME WINDOW* [Function]
Set WINDOW as the new window for ONAME on DNAME.

`diagram::get_object_window` *DNAME ONAME* [Function]
Get the window from ONAME on DNAME.

`diagram::get_object_type` *DNAME ONAME* [Function]
Return the type of ONAME in DNAME.

`diagram::set_object_attribute` *DNAME ONAME ANAME VALUE* [Function]
Set VALUE as the new value for ANAME on ONAME in DNAME.

`diagram::get_object_attribute` *DNAME ONAME ANAME* [Function]
Return the value of ANAME from ONAME in DNAME.

`diagram::set_object_attributes` *DNAME ONAME ATTRIBUTES* [Function]
Set ATTRIBUTES as the new attribute set for ONAME on DNAME.

`diagram::get_object_attributes` *DNAME ONAME* [Function]
Return the attributes structure from ONAME on DNAME.

`diagram::get_object_connectors` *DNAME ONAME* [Function]
Return a list with the name of all the connectors that leads to ONAME on DNAME.

`diagram::set_object_connectors` *DNAME ONAME CNNS* [Function]
Set CNNS as the new connector list for OBJECT.

`diagram::object_connector_exist` *DNAME ONAME CNN* [Function]
Return 1 if CNN is on the ONAMEs connector list. Return 0 else.

`diagram::add_connector_to_object` *DNAME ONAME CNN* [Function]
Add CNN to the ONAMEs connector list.

`diagram::remove_connector_from_object` *DNAME ONAME CNN* [Function]
Remove CNN from the ONAMEs connector list.

`diagram::update_object` *DNAME ONAME* [Function]
Update the geometry and the connectors of ONAME.

4.2.4 Port manipulation

`diagram::create_port` *POINT ORIENT* [Function]
Create a new port.

`diagram::get_port_point` *PORT* [Function]

`diagram::set_port_point` *PORT POINT* [Function]

`diagram::get_port_orient` *PORT* [Function]

`diagram::set_port_orient` *PORT ORIENT* [Function]

4.2.5 Connector manipulation

`diagram::get_connector` *DNAME CNAME* [Function]
Return the structure for CNAME on DNAME.

`diagram::set_connector` *DNAME CNAME CNN* [Function]
Set CNN as the new structure for CNAME on DNAME.

`diagram::exist_connector` *DNAME CNAME* [Function]
Return 1 if CNAME is in DNAME. Return 0 else.

`diagram::get_connector_obj1` *DNAME CNAME* [Function]

`diagram::set_connector_obj1` *DNAME CNAME ONAME* [Function]

`diagram::get_connector_port1` *DNAME CNAME* [Function]

`diagram::set_connector_port1` *DNAME CNAME PORT* [Function]

`diagram::get_connector_obj2` *DNAME CNAME* [Function]

`diagram::set_connector_obj2` *DNAME CNAME ONAME* [Function]

`diagram::get_connector_port2` *DNAME CNAME* [Function]

`diagram::set_connector_port2` *DNAME CNAME PORT* [Function]

`diagram::get_connector_drawproc` *DNAME CNAME* [Function]

`diagram::set_connector_drawproc` *DNAME CNAME DRAWPROC* [Function]

`diagram::get_connector_cp` *DNAME CNAME* [Function]

`diagram::set_connector_cp` *DNAME CNAME CP* [Function]

`diagram::create_connector` *DNAME CNAME OBJ1 OBJ2 DRAW_PROC* [Function]
Create a new connector named CNAME that connect OBJ1 and OBJ2, with DRAW_PROC as the custom drawing procedure.

`diagram::update_connector` *DNAME CNAME* [Function]
Update the geometry and paint CNAME.

`diagram::remove_connector` *DNAME CNAME* [Function]
Remove the connector named CNAME from DNAME.

Global variable index

Data type index

Function index

(

*

A

D

F