OCEAN GEHR-compliant Kernel

Application Programmer's Interface

Author: Thomas Beale

Revision: 2.1 Draft A

Pages: 33

Amendment Record

Issue	Details	Who	Date
1.1 Draft A	Initial Writing	T Beale	Oct 1999
1.1 Draft B	Standalone document	T Beale	Nov 1999
2.1 Draft A	API for content level added; examples included.	T Beale	17 May 2000

1 Introduction

1.1 Status

This document constitutes the beginnings of an API description. At this stage, it should be read by application developers, and their feedback used to determine the flavour of the interface (e.g. use of logical handles etc). When the rules of the API have become firm, the remainder will be completed fairly quickly.

1.2 Overview

If we consider that the GEHR kernel as a constructor and processor of the informational structures of EHRs, the kernel API is the means of controlling it from the outside. In object-oreinted terms, the API is nothing more than the interface to an outer layer of classes whose job is to provide an interface to the underlying model, allowing procedural usage. These classes are sometimes called *policy* classes, because they represent a certain way of using the underlying model. In more traditional programming terms, the exported interface of these classes consititutes the application programmer's interface, and this is a perfectly reasonable view to take, especially when using more procedural languages.

There are a number of issues with the design of the exported interface of the kernel. The primary issue is that if the kernel is to act as a component, and perform its functions consistently, regardless of details of the caller or the mechanism of runtime communication with applications or other components, then there are *constraints on the semantics of the function in the API*. Additionally, the design of the functions has as much to do with the *requirements of calling applications* as the underlying kernel.

1.2.1 Types of API

Starting from the GOM, there are a number of layers of classes which are potentially needed for different purposes. The first layer of classes can be thought of as a "native" object-oriented interface: they represent the interface that would be used to the kernel for an application being written in the same language, or a very similar one. The calls and structure of this API can retain most of the features of Eiffel, thus its main purpose is to provide calls reflecting logical patterns of usage of GOM classes. A native interface allows us to write applications in Eiffel.

From the native API, we can fairly easily provide a *standardised native* interface, using CORBA. A mapping already exists from Eiffel to CORBA, allowing the native interface to be made available to languages which can use the object-oriented style interface provided by a CORBA ORB. With such an interface, can write applications in C++ and Java. This logic may also apply to Enterprise Java Beans; more investigation is required on this.

Moving further away from the strong object-oriented paradigm of Eiffel, Java and friends, we come to languages such as Delphi and VB, which have a modicum of object-orientation. To write an application in these languages, we need to provide yet another interface layer which makes the kernel available across the COM interface they typically use, with a concomitant simiplification of types and other semantics. We will call the API of this layer the *component* API. Applications written to this API are likely to be end-user GUI applications.

The following table summarises the characteristics of the three levels of API described above in terms of the object-oriented concepts supported.

OO Concept	Native API (Eiffel)	Standardised Native API (IDL)	Component API (COM)
simple data types	Y	Y	Y
arrays of simple types	Y	Y	Y
constructed types	Y	Y	Y
generic classes	Y	containers only	-
inheritance	Y	Y	-
assertions	Y	-	-
exceptions	Y	Y	Y

Where the concept is not supported directly does not mean it cannot be implemented, it simply means that some mapping or explicit work is required to make it available.

2 Working Example

In order to give some substance to the following discussion, we will posit a hypothetical example of API usage. The following is not in any way definitive of the kernel API - it will just serve the needs of elucidating the issues of API construction. The example performs the simple logical task of initialising a session with a GEHR kernel.

At the native API, the first thing we want to do is to make a call to a function like:

```
KERNEL_SESSION.init_kernel_session(a_user:STAFF_MEMBER; an_hcf:HCF)
        require
              User_exists: a_user /= Void
              Hcf_exists: an_hcf /= Void
```

This is a notional "logging in" function in a KERNEL_SESSION object, which we assume is in the kernel API. To get this far, we need to have created STAFF_MEMBER and HCF objects as arguments. One way to do this might have been to pass both in the form of XML or other parseable strings; this would require an alternative function of the form:

```
KERNEL_SESSION

feature --
        init_kernel_session_from_doc(a_user:XML_DOC; an_hcf:XML_DOC)
              require
                    User_exists: a_user /= Void and then not a_user.empty
                    Hcf_exists: an_hcf /= Void and then not an_hcf.empty
```

Here we assume that XML_DOC is a descendant of the Eiffel STRING class. This function is entirely legitimate, and is likely to be used in the kernel. However, it may not be possible or suitable in all circumstances. What if the application is designed to build a STAFF_MEMBER and an HCF piece-by-piece, and pass them as arguments to the kernel session function? In the underlying GOM, the class STAFF_MEMBER has two make functions with the signatures:

```
make(a_name:PERSON_NAME_IMPL; a_position:PLAIN_TEXT; dob:DATE_IMPL; pob:STRING)
        require
              Person_name_exists: a_name /= Void
              Position_exists: a_position /= Void
              Dob_exists: dob /= Void
              Pob_exists: pob /= Void and then not pob.empty


make_all(a_name:PERSON_NAME_IMPL; a_position:PLAIN_TEXT;
          an_aliases:HASH_TABLE[PARTY_NAME_IMPL, TERM_TEXT];
          dob:DATE_IMPL; pob:STRING; a_contacts:ARRAYED_LIST [CONTACT_DESCRIPTOR])
        require
              Person_name_exists: a_name /= Void
              Position_exists: a_position /= Void
              Aliases_exists: an_aliases /= Void
              Dob_exists: dob /= Void
              Pob_exists: pob /= Void and then not pob.empty
              Contacts_exists: a_contacts /= Void
```

The first of these is a minimal constructor, while the second is designed for use when a complete staff member object is imported from elsewhere. Similarly, the HCF class has a make function:

```
make(a_name:PARTY_NAME_IMPL; a_business_address:DEFINITION_CONTENT;
a_reg:REGISTRATION)
        require
```

```
            Name_exists: a_name /= Void
            Address_exists: a_business_address /= Void
            Registration_exists: a_reg /= Void
```

In the native API, we might provide features like the following (classes in this API are prefixed for the moment with NAPI):

```
class NAPI_DEMOGRAPHIC_FACTORY

feature
    create_hcf(hcf_name:STRING; business_address:DEFINITION_CONTENT;
            reg_country, reg_body, reg_id:STRING):HCF
        require
            Hcf_name_exists: hcf_name /= Void and then not hcf_name.empty
            Address_exists: business_address /= Void
            Reg_country_exists: reg_country /= Void and then not reg_country.empty
            Reg_body_exists: reg_body /= Void and then not reg_body.empty
            Reg_id_exists: reg_id /= Void and then not reg_id.empty

    create_staff_member(
            a_name:PERSON_NAME_IMPL; a_position:PLAIN_TEXT;
            dob:DATE_IMPL; pob:STRING)
        require
            Hcf_name_exists: hcf_name /= Void and then not hcf_name.empty
            Address_exists: business_address /= Void

    create_address(tags:ARRAY[STRING];
            values:ARRAY[STRING]):DEFINITION_CONTENT is
        require
            Tags_exists: tags /= Void and then tags.count > 0
            Values_exists: values /= Void and then values.count > 0
            Data_validity: tags.count = values.count
```

These functions are not suitable to be directly exposed in the component API, due to the constructed Eiffel types, so some logical equivalent is needed. For the moment, we will assume component API functions of the form:

```
class CAPI_DEMOGRAPHIC_FACTORY

feature --
       create_staff_member(... args ....)

       create_hcf(... args ....)
```

The arguments to these functions (yet to be defined) carry the same logical information as those of the underlying `make` functions. When either of these functions is called, they will call the appropriate underlying `make` function, above. To do this, they will create objects of type `PERSON_NAME_IMPL`, `PLAIN_TEXT`, and `DEFINITION_CONTENT`, based on the arguments. How will this be done? Let us take the example of the business address argument. The kernel uses the `DEFINITION_CONTENT` generic type for such things because it does not want to use a concrete class (such as "BUSINESS_ADDRESS") for demographic entities. But how is a `DEFINITION_CONTENT` created? Let us assume that for the example, we want to create a `DEFINITION_CONTENT` whose logical form is "list", and that the contents will be tagged strings. Thus, for the logical address:

```
73 High St,
Kingston,
Norfolk
ABC123
England
```

we intend to create a structure of the form:

```
PC:content-root
        |
        +---->HP/HG: “business address”
                |
                +---->HV: “street number” = 73
                |
                +---->HV: “street name” = “High St”
                |
                +---->HV: “locality” = “Kingston”
                |
                +---->HV: “county” = “Norfolk”
                |
                +---->HV: “postcode” = “ABC123”
                |
                +---->HV: “country” = “England”

(where PC = DEFINITION_CONTENT; HP = HIERARCHICAL_PROPOSITION; HG =
HIERARCHICAL_GROUP, HV = HIERARCHICAL_VALUE)
```

To start with there is a `make` function in the `DEFINITION_CONTENT` class in the GOM:

```
make(a_form:INTEGER; a_model:TERM_TEXT)
```

This will create an empty content object. Two further problems arise: how to create the `TERM_TEXT` argument, and how to populate the content object with data representing the address? We will deal with the second of these first. Based on the GOM, we want to be able to make calls to the API which eventually translate to calls to certain calls to the relevant kernel objects perhaps similar to the following:

```
local
        i:INTEGER
        tags, values:ARRAY[STRING]
        a_term:TERM_TEXT
        a_tag, a_val:PLAIN_TEXT
        a_content_root:DEFINITION_CONTENT
        a_content:LIST_PROPOSITION[NO_CONTEXT]
do
        tags := <<"street number", "street name", "locality", "county",
              "postcode", "country">>
        values := <<"73",           "High St",     "Kingston", "Norfolk",
              "ABC 123",  "England">>
        create a_term.make(“business address”, Gehr_termset)
        create a_content_root.make(Form_list, a_term)
        a_content := my_pure_content.content
        from
              i := 1
        until
              i > tags.count
        loop
              create a_tag.make(tags.item(i))
              create a_val.make(values.item(i))
```

```
                a_content.put_value(a_tag,a_value)
                i := i + 1
            end
        end
```

The above is somewhat contrived, but illustrative of certain problems of the component API: how are progressively larger objects to be created, remembered, passed to higher level creation functions and so on, when the caller is on the other side of a component interface, and quite likely written in a completely different language from the kernel? Not to mention that the application developer does not want to write torturous, bulky code to perform even the simplest of tasks.

3 API Design

It is a primary aim for the GEHR kernel to be able to act as a component which is usable by applications written in other languages, and regardless of the method of building it into the runtime system. Depending on the semantics of the language, and its preferred method of binary integration, either the Native API or the Component API will be used. The former is developed from the GOM, while the latter is a mapping of the Native API, suitable for use by COM clients. The following sections discuss the requirements and design constraints of each type of API, in order to develop a set of rules for writing APIs.

3.1 Native API Requirements

The native API is designed for use by applications written in object-oriented languages. The mode of usage is slightly different depending on the language:

- Eiffel applications can use it directly
- C and C++ applications must use the CECIL interface, i.e. the C mappings of the Eiffel API
- Java and C++ applications can use the CORBA mapping of the native API

In the interests of enabling the use of as many languages as possible, the API takes into account a number of constraints, including:

- Inheritance may be limited to single inheritance
- No genericity (i.e. no template classes)
- Different idioms for programming
- Exception handling is typically different
- Inability to include pre- post-conditions in the natural form available in Eiffel.

To Be Determined:

3.1.1 Inheritance

The simplest approach with respect to inheritance is simply not to use it in the API. Whilst it serves an important purpose in the GOM, there is no strong need for it in the API, since only concrete classes need to be exposed in the API.

3.1.2 Genericity

Genericity in Eiffel is a means of generating types from classes which have arguments. For example, the class `LINKED_LIST` can take as its generic parameter another type. As a result, types such as `LINKED_LIST[INTEGER]`, `LINKED_LIST[STRING]`, `LINKED_LIST[PERSON]`, and `LINKED_LIST[ARRAYED_LIST[ITEM]]` can be generated. Of mainstream object-oriented languages, only C++ has the same facility. However, for languages using the CORBA interface, linear containers such as `LINKED_LIST` can be used in a limited sense, since they have equivalents in IDL.

IDL has the pseudo-types `List<>`, `Bag<>`, `Array<>` and `Set<>`, each with different semantics. The simplest way to use these is to map all Eiffel `LIST` types to `List<>`, `ARRAY[G]` to `Array<>` and so on.

The remaining problem for generic types in the GOM is for non-linear containers, of which, as it happens, there are very few. However these can be mapped to their equivalent types. For example the Eiffel generic type `EXTERNAL_REF[HCP]` can be mapped to the type `EXTERNAL_REF_HCP`.

3.1.3 Assertions

One of the most powerful aspects of Eiffel is its ability to express contracts, using assertions. The GOM exploits these extensively, making it far more reliable and comprehensible than if it had been expressed in a weaker language.

In a kernel which is implemented in Eiffel, the assertions are of course available at runtime. In a production kernel, it is most likely that pre-conditions would be left activated, meaning that violations of preconditions cause an assertion violation exception, rather than some obscure result or exception later in processing. All assertions - pre-conditions, post-conditions and invariants - will always be available in documentation to API developers as well, enabling (if not guaranteeing) the same level of quality.

The question remains of how to represent assertions (if at all) in the API. The main assertion of importance to the client is the precondition: it both expresses the condition the client must satisfy, and provides functions which the client can call at runtime to determine whether the main call can be made. This enables the client to avoid causing an exception in the system due to a condition which was in fact knowable in advance (i.e. by the client testing it first), and hence to write more robust software.

It makes sense therefore to supply all preconditions in the API as standalone functions, to which the client can pass arguments, in order to determine their validity for passing to the main function. The client is not obliged to use these functions; their use is recommended in parts of the client code where the quality of arguments may not be guaranteed (e.g. unknown user input, data from a database etc). In formal terms, this is done as follows:

For an Eiffel feature of the form:

```
class CLASS_1

feat_1(....args...):RESULT
        require
                condition_1: some boolean condition
                condition_2: some boolean condition
        do
                -- work...
        end
```

The API will contain functions of the form:

```
class_1_feat_1(.... args ...):RESULT
class_1_feat_1_condition_1( ..... args ....):BOOLEAN
class_1_feat_1_condition_2( ..... args ....):BOOLEAN
```

As a simplification, numerous argument-testing conditions could be rolled into one, as follows.

```
class_1_feat_1(.... args ...):RESULT
class_1_feat_1_args_valid( ..... args ....):BOOLEAN
```

The exact mapping of Eiffel precondition functions to the API is a matter of ensuring clarity. However, condition tests for which the client might reasonably want to do something separate in the case of failure should remain separate functions in the API.

For certain arguments there may be special preconditions, for example, for XML arguments, there may be a precondition which tests validity of the passed string as XML.

3.1.4 Exceptions

One last issue relating to assertions needs to be addressed: what happens if there is an assertion violation in the kernel? This brings us to the subject of exception handling.

The Eiffel runtime system will generate an exception, in a number of circumstances:

- Assertion violation
- Divide-by-zero
- Operating system errors such as segmentation fault, illegal address, etc
- Programmed software exception

By default, an Eiffel system will die if an exception is not handled. There are two possibilities to avoid such undesirable behaviour:

- Handle the exception in the kernel, using the Eiffel **rescue**/**retry** mechanism. A **rescue** clause is written for a routine which assesses the exception and may try to fix the problem and then use the **retry** instruction in order to re-enter the main routine (at the top).
- In the case of an Eiffel component integrated into an application, if there is no rescue clause, the main runtime exception handler is the application. It is the component's job to pass back sufficient exception information to enable the client to deal with it sensibly.

To Be Determined: what is the actual behaviour of exceptions in an eiffel system where the eiffel part is a component? If the exception is passed to the client, does the eiffel runtime still remember that there is an outstanding exception?

There are a number of questions to be answered here:

- Which conditions should be handled internally in the kernel?
- Even for an internally handled exception, what happens if it continues to fail, i.e. the rescue/retry fails to establish the conditions for correct execution?
- What should the client do with exceptions which are passed to it?

Handling the exception may well involve calling clean-up functions in the kernel, which would need to be supplied in the API. There may be a need for high-level "reset" functions, which can put the kernel into a guaranteed correct, known state.

At a minimum, the `default_rescue` clause of a class such as `KERNEL_SESSION` should post the name and type of any exception in a form which can be read by the calling application. Various information about exceptions in Eiffel can be obtained from the `EXCEPTIONS` class.

To Be Determined: more research required.

3.1.5 Example

To Be Determined:

3.2 Component API Requirements

Usage of the kernel as a binary component, e.g. via Microsoft's Component Object Model (COM) adds further constraints, including:

- Languages may not be object-oriented at all, or may have very different concepts of class, method, and attribute.

- In general, pointers or references to objects created by the kernel will have no meaning to the client, since return values are passed outside the kernel execution context (i.e. process).
- Certain data types cannot be handled by COM

To Be Determined: what are these

In fact, these constraints are not necessarily disadvantages, since the concerns of user application development are different from those of a high-reliability underlying business component such as the kernel. Thus, while the benefits of a powerful language like Eiffel are indispensable for a reliable kernel, they are not nearly so important in a GUI application which is likely to be concerned with intelligent window manipulation, XML processing and internet connections.

Thus, the limitations of data types and structures imposed upon the API are useful in the sense that they draw a clear boundary between a very object-oriented type of software which handles complex information structures, and typically event-driven style of software, handling documents and user I/O. However, they do require that the API be devised in a certain way, so as to be valid.

3.2.1 Data Types

In order to guarantee that applications written in any language can use the kernel, only a minimal set of data types will be passed to and received from the kernel. These include the basic programming types recognised in almost all languages, as well as arrays of these types. These are shown below in Eiffel, C and Java syntax.

Eiffel	C
INTEGER	long, int
BOOLEAN	bool
CHARACTER	char
	char
STRING	char *
REAL	float
DOUBLE	double
ARRAY[INTEGER]	int *, int []
ARRAY[BOOLEAN]	bool *, bool []
ARRAY[CHARACTER]	char *, char []
ARRAY[STRING]	char **, (char *) []
ARRAY[REAL]	float *, float []
ARRAY[DOUBLE]	double *, double []

Important things to note:

- There is no pointer or reference type
- The STRING type can be used to pass XML or other document content.

3.2.2 References

Due to the fact that pointers or other references will not generally be valid across a component invocation interface, a different approach is needed when constructing EHR structures. Normally, each part of a transaction would be constructed by calling functions whose arguments were objects constructed by earlier calls, each of which in turn would have been created by calls taking as arguments references to still smaller objects and so on. Without pointers or references, this is no longer possible, and a logical equivalent must be found.

A fairly common approach is to use some kind of "handle" as a way of the client referencing each newly created item. Following this method, actual references or pointers are remembered in the server (note that we call the kernel a "server" even if it is built into the calling application), and logical handles generated for them are returned to the client. Logically, all that is happening is that the server is doing the job of remembering local variables on behalf of the client.

There are a couple of questions which need to be addressed regarding the use of handles:

- What is their type and syntax? For example, are they simply integers, or could a meaningful string be generated instead (this might be useful for debugging of the client application)? Should handles be generated for all objects created in the kernel - even the smallest, such as date-times - or is there an alternative for these?
- In the server (the kernel), each handle is remembered along with the object it stands for. How long should the handle be kept in the server, and what prevents a build-up of handle/object pairs, bloating the memory footprint of the application?

The method proposed for the kernel is as follows:

1. To create objects at the data level, such as terms, text, quantities and so on, handles will not be used; instead, the caller will simply pass all the arguments required, which are generally not numerous. Some simplifications can be made, such as for date-times: a date-time can be specified by the client by passing a single parseable string of standard format rather than the six parameters d, mo, y, h, mi, s.
2. Handles will be used for all other objects, including transactions, content trees, demographic entities and so on. Each handle will be a string of the form `type_name-unique_number`. Thus, the handle for a `STAFF_MEMBER` object created in the kernel could be "STAFF_MEMBER-17".
3. To enable the client to control the lifetime of handles, a "construction context" object will be used. The two contexts in which clients want to construct EHR objects are "session" and "transaction". Thus session context (one only) and transaction context (multiple) objects can be created in the kernel, on behalf of the client. These will retain the set of handles and object references for a session, and for a transaction respectively; when the session or transaction is finished, the handle set is thrown away, just as if they were local variables in a client function.

3.2.3 Example

Let us now return to our example above, and see how these rules would apply. Firstly we want to establish a construction context for the session. For this we need a context management functions.

```
class KERNEL_SESSION

feature -- Context management

        start_context
```

```
        end_context
end
```

Now we want to create the STAFF_MEMBER and HCF objects using API functions of the form:

```
class CAPI_DEMOGRAPHIC_FACTORY

feature -- Context management

        start_context

        end_context

feature -- Factory

        create_staff_member(a_name:STRING; a_position:STRING;
              dob:STRING; pob:STRING):HANDLE
        create_staff_member_args_valid(a_name:STRING; a_position:STRING;
              dob:STRING; pob:STRING):BOOLEAN

        create_hcf(a_name:STRING; a_business_address:HANDLE;
              a_reg:STRING):HANDLE
        create_hcf_args_valid(a_name:STRING; a_business_address:HANDLE;
              a_reg:STRING):BOOLEAN

        create_address(tags:ARRAY[STRING],
              values:ARRAY[STRING]):HANDLE
        create_address_args_valid(tags:ARRAY[STRING],
              values:ARRAY[STRING]):BOOLEAN

end
```

Here, the values passed to create_staff_member are parseable strings, in accordance with rule 1 above. The return type is HANDLE, which is a descendant of STRING. In create_hcf, the business address (logically a DEFINITION_CONTENT) is passed as a HANDLE. This has to have been created by a call to create_address. It is up to the caller to manage the handles. The beginnings of a certain style of API programming can now be seen. Let us complete the original example as it might be programmed in the caller's application (a pseudo-C syntax is used; the preconditions are not used for purposes of clarity):

```
char *H_sm, *H_hcf, *H_addr;

-- some early initialisation stuff
kernel_api.initialise(...args....);

kernel_session.start_context;

H_sm = demographic_factory.create_staff_member("JONES, Dr Robert", "Head of
Department", "13/05/1954", "Melbourne, Australia");

H_addr = demographic_factory.create_address(
<<"street number", "street name", "locality", "county", "postcode", "country">>
<<"73",           "High St",     "Kingston", "Norfolk","ABC 123",  "England">>);

H_hcf = demographic_factory.create_hcf("Nambour Base Hospital", H_addr,
       "Australia - DOH xxx")
```

```
kernel_session.init_kernel_session(H_sm, H_hcf)
```

In this example, the handles `H_sm` and `H_hcf` will remain valid until a call such as the following is made:

```
kernel_session.end_context;
```

The above style of interface, while quite different from the underlying GOM functions, respects the GOM, while allowing the kernel to be used across a binary interface such as COM, from a language such as Visual Basic.

Let us now consider an example in which a record is retrieved, a new transaction is added, and the record committed, to see how the above rules shape a possible component API.

To Be Determined:

3.3 Application Requirements

It is the requirements of applications which decide the "shape" of the API, within the constraints discussed above. There are a number of general requirements, including:

Efficiency: minimum number of calls to achieve common tasks.

Clarity: the calls required to achieve a task should appear sensible to the code reader.

Type compatibility: types of objects required by the kernel must be compatible or dependably convertable to those of the application.

Underlying these, there is the assumption that an application has available to it API calls which can perform each step of a process, which is typically procedural, in the case of GUI applications. That is to say, the API needs to provide calls which in some way satisfy the *temporal calling requirements* of the application. A common API may not completely satisfy the exact needs of every single application, but it should be able to go a long way to satisfying most. For very different applications, there is nothing to stop additionals API being written, while respecting the basic rules stated above.

To Be Determined: more application requirements here.

4 The Ocean GEHR API

In this section, a complete API is described. It is identified as the "Ocean" API, since it was originally proposed by Ocean Informatics, Australia.

To Be Determined:

4.1 Native API

The following sections describe the native API as would be available to an application developer using Eiffel. It is the basis of the API in all its other forms, eg COM etc.

The statistics on the following class groups are approximate; they can be reduced by using a more parameterised style of interface, i.e. feewer functions with extra parameters which are dispatched to the "real" functions at the XXX_IMP class level.

4.1.1 Server & Session

4.1.2 Database (3 classes, 40 features)

4.1.3 Demographics (5 classes, 50 features)

4.1.4 EHRs (2 classes, 20 features)

4.1.5 Transactions (2 classes, ~25 features)

4.1.6 Organisers (2 classes, 20 features)

4.1.7 Content (12 classes, ~200 features)

Content Factories (4 classes, ~36 features)
DEFINITION_CONTENT_FACTORY (6 features)

PHENOMENON_CONTENT_FACTORY (DEFINITION_CONTENT_FACTORY + 7 additional)

SUBJECTIVE_CONTENT_FACTORY (DEFINITION_CONTENT_FACTORY + 7 additional)

INSTRUCTION_CONTENT_FACTORY (DEFINITION_CONTENT_FACTORY + 15? additional)

Content (4 classes, ~40 features)
DEFINITION_CONTENT (4 features)

PHENOMENON_CONTENT (DEFINITION_CONTENT + 8 additional features)

SUBJECTIVE_CONTENT (DEFINITION_CONTENT + 8 additional features)

INSTRUCTION_CONTENT (DEFINITION_CONTENT + 20 additional features)

Propositions (4 classes, ~120 features)
HIERARCHICAL_PROPOSITION (64 features)

SIMPLE_PROPOSITION (HIERARCHICAL_PROPOSITION + 4 additional)

LIST_PROPOSITION (HIERARCHICAL_PROPOSITION + 7 additional)

TABLE_PROPOSITION (HIERARCHICAL_PROPOSITION + 40 additional features)

MATRIX_PROPOSITION

TIME_SERIES_PROPOSITION

REGULAR_TIME_SERIES_PROPOSITION

Examples

The following examples show a fairly accurate use of the content level of the kernel to create new EHR content, using archetypes.

Example: Creating a Weight. This example shows the basic mentality of the interface: to create content, a factory is first instantiated; it is asked to create the actual content (*cf.create_content* call below). Behind the scenes, this call causes an archetype for the concept "weight" to be retrieved, and for it to build its default content, which in this case is a single value (`SIMPLE_PROPOSITION`), and a protocol (`LIST_PROPOSITION`). The succeeding calls simply visit the nodes that have been created, call a pre-condition function (this is a basic feature of programming by contract, which is directly supported in Eiffel), and if it succeeds, call the data-setting function.

This approach is common throughout the GEHR kernel: default content is created, and modification the proceeds by visiting what has been created. The default content can also be used to immediately present something in the graphical interface.

```
execute is
        local
                sp: SIMPLE_PROPOSITION;
                vc: VALUE_CURSOR;
                cf: OBSERVATION_CONTENT_FACTORY;
                pt: PLAIN_TEXT;
                q: QUANTITY
        do
                create cf;
                cf.create_current_context;
                cf.create_item (df.create_plain_text ("weight"));
                vc ?= cf.item_protocol_cursor;
                vc.go_to_name ("instrument");
                pt := df.create_plain_text ("Seca scales");
                if vc.data_value_valid (pt) then
                        vc.replace_data_value (pt)
                else
                        log_event (generator, "execute", "vc.replace_data_value " + pt.value + " failed", error)
                end;
                vc.go_to_name ("precision");
                q := df.create_dimensioned_quantity (0.5, "mass", "kg");
                if vc.data_value_valid (q) then
                        vc.replace_data_value (q)
                else
                        log_event (generator, "execute", "vc.replace_data_value " + q.out + " failed", error)
                end;
                sp ?= cf.item_proposition;
                q := df.create_dimensioned_quantity (85, "mass", "kg");
```

```
            if sp.data_value_valid (q) then
                sp.replace_data_value (q)
            else
                log_event (generator, "execute", "sp.replace_data_value " + q.out + " failed", error)
            end;
            io.put_string ("%N---------- Populated value ----------%N");
            io.put_string (cf.item.out)
        end;
```

Example: Creating a Blood Pressure. This example is of the same approach as the last, except using a blood pressure archetype.

```
    execute is
        local
            cf: OBSERVATION_CONTENT_FACTORY;
            vc: VALUE_CURSOR;
            tt: TERM_TEXT;
            q: QUANTITY
        do
            create cf;
            cf.create_current_context;
            cf.create_item (df.create_plain_text ("blood pressure"));
            vc ?= cf.item_protocol_cursor;
            vc.go_to_name ("instrument");
            tt := df.create_term_text (<<gehr_clinical_ts, "0015", expansion_for_code ("0015"), "0">>);
            if vc.data_value_valid (tt) then
                vc.replace_data_value (tt)
            else
                log_event (generator, "execute", "bp_protocol.replace_data_value " + tt.value + " failed", error)
            end;
            vc.go_to_name ("cuff size");
            tt := df.create_term_text (<<gehr_clinical_ts, "0019", expansion_for_code ("0019"), "0">>);
            if vc.data_value_valid (tt) then
                vc.replace_data_value (tt)
            else
                log_event (generator, "execute", "bp_protocol.replace_data_value " + tt.value + " failed", error)
            end;
            vc.go_to_name ("position");
            tt := df.create_term_text (<<gehr_clinical_ts, "0018", expansion_for_code ("0018"), "0">>);
            if vc.data_value_valid (tt) then
                vc.replace_data_value (tt)
            else
                log_event (generator, "execute", "bp_protocol.replace_data_value " + tt.value + " failed", error)
            end;
            vc ?= cf.item_cursor;
            vc.go_to_name ("systolic");
            q := df.create_dimensioned_quantity (110, "pressure", "mm[Hg]");
```

```
            if vc.data_value_valid (q) then
                vc.replace_data_value (q)
            else
                log_event (generator, "execute", "bp.replace_data_value " + q.out + " failed", error)
            end;
            vc.go_to_name ("diastolic");
            q := df.create_dimensioned_quantity (80, "pressure", "mm[Hg]");
            if vc.data_value_valid (q) then
                vc.replace_data_value (q)
            else
                log_event (generator, "execute", "bp.replace_data_value " + q.out + " failed", error)
            end;
            io.put_string ("%N---------- Populated value ----------%N");
            io.put_string (cf.item.out)
        end
```

Example: Creating an Audiogram. In this example, no pre-condition calls are used; if it is known *a priori* that the intended data-setting calls will work, there is no great risk.

```
    execute is
        local
            tc: TREE_CURSOR;
            vc: VALUE_CURSOR;
            cf: OBSERVATION_CONTENT_FACTORY;
            pt: PLAIN_TEXT;
            qt: QUANTITY
        do
            create cf;
            cf.create_current_context;
            cf.create_item (df.create_plain_text ("audiogram"));
            vc ?= cf.item_protocol_cursor;
            vc.go_to_name ("equipment");
            pt := df.create_plain_text ("Welsh-Allen model A1");
            if vc.data_value_valid (pt) then
                vc.replace_data_value (pt)
            else
                log_event (generator, "execute", "audiogram_protocol.replace_data_value " + pt.value + " failed: ",
error)
            end;
            vc.go_to_name ("soundproof");
            pt := df.create_plain_text ("Std AS9999");
            if vc.data_value_valid (pt) then
                vc.replace_data_value (pt)
            else
                log_event (generator, "execute", "audiogram_protocol.replace_data_value " + pt.value + " failed", error)
            end;
            vc.go_to_name ("duration");
```

```
qt := df.create_dimensioned_quantity (180, "time", "ms");
if vc.data_value_valid (qt) then
        vc.replace_data_value (qt)
else
        log_event (generator, "execute", "audiogram_protocol.replace_data_value " + qt.value.out + " failed",
error)
end;
vc.go_to_name ("amplitude");
vc.replace_data_value (df.create_dimensioned_quantity (21, "voltage", "uV"));
tc ?= cf.item_cursor;
tc.set_to_path ("|%"audiogram%"|%"left ear%"");
vc := tc.value_cursor;
vc.go_to_name ("1000 Hz threshold");
qt := df.create_dimensioned_quantity (0, "pressure", "dB");
if vc.data_value_valid (qt) then
        vc.replace_data_value (qt)
else
        log_event (generator, "execute", "audiogram.replace_data_value " + qt.value.out + " failed", error)
end;
vc.go_to_name ("2000 Hz threshold");
qt := df.create_dimensioned_quantity (5, "pressure", "dB");
if vc.data_value_valid (qt) then
        vc.replace_data_value (qt)
else
        log_event (generator, "execute", "audiogram.replace_data_value " + qt.value.out + " failed", error)
end;
vc.go_to_name ("4000 Hz threshold");
qt := df.create_dimensioned_quantity (20, "pressure", "dB");
if vc.data_value_valid (qt) then
        vc.replace_data_value (qt)
else
        log_event (generator, "execute", "audiogram.replace_data_value " + qt.value.out + " failed", error)
end;
tc.set_to_path ("|%"audiogram%"|%"right ear%"");
vc := tc.value_cursor;
vc.go_to_name ("1000 Hz threshold");
qt := df.create_dimensioned_quantity (0, "pressure", "dB");
if vc.data_value_valid (qt) then
        vc.replace_data_value (qt)
else
        log_event (generator, "execute", "audiogram.replace_data_value " + qt.value.out + " failed", error)
end;
vc.go_to_name ("2000 Hz threshold");
qt := df.create_dimensioned_quantity (10, "pressure", "dB");
if vc.data_value_valid (qt) then
        vc.replace_data_value (qt)
```

```
            else
                log_event (generator, "execute", "audiogram.replace_data_value " + qt.value.out + " failed", error)
            end;
            vc.go_to_name ("4000 Hz threshold");
            qt := df.create_dimensioned_quantity (30, "pressure", "dB");
            if vc.data_value_valid (qt) then
                vc.replace_data_value (qt)
            else
                log_event (generator, "execute", "audiogram.replace_data_value " + qt.value.out + " failed", error)
            end;
            io.put_string ("%N---------- Populated value ----------%N");
            io.put_string (cf.item.out)
        end;
```

4.2 COM API

The following selection of classes is intended to give an idea of the size of the COM interface (see the kernel classes for latest interface definitions).

4.2.1 COM Server

Interface of *COM_SERVER*:

(from Class *DB_APPLICATION*:)

make

(from Class *PROXY_DEMOGRAPHIC_MANAGER*:)

demographic_manager: *DEMOGRAPHIC_MANAGER*

(from Class *SHARED_KERNEL_SESSION*:)

initialise_kernel_session (an_ehr_source_id: *STRING*)

kernel_session: *KERNEL_SESSION*

set_hcf (an_hcf: *STRING*)

(from Class *ERROR_STATUS*:)

```
    fail_reason: STRING        -- Error features to be replaced by shared ERROR objects
    last_op_fail: BOOLEAN
```

(from Class *GEHR_APPLICATION*:)

```
    kernel: KERNEL             -- Not sure if this is needed in COM
```

(from Class *COM_SERVER*:)

main

4.2.2 KERNEL_SESSION

Class *KERNEL_SESSION*:

active_user: *INTEGER*

active_user_id: *STRING*

add_user (a_user_id: *STRING*; a_user_name: *STRING*; an_access_level: *INTEGER*; a_security_token: *STRING*; a_pin: *INTEGER*)

```
    ehr_context: EHR_FACTORY           -- Start from here to create/retrieve EHRs
```

ehr_source_id: *STRING*

has_user (a_pin: *INTEGER*): *BOOLEAN*

has_user_id (a_user_id: *STRING*): *BOOLEAN*

hcf: *STRING*

lowest_access_level: *INTEGER*

lowest_access_user: *INTEGER*

maximum_access_level: *INTEGER*

remove_user (a_pin: *INTEGER*)

set_active_user (a_pin: *INTEGER*)

set_hcf (an_hcf: *STRING*)

user (a_pin: *INTEGER*): *KERNEL_SESSION_USER*

valid_login (a_user_id: *STRING*; an_access_level: *INTEGER*; a_security_token: *STRING*): *BOOLEAN*

4.2.3 DEMOGRAPHIC_MANAGER

Class *DEMOGRAPHIC_MANAGER*:

has_party (key: *STRING*): *BOOLEAN*

is_valid: *BOOLEAN*

make

party (key: *STRING*): *PARTY* `-- probably not needed in COM`

party_ids: *ARRAYED_LIST* [*STRING*]

put_party (an_id: *STRING*)

4.2.4 EHR_FACTORY

Class *EHR_FACTORY*:

commit_ehr

create_ehr (patient_id: *STRING*; hca_auth: *STRING*)

create_ehr_commit_reason: *STRING*

ehr: *EHR*

ehr_exists (patient_id: *STRING*): *BOOLEAN*

ehr_ids: *STRING*

rep: *REP_CLIENT*

retrieve_ehr (patient_id: *STRING*)

4.2.5 TRANSACTION_FACTORY

Class *TRANSACTION_FACTORY*:

commit_transaction

commit_version (a_parent_version_id: *STRING*; hca_auth: *STRING*; a_reason: *STRING*; content: *EHR_CONTENT*)

create_versioned_transaction

ehr_id: *STRING*

rep: *REP_CLIENT*

set_ehr_id (an_ehr_id: *STRING*)

set_versioned_transaction (vt_id: *STRING*)

valid_demographic_id (an_id: *STRING*): *BOOLEAN*

versioned_transaction: *VERSIONED_TRANSACTION*

4.2.6 deferred ARCHETYPED_FACTORY

Class *ARCHETYPED_FACTORY*:

archetype_exists: *BOOLEAN*

archetype_id: *STRING*

create_default

create_item (a_concept: *PLAIN_TEXT*)

item: [like *item_anchor*]: *ORGANISER_ROOT*

modify_item (an_item: [like *item_anchor*]: *ORGANISER_ROOT*)

retrieve_archetype (a_concept: *PLAIN_TEXT*)

select_sub_archetype (an_archetype_id: *STRING*; a_sub_archetype_key: *STRING*)

selected_sub_archetypes: *HASH_TABLE* [*ARRAYED_LIST* [*STRING*], *STRING*]

sub_archetype_factories: *HASH_TABLE* [*ARRAYED_LIST* [*ARCHETYPED_FACTORY*], *STRING*]

sub_archetype_id_patterns: *HASH_TABLE* [*STRING*, *STRING*]

sub_archetype_ids: *STRING*

valid_archetype_concept (a_concept: *PLAIN_TEXT*): *BOOLEAN*

valid_archetype_id (an_id: *STRING*): *BOOLEAN*

valid_sub_archetype_id (an_archetype_id: *STRING*; a_sub_archetype_key: *STRING*): *BOOLEAN*

4.2.7 ORGANISER_FACTORY

Class *ORGANISER_FACTORY*:

(inherit Class *ARCHETYPED_FACTORY*:)

context_valid: *BOOLEAN*

handles_set: *BOOLEAN*

install_sub_archetypes

organiser: *ORGANISER_ROOT*

valid_archetype (an_archetype: *ARCHETYPE*; a_sub_archetype_key: *STRING*): *BOOLEAN*

4.2.8 ORGANISER

Class *ORGANISER*:

add_content_item (a_content_item: *DEFINITION_CONTENT*)

add_organiser (an_organiser: *ORGANISER*)

all_items: *LINKED_LIST* [*DEFINITION_CONTENT*]

arch_add_content_item_valid (a_content_item: *DEFINITION_CONTENT*): *BOOLEAN*

arch_add_organiser_valid (an_organiser: *ORGANISER*): *BOOLEAN*

arch_remove_content_item_valid (a_content_item: *DEFINITION_CONTENT*): *BOOLEAN*

arch_remove_organiser_valid (an_organiser: *ORGANISER*): *BOOLEAN*

arch_set_name_valid (a_name: *PLAIN_TEXT*): *BOOLEAN*

content: *LINKED_LIST* [*DEFINITION_CONTENT*]

content_with_name (a_name: *STRING*): *DEFINITION_CONTENT*

data_value_at_path (a_path: *STRING*): *DATA_VALUE*

default_create

is_root: *BOOLEAN*

make (a_name: *PLAIN_TEXT*)

name: *PLAIN_TEXT*

organiser_at_path (a_path: *STRING*): *ORGANISER*

organiser_with_name (a_name: *STRING*): *ORGANISER*

organisers: *LINKED_LIST* [*ORGANISER*]

out: *STRING*

remove_content_item (a_content_item: *DEFINITION_CONTENT*)

remove_organiser (an_organiser: *ORGANISER*)

set_name (a_name: *PLAIN_TEXT*)

valid_content_path (a_path: *STRING*): *BOOLEAN*

valid_organiser_path (a_path: *STRING*): *BOOLEAN*

4.2.9 DEFINITION_CONTENT_FACTORY

Class *DEFINITION_CONTENT_FACTORY*:
(inherit Class *ARCHETYPED_FACTORY*:)

context_valid: *BOOLEAN*

handles_set: *BOOLEAN*

install_sub_archetypes

item_cursor: *HIERARCHICAL_CURSOR*

item_proposition: *HIERARCHICAL_PROPOSITION*

valid_archetype (an_archetype: *ARCHETYPE*; a_sub_archetype_key: *STRING*): *BOOLEAN*

4.2.10 DEFINITION_CONTENT

Class *COMPOSED_OBJECT*:

is_valid: *BOOLEAN*

(from Class *ARCHETYPED*:)

concept: *PLAIN_TEXT*

gehr_archetype_id: *STRING*

key: *STRING*

Class *DEFINITION_CONTENT*:

item_at_locator (a_locator: *STRING*): *G1_ANY*

links: *HASH_TABLE* [*LOCATOR*, *TERM_TEXT*]

locator_id: *STRING*

out: *STRING*

proposition: *HIERARCHICAL_PROPOSITION*

valid_locator (a_locator: *STRING*): *BOOLEAN*

4.2.11 DATA_FACTORY

Class *DATA_FACTORY*:

create_date (date_str: *STRING*): *DATE_IMPL*

create_date_time (date_time_str: *STRING*): *DATE_TIME_IMPL*

create_date_time_duration (date_time_duration_str: *STRING*): *DATE_TIME_DURATION_IMPL*

create_dimensioned_quantity (a_value: *REAL*; a_property: *STRING*; a_units: *STRING*): *QUANTITY*

create_dimensioned_quantity_range (a_lower_value: *REAL*; an_upper_value: *REAL*; a_property: *STRING*; a_units: *STRING*): *QUANTITY_RANGE*

create_dimensionless_quantity (a_value: *REAL*): *QUANTITY*

create_dimensionless_quantity_range (a_lower_value: *REAL*; an_upper_value: *REAL*): *QUANTITY_RANGE*

create_plain_text (s: *STRING*): *PLAIN_TEXT*

create_quantity_ratio (q1: *QUANTITY*; q2: *QUANTITY*): *QUANTITY_RATIO*

create_term_text (tokens: *ARRAY* [*STRING*]): *TERM_TEXT*

create_time (time_str: *STRING*): *TIME_IMPL*

4.2.12 deferred HIERARCHICAL_PROPOSITION

(from Class *HIERARCHICAL_ITEM*:)

path: *LOCATOR_PATH*

Class *HIERARCHICAL_PROPOSITION*:

as_string: *STRING*

context: *ANY_CONTEXT*

name: *PLAIN_TEXT*

valid_value_path (path_str: *STRING*): *BOOLEAN*

4.2.13 SIMPLE_PROPOSITION

(inherit Class *HIERARCHICAL_PROPOSITION*:)

Class *SIMPLE_PROPOSITION*:

as_string: *STRING*

cursor: *HIERARCHICAL_CURSOR*

data_value_valid (a_value: *DATA_VALUE*): *BOOLEAN*

form: *INTEGER*

replace (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*)

replace_data_value (a_value: *DATA_VALUE*)

value_valid (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*): *BOOLEAN*

4.2.14 TREE_PROPOSITION

(inherit Class *HIERARCHICAL_PROPOSITION*:)

Class *TREE_PROPOSITION*:

cursor: *TREE_CURSOR*

remove_i_th_branch (i: *INTEGER*) -- probably shouldn't be visible

remove_i_th_value (i: *INTEGER*) -- probably shouldn't be visible

tree_consistent: *BOOLEAN*

value_count: *INTEGER*

4.2.15 LIST_PROPOSITION

(inherit Class *HIERARCHICAL_PROPOSITION*:)

Class *LIST_PROPOSITION*:

cursor: *VALUE_CURSOR*

form: *INTEGER*

4.2.16 TABLE_PROPOSITION

(inherit Class *HIERARCHICAL_PROPOSITION*:)

Class *TABLE_PROPOSITION*:

add_row (new_row: *PLAIN_TEXT*)

append_row (a_row: *ARRAY* [*DATA_VALUE*])

as_string: *STRING*

cell (row: *INTEGER*; column: *INTEGER*): *DATA_VALUE*

column_count: *INTEGER*

column_name_index (a_name: *STRING*): *INTEGER*

column_path_specifier: *STRING*

cursor: *GROUP_CURSOR*

data_value_valid (a_value: *DATA_VALUE*; a_col: *INTEGER*; a_row: *INTEGER*): *BOOLEAN*

default_row: *ARRAY* [*DATA_VALUE*]

form: *INTEGER*

i_th_column (i: *INTEGER*): *ARRAY* [*DATA_VALUE*]

i_th_row (i: *INTEGER*): *ARRAY* [*DATA_VALUE*]

insert_row (before_row: *INTEGER*; a_row: *ARRAY* [*DATA_VALUE*])

remove_column (i: *INTEGER*)

remove_row (i: *INTEGER*)

replace (a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*; a_col: *INTEGER*; a_row: *INTEGER*)

replace_row (new_row: *ARRAY* [*DATA_VALUE*]; i: *INTEGER*)

replace_value (a_value: *DATA_VALUE*; a_col: *INTEGER*; a_row: *INTEGER*)

row_count: *INTEGER*

row_cursor: *ROW_CURSOR*

row_morphable: *BOOLEAN*

row_path (ix: *INTEGER*): *LOCATOR_PATH*

row_path_specifier: *STRING*

set_column_name (new_name: *PLAIN_TEXT*; col_nr: *INTEGER*)

table_consistent: *BOOLEAN*

valid_column (n: *INTEGER*): *BOOLEAN*

valid_row (n: *INTEGER*): *BOOLEAN*

valid_row_data (a_row: *ARRAY* [*DATA_VALUE*]): *BOOLEAN*

value_valid (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*; a_col: *INTEGER*; a_row: *INTEGER*): *BOOLEAN*

variable_table: *BOOLEAN*

4.2.17 deferred HIERARCHICAL_CURSOR

Class *HIERARCHICAL_CURSOR*:

exhausted: *BOOLEAN*

full: *BOOLEAN*

readable: *BOOLEAN*

writable: *BOOLEAN*

4.2.18 deferred HIERARCHICAL_LINEAR_CURSOR

Class *HIERARCHICAL_LINEAR_CURSOR*:

empty: *BOOLEAN*

remove_left

remove_right

valid_key (v: *ANY*): *BOOLEAN*

4.2.19 deferred HIERARCHICAL_GROUP_CURSOR

Class *LINKED_LIST_CURSOR* [G]:

after: *BOOLEAN*

before: *BOOLEAN*

(inherit Class *HIERARCHICAL_CURSOR*:)

(inherit Class *HIERARCHICAL_LINEAR_CURSOR*:)

Class *HIERARCHICAL_GROUP_CURSOR* [G -> *HIERARCHICAL_ITEM*]:

infix@ (i: *INTEGER*): [like *item*]: *HIERARCHICAL_VALUE*

back

count: *INTEGER*

extendible: *BOOLEAN*

finish

first: [like *item*]: *HIERARCHICAL_VALUE*

forth

go_i_th (i: *INTEGER*)

go_to_name (a_name: *STRING*)

has (v: [like *item*]: *HIERARCHICAL_VALUE*): *BOOLEAN*

has_item_with_name (a_name: *STRING*): *BOOLEAN*

i_th (i: *INTEGER*): [like *item*]: *HIERARCHICAL_VALUE*

index: *INTEGER*

index_of (v: [like *item*]: *HIERARCHICAL_VALUE*; i: *INTEGER*): *INTEGER*

is_at_first: *BOOLEAN*

is_at_last: *BOOLEAN*

item_context: *ANY_CONTEXT*

item_name: *PLAIN_TEXT*

item_with_name (a_name: *STRING*): [like *item*]: *HIERARCHICAL_VALUE*

last: [like *item*]: *HIERARCHICAL_VALUE*

move (i: *INTEGER*)

name: *PLAIN_TEXT*

occurrences (v: *HIERARCHICAL_VALUE*): *INTEGER*

off: *BOOLEAN*

path: *LOCATOR_PATH*

prunable: *BOOLEAN*

start

target: *HIERARCHICAL_GROUP*

valid_index (i: *INTEGER*): *BOOLEAN*

4.2.20 TREE_CURSOR

(inherit Class *HIERARCHICAL_CURSOR*:)

Class *TREE_CURSOR*:

after: *BOOLEAN*

before: *BOOLEAN*

detach_item

down (i: *INTEGER*)

empty: *BOOLEAN*

extendible: *BOOLEAN*

group_at_path (path_str: *STRING*): [like *item*]: *HIERARCHICAL_GROUP*

group_cursor: *GROUP_CURSOR*

group_item_detachable: *BOOLEAN*

is_at_first: *BOOLEAN*

item_set: *BOOLEAN*

make (t: [like *target*]: *TREE_PROPOSITION*)

name: *PLAIN_TEXT*

off: *BOOLEAN*

path: *LOCATOR_PATH*

prunable: *BOOLEAN*

remove_i_th (i: *INTEGER*)

set_to_path (path_str: *STRING*)

start

up

valid_group_path (path_str: *STRING*): *BOOLEAN*

valid_index (i: *INTEGER*): *BOOLEAN*

valid_path (path_str: *STRING*): *BOOLEAN*

value_cursor: *VALUE_CURSOR*

4.2.21 VALUE_CURSOR

Class *VALUE_CURSOR*:

(inherit Class *HIERARCHICAL_GROUP_CURSOR* [*HIERARCHICAL_VALUE*]:)

add_end (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*)

add_front (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*)

add_left (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*)

add_right (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*)

data_value_valid (a_value: *DATA_VALUE*): *BOOLEAN*

make (the_target: *HIERARCHICAL_GROUP*)

new_value_valid (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*): *BOOLEAN*

replace (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*)

replace_data_value (a_value: *DATA_VALUE*)

value_valid (a_name: *PLAIN_TEXT*; a_value: *DATA_VALUE*; a_context: *ANY_CONTEXT*): *BOOLEAN*

values_extendible: *BOOLEAN*

4.2.22 GROUP_CURSOR

Class *GROUP_CURSOR*:

(inherit Class *HIERARCHICAL_GROUP_CURSOR* [*HIERARCHICAL_GROUP*]:)

add_end (a_name: *PLAIN_TEXT*; a_context: *ANY_CONTEXT*)

add_front (a_name: *PLAIN_TEXT*; a_context: *ANY_CONTEXT*)

add_left (a_name: *PLAIN_TEXT*; a_context: *ANY_CONTEXT*)

add_right (a_name: *PLAIN_TEXT*; a_context: *ANY_CONTEXT*)

group_valid (a_name: *PLAIN_TEXT*; a_context: *ANY_CONTEXT*): *BOOLEAN*

groups_extendible: *BOOLEAN*

make (the_target: *HIERARCHICAL_GROUP*)

new_group_valid (a_name: *PLAIN_TEXT*; a_context: *ANY_CONTEXT*): *BOOLEAN*

replace (a_name: *PLAIN_TEXT*; a_context: *ANY_CONTEXT*)

set_name (a_name: *PLAIN_TEXT*)

4.2.23 ROW_CURSOR

(inherit Class *HIERARCHICAL_CURSOR*:)

(inherit Class *HIERARCHICAL_LINEAR_CURSOR*:)

Class *ROW_CURSOR*:

infix@ (i: *INTEGER*): [like *item*]: *ARRAY* [*DATA_VALUE*]

after: *BOOLEAN*

append (s: [like *item*]: *ARRAY* [*DATA_VALUE*])

back

before: *BOOLEAN*

column_count: *INTEGER*

count: *INTEGER*

extend (v: [like *item*]: *ARRAY* [*DATA_VALUE*])

extendible: *BOOLEAN*

fill (other: [like *item*]: *ARRAY* [*DATA_VALUE*])

finish

first: [like *item*]: *ARRAY* [*DATA_VALUE*]

forth

go_i_th (i: *INTEGER*)

go_to_name (a_name: *STRING*)

has (key: [like *item*]: *ARRAY* [*DATA_VALUE*]): *BOOLEAN*

has_item_with_name (a_name: *STRING*): *BOOLEAN*

i_th (i: *INTEGER*): [like *item*]: *ARRAY* [*DATA_VALUE*]

i_th_name (i: *INTEGER*): *PLAIN_TEXT*

index: *INTEGER*

index_of (key: [like *item*]: *ARRAY* [*DATA_VALUE*]; occurrence: *INTEGER*): *INTEGER*

is_at_first: *BOOLEAN*

is_at_last: *BOOLEAN*

item: *ARRAY* [*DATA_VALUE*]

item_with_name (a_name: *STRING*): [like *item*]: *ARRAY* [*DATA_VALUE*]

last: [like *item*]: *ARRAY* [*DATA_VALUE*]

move (i: *INTEGER*)

name: *PLAIN_TEXT*

occurrences (key: [like *item*]: *ARRAY* [*DATA_VALUE*]): *INTEGER*

off: *BOOLEAN*

path: *LOCATOR_PATH*

prunable: *BOOLEAN*

prune (key: [like *item*]: *ARRAY* [*DATA_VALUE*])

prune_all (key: [like *item*]: *ARRAY* [*DATA_VALUE*])

put (v: [like *item*]: *ARRAY* [*DATA_VALUE*])

put_front (v: [like *item*]: *ARRAY* [*DATA_VALUE*])

put_i_th (v: [like *item*]: *ARRAY* [*DATA_VALUE*]; i: *INTEGER*)

put_left (v: [like *item*]: *ARRAY* [*DATA_VALUE*])

put_right (v: [like *item*]: *ARRAY* [*DATA_VALUE*])

remove

remove_i_th (i: *INTEGER*)

replace (v: [like *item*]: *ARRAY* [*DATA_VALUE*])

search (key: [like *item*]: *ARRAY* [*DATA_VALUE*])

start

valid_index (i: *INTEGER*): *BOOLEAN*

valid_key (key: [like *item*]: *ARRAY* [*DATA_VALUE*]): *BOOLEAN*

END OF DOCUMENT