
# The GEHR Kernel Architecture

*Authors: Thomas Beale*

Revision: 2.1 draft C

Pages: 41


**email**: info@gehr.org **web**: www.gehr.org

## Amendment Record

| Issue | Details | Who | Date |
|---|---|---|---|
| 1.1 draft A | Initial Writing. Content taken from archetype system document. | T Beale | 2 Aug 2000 |
| 2.1 | Updated to include factory and archetype design, native API. | T Beale | 20 Aug 2000 |
| 2.1 draft B | Changed some SHARED_ classes to PROXY_ classes. | T Beale | 27 Sep 2000 |
| 2.1 draft C | Update to kernel version 36 level - changed some details of archetype processing. | T Beale | 8 Nov 2000 |

## Table of Contents

# 1 Introduction

## 1.1 Purpose

This document describes the architecture of the GEHR kernel, in terms of class models, exported interfaces, additional facilities and archetype implementation details. It is intended as a guide to the design intention and programmatic interfaces of the kernel. Note however that only the native interface is described here. Interfaces such as COM and CORBA are detailed in the OCEAN GEHR-compliant Kernel Application Programmer's Interface document.

## 1.2 Audience

The primary users of this document are:

*GEHR kernel developers*: this document describes the design and implementation decisions taken in the current version of the kernel.

*GEHR-based application and system developers*: the native API is described.

## 1.3 Status

This document is under construction. Known omissions or questions are indicated in the text with paragraphs like the following:

*To Be Determined:* indicating not yet resolved

*To Be Continued:* indicating more work required

Reviewers are invited to comment on these paragraphs as well as the main content.

# 2 Overview

## 2.1 System Architecture

Because GEHR proposes an information model from which software artifacts can be derived, numerous architectural possibilities are available, undoubtedly including some unforeseen by the original authors of GEHR. The aim of this document is therefore to describe the most likely architectural possibilities. While the details may be quite different, all GEHR-based systems consist of the following elements:

- **User Applications**, consisting of:
  - An application specific part, typically a GUI interface.
  - The GEHR kernel.
  - A client for a persistence mechanism (database).
- **Archetype Initialiser** application, which converts XML archetypes to internal form for GEHR applications to use.
- **Demographic Manager** application, which converts information from the demographic server to internal form for GEHR applications to use.
- A **database**, containing repositories for:
  - EHRs created/modified/viewed by applications.
  - Locally used GEHR archetypes.
  - Locally created GEHR archetype XML documents, where these exist.
- **Terminology server**(s), which provide access to well-known term sets such as UMLS, ICPC and so on. The simplest version of this might be a small local application serving terms from a file of locally defined terms; a more sophisticated version might be an implementation of the CORBAmed TQS specification.
- **Demographic server**, providing identification of person and organisational entities. At its most minimal form, this might be a simple local database; the most sophisticated version might be an implementation of the CORBAmed PIDS specification.
- **Archetype domain server**, providing access to the GEHR archetype domain system.

FIGURE 1 illustrates these elements.

## 2.2 Application Architecture

The software architecture of GEHR applications is illustrated in FIGURE 2. A GEHR kernel application inherits from two important classes: `GEHR_APPLICATION` and `*_DB_APPLICATION`, where the latter is one of the concrete descendants of the OSTORE class `DB_APPLICATION`.

`GEHR_APPLICATION` provides access to shared services, such as the kernel session, term services and demographic server, as well as some basic services, including event logging and configuration file access.

Using the OSTORE class simply provides the application - in particular, the `EHR_FACTORY` - access to database services, via a number of self-contained interfaces. These interfaces form the upper layer of the OSTORE library, and are implemented by bindings to various databases, which are interchangeable.

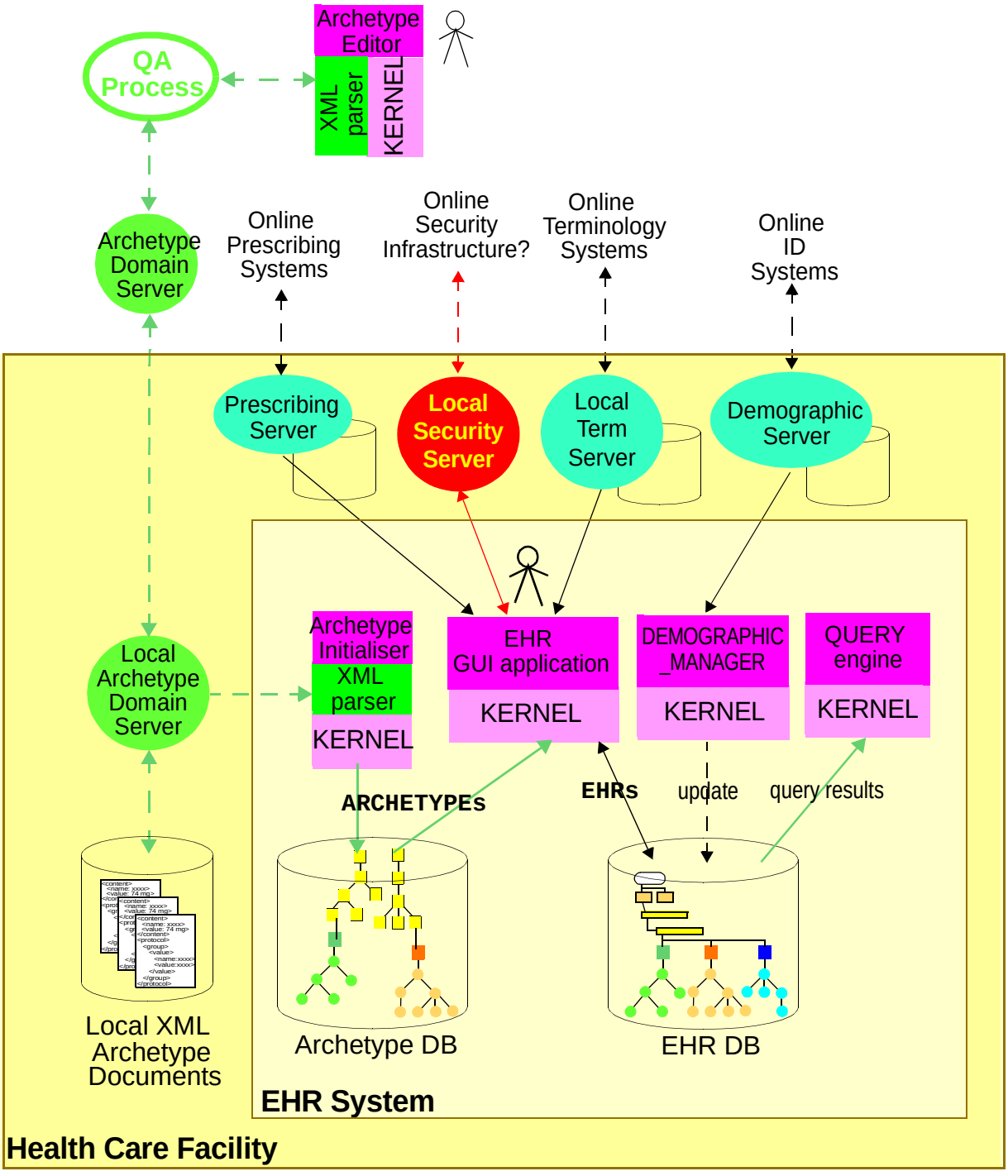


**FIGURE 1** GEHR System Architecture

External services visible to `GEHR_APPLICATION` are shown as being connected by classes named `PROXY_*`. In test systems where the external services are incorporated in the main application, `PROXY_*` classes are implemented with the Eiffel once routine; in multi-process systems, these classes use an IPC (inter-process communication) mechanism, such as EiffelNet, if in Eiffel, or CORBA or COM, or even TCP/IP socket communication. The outer dashed box marks the notional separation of processes in larger systems.

Applications connect to the kernel via the `KERNEL_SESSION` class, the `KERNEL` root class, and via other classes exposed through the API (illustrated in the abstract as class “aa”, “bb”, ... “xx”, “yy”, “zz” in the diagram). The dashed box marking the kernel boundary corresponds to the kernel API visible to client programmers.

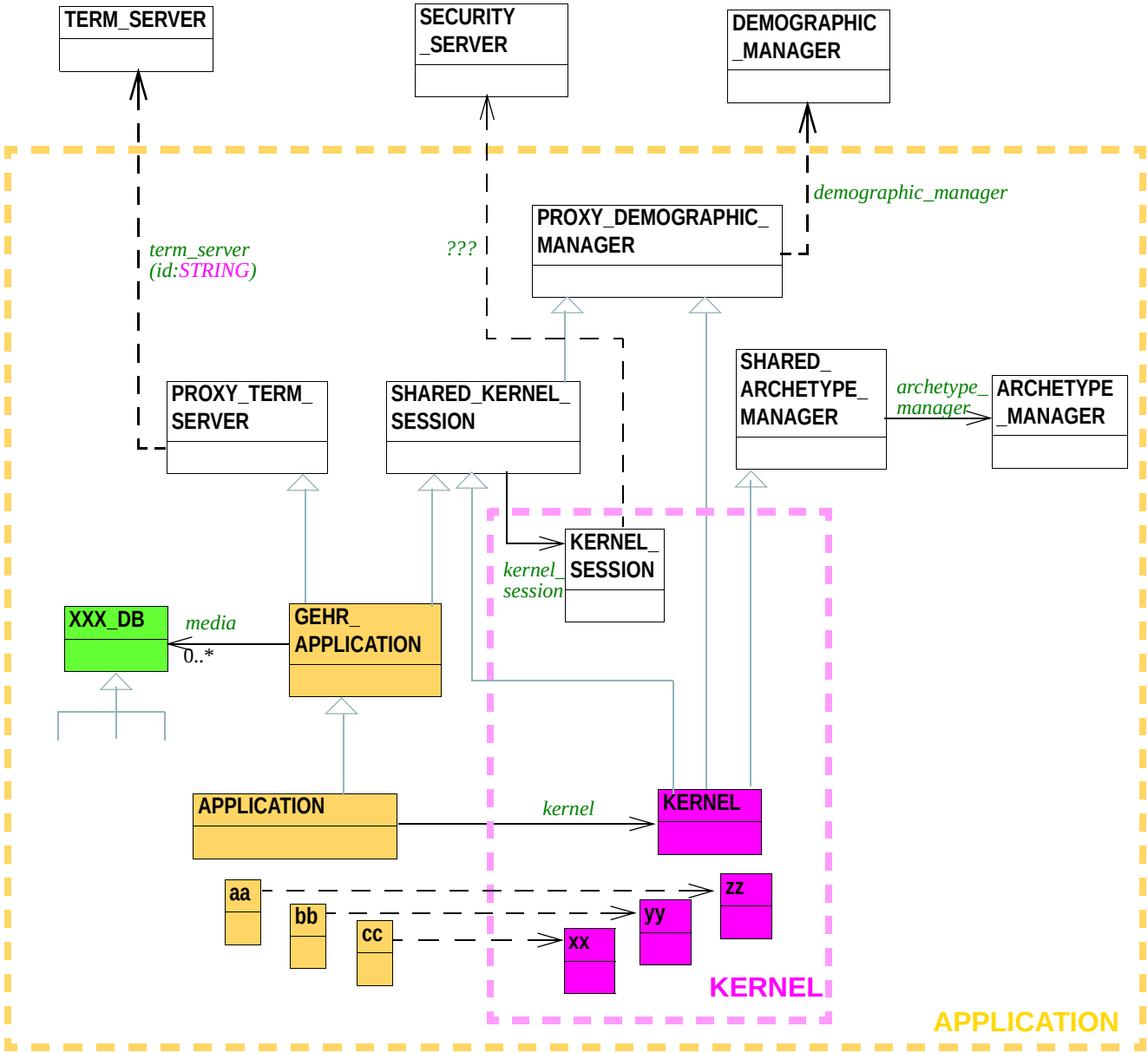


**FIGURE 2** General GEHR Application Structure

# 3 Sessions and Security

## 3.1 System Architecture

To Be Determined: this section under construction

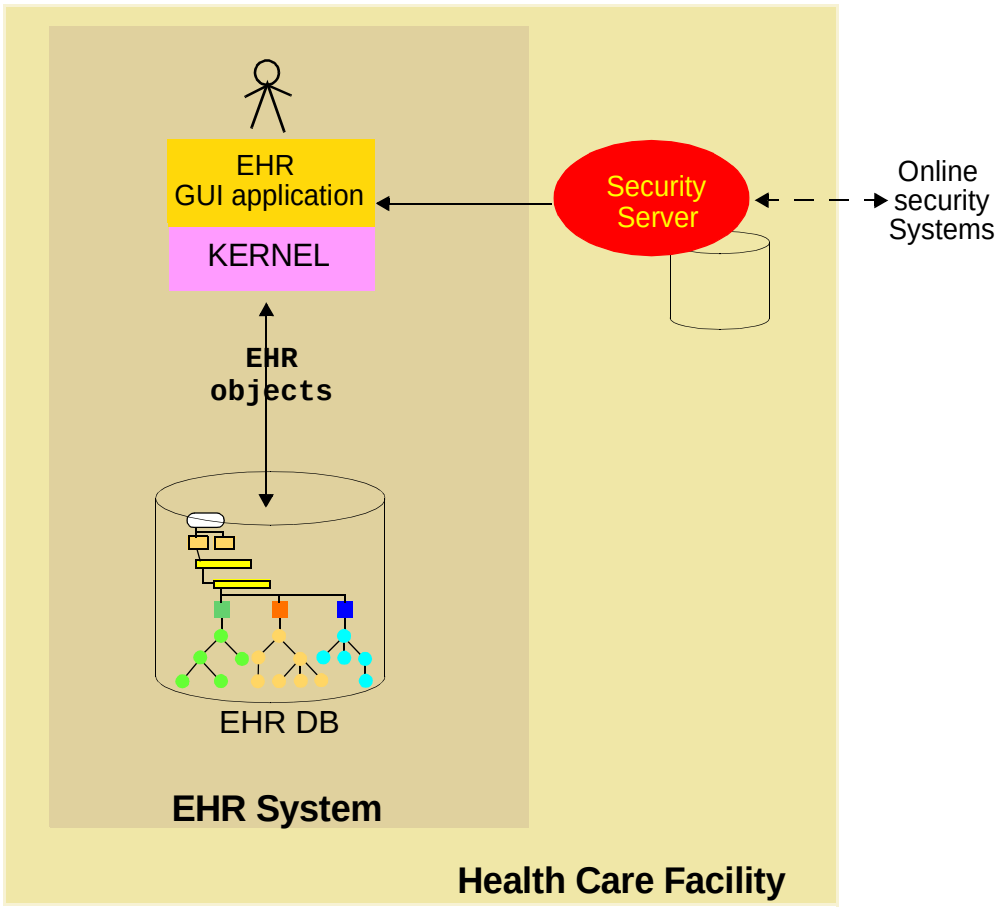


**FIGURE 3** GEHR Security System

## 3.2 Software Architecture

To Be Determined: this section under construction

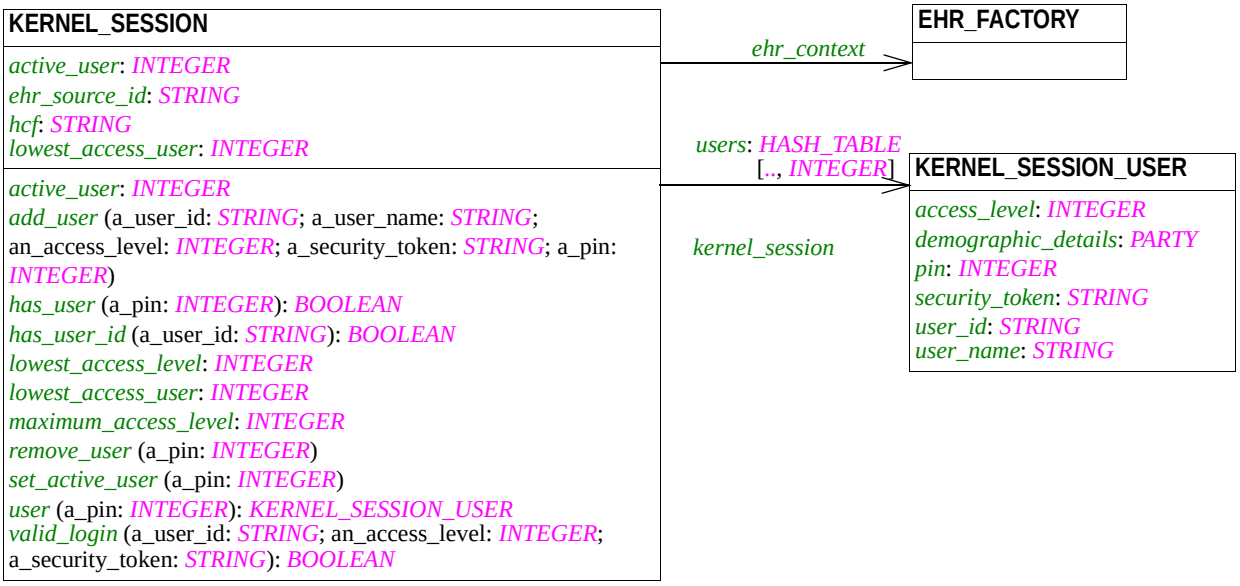


**FIGURE 4** Kernel Cluster

# 4 Demographic Interface

## 4.1 System Architecture

In dealing with demographic information, the underlying assumption of the GEHR kernel architecture is that there will usually be an existing demographic server or service, such as a hospital PMI (patient master index) or a small database at a GP clinic. Other services might exist as well, such as the distributed online CORBAmed PIDS (Party Identification Service). In general, nothing can be assumed about these services by a generic component such as the GEHR kernel.

However, as described in Versioning of Demographic Information on page 23 of the The GEHR Object Model Architecture, the kernel needs access to snapshots of demographic information for versioning purposes, and to guarantee that record extracts are sensible.

In order to deal with this problem, the architecture shown in FIGURE 5 is used.

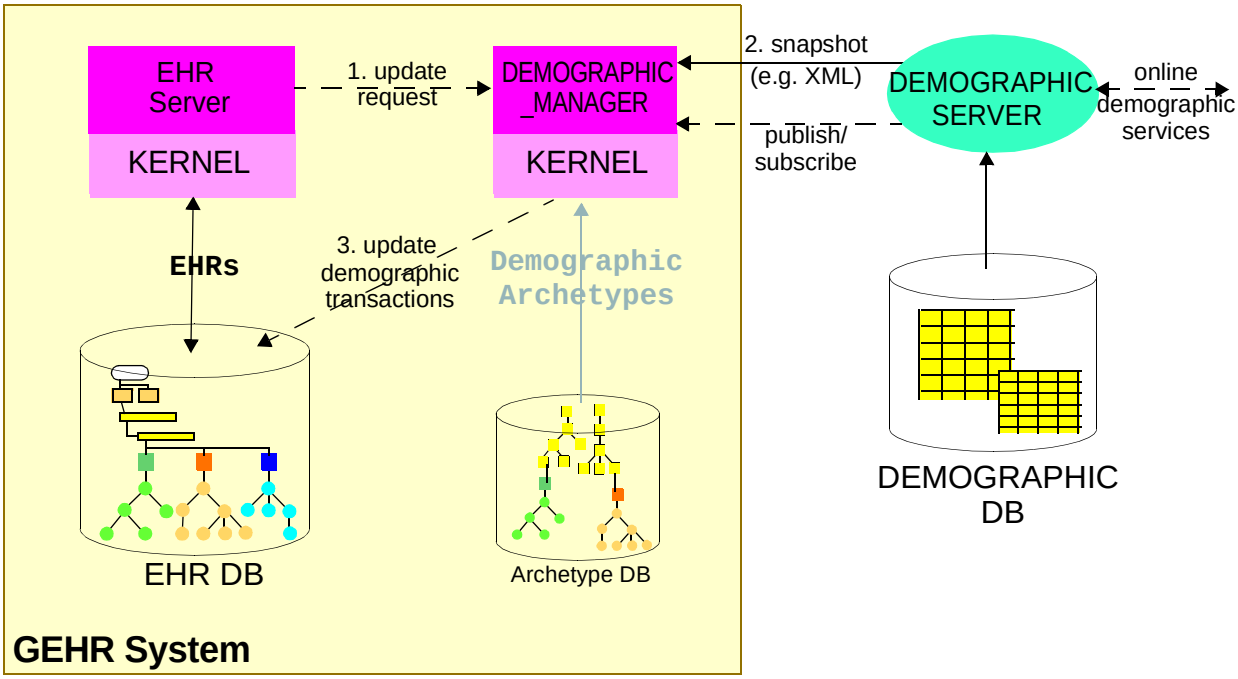


**FIGURE 5** Demographic Architecture

The existing demographic server, interface to an online demographic information is shown on the right hand side. The Demographic Manager application is used to import snapshots of demographic information from the server. This will occur on two occasions:

- When changes are being made to the patient's EHR, and demographic transactions need to be added to represent the demographic entities referenced in the changes.
- When changes occur in the PARTY records in the server, the relevant demographic transactions in all EHRs should be synchronised. This requires a publish/subscribe interface between the Demographic Server and the Demographic Manager.

Typically the conversion carried out by the Demographic Manager will follow the scheme: relational -> XML instance -> GEHR `VERSIONED_TRANSACTION` update.

## 4.2 Software Architecture

Demographic Manager illstrates the software architecture of the demographic manager. In a productin system, the demographic manager would be a separate process; in test systems or small systems, it can be incorporated into the main application.

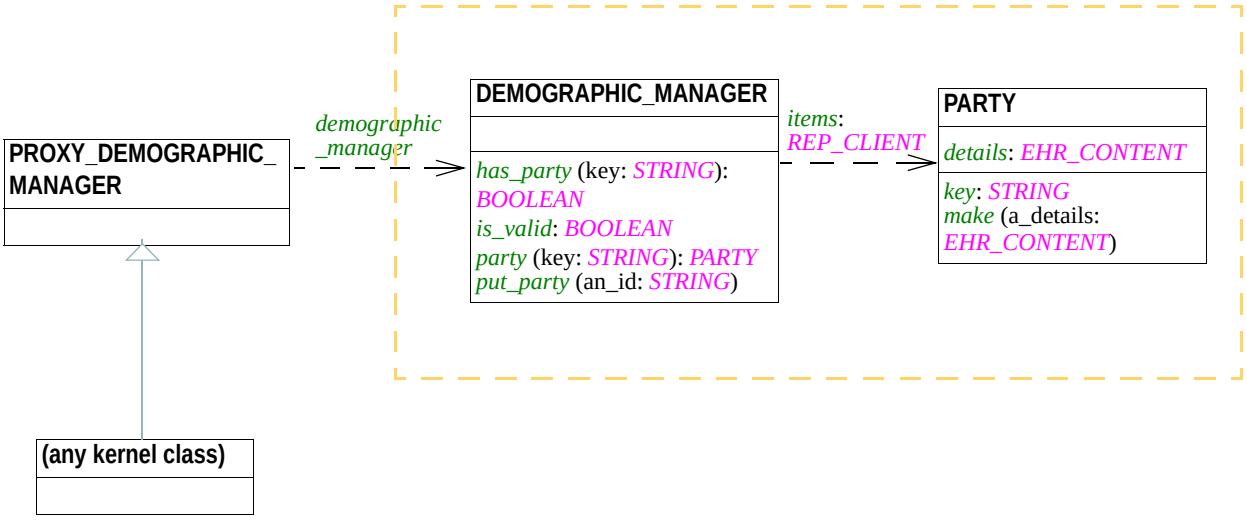


**FIGURE 6** Demographic Manager

# 5 Term Server Interface

## 5.1 System Architecture

## 5.2 Software Architecture

# 6 Constructing EHRs With Archetypes

## 6.1 System Architecture

FIGURE 7 illustrates an operational EHR system.

Archetype documents are authored offline, by the GEHR Foundation, national health institutions, local health care facilities and software vendors. They are accepted into the Archetype Domain System according to a defined process, and become available via Archetype Domain Servers, which are essentially intelligent file or document servers.

In an EHR system, there may be several GEHR applications, each of which requires a set of archetypes specific to its purpose, for example, diabetes, or a GP's general purpose application. The set of archetypes is defined by a "clinical concept/archetype equivalence table", which matches clinical concepts such as "blood pressure", "blood pressure with protocol", "prescription", "diabetic summary" and so on, with actual archetype names such as `"au-dhac.cont.bp.v2"` (Australian Department of Health & Aged Care Blood Pressure clinical content archetype version 2) or `"gehr.cont.bp.v1"` (GEHR simple blood pressure clinical content archetype, version 1).

```
To Be Determined:     these archetype names are invented for now - arche-
      type naming has not yet been decided
```

This system allows applications to refer to archetypes using their logical concept name, rather than hardwiring in actual archetype names.

A further advantage of the equivalence table is that it can be used by an archetype initialiser application to know in advance which archetypes will be required by a given application. By traversing the list of archetype names in the table, the initialiser can request each archetype from the local domain server, and for those which are found, parse the XML into kernel archetype object structures, which are stored into the Archetype database. The archetype initialiser is shown above as a separate application, but it could just as easily be a component of each application. One advantage of a separate application is that only one software entity has to be replaced if modifications to the XML parser are made.

The Archetype database contains the results of parsing: instances of the classes in the archetype cluster of the kernel. Their purpose is twofold: to act as builder/controller objects during the construction of EHR content, and to supply a content default content "template", enabling applications to instantly display something, which in many cases would be only slightly modified to arrive at the content required by the user. For example, the default content for a blood pressure need only have values inserted to create the desired result; a prescription would have values inserted, and possibly some optional items deleted.

Finally, EHR content is created when an application creating the relevant content factory (which acts as the current build context), and calling a content creation feature with a clinical concept name (as found in the equivalence table) as an argument. The content factory will then request the `ARCHETYPE_MANAGER` class to retrieve the named archetype object, which will in turn install a copy of the default content object into the current build context. The application will continue to make calls to the content factory in order to modify the content as required by the user, with each call being mediated by the content archetype. Attempts to make invalid modifications will fail. Eventually, the finished content item will be written into the current transaction, and the transaction committed to the EHR database.

The above architecture allows the Archetype Initialiser to be written in any language, since communication of archetype objects is via the database; most databases are language neutral, so the only

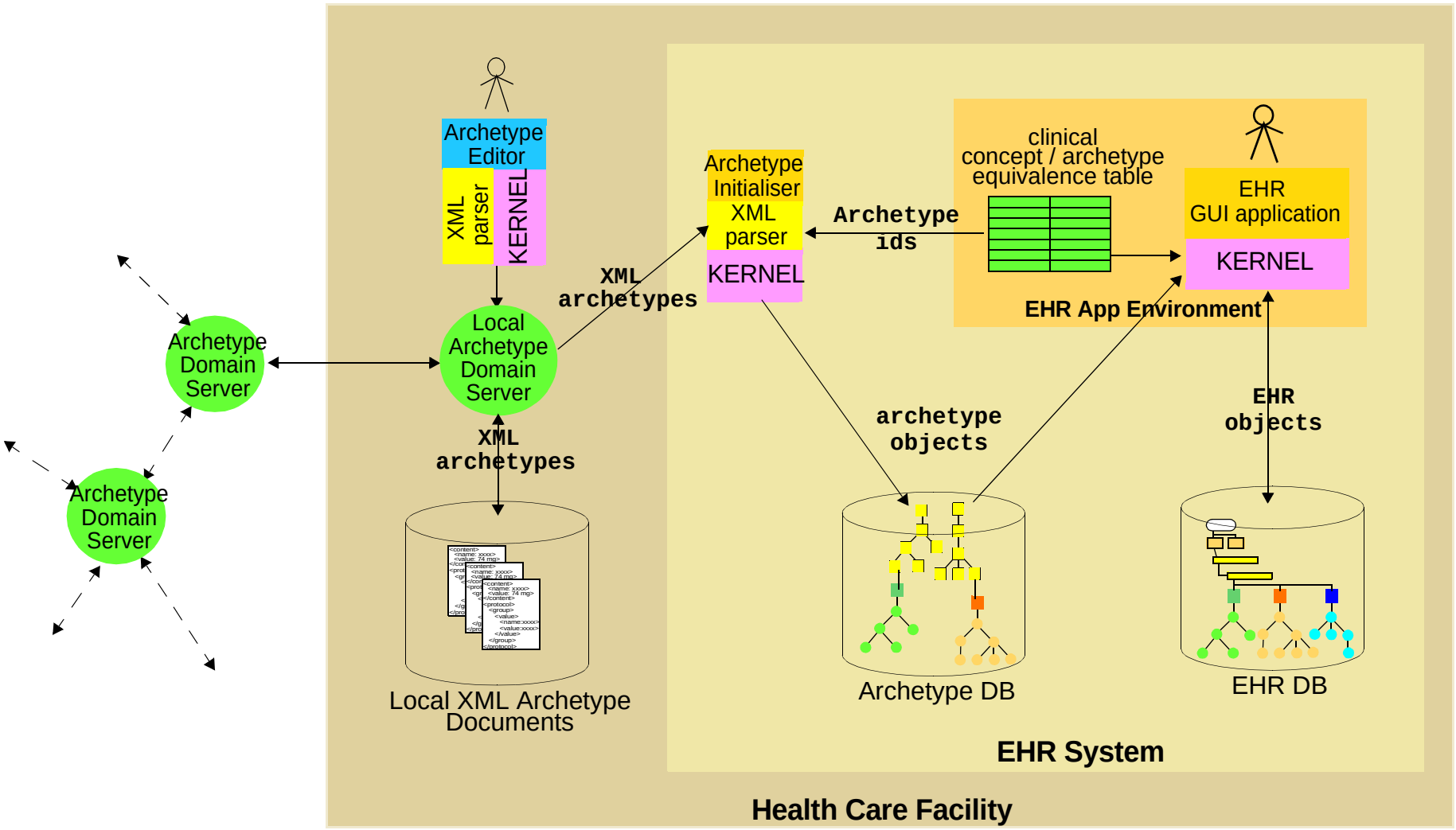


**FIGURE 7** Archetypes in an EHR system

requirement is that both the Initialiser and the application understand the database schema in the same way, which is achieved by assuming a common standard.

## 6.2 Software Architecture

### 6.2.1 Overview

This section deals with archetypes which have already been parsed and are available to the EHR system in native form. Archetype parsing and the archetype initialiser application are dealt with in Archetype Parsing on page 33.

The process of creating content with the kernel is designed to be very simple for the application programmer, but of course this mandates some sophistication behind the scenes. Archetypes add a further element of complexity, so we will explore the content construction process in some detail.

The technical problem of building content can be described as follows:

- Application requests to create a new transaction version, and is able to specify at least some details of the type of content required.
- Archetypes for each level of content - transaction (`EHR_CONTENT`), organiser (`ORGANISER_ROOT`) and content (`DEFINITION_CONTENT`) - need to be specified. Note that higher level archetypes may specify legal archetypes at lower levels, using a match pattern for names.
- The archetypes are retrieved from the object database.
- The application may need to be able to choose piece-wise which lower archetypes are to be used from the range of possible ones; an example is for content under SOAP orgqnisers. In other cases, there is a default which should be followed without intervention from the application; for example, dempgraphic content.
- Default content is created from the archetype, enabling the application to start with a sensible initial state, and reducing the work for the application (and thus, the user).
- The application then makes changes to the default content, with each change being checked in advance by the archetype, to ensure it is legal. Illegal changes are prevented before they can be made.
- When all changes are finished, the created content can be committed to a transaction, and thence to the database.

In terms of design, this process has been separated into two levels, namely:

1. Archetype retrieval, choosing and chaining of sub-archetypes, committal.
2. For a single archetype, content creation, modification and validity checking

This section deals with the first, while the section Archetype-governed Content Construction below deals with the detailed processing of archetypes.

### 6.2.2 Factories

To understand how the GEHR factories work, let us begin with a simple (yet sufficiently detailed) example of realistic EHR content, so that it is clear what content we want to create. Consider FIGURE 8 and FIGURE 9 together. These show the default and populated content of an `EHR_CONTENT` object, in a hierarchically exploded instance form, in which class and attribute names are in the familiar fonts from the UML class models in this document. With a couple of exceptions due to readability

concerns, the attribute and class names correspond exactly to GOM classes and attributes. (Note that the format here might be quite different fromthat used in a user-friendly GUI).

```
EHR_CONTENT                                          (gehr.trans.patient-core-demographics.v1)
concept: “patient demographics”
context:DEFINITION_CONTENT:                          (gehr.cont.demographic-snapshot.v1)
        concept = “demographic snapshot”
        proposition: HIERARCHICAL_PROPOSITION
                name = “demographic snapshot” (Gehr_clin_1.0::0200)
                item: HG
                        values: HV; “demographic server id” = “default”
                                HV; “last changed” = 01/01/1800
                        groups:HG; “address” (Gehr_clin_1.0::0005)
                                values: HV; “street number” (Gehr_clin_1.0::0240) = “default”
                                        HV; “street name” (Gehr_clin_1.0::0241) = "default"
                                        HV; “locality” (Gehr_clin_1.0::0242) = "default"
                                        HV; “county” (Gehr_clin_1.0::0243) = "default"
                                        HV; “postcode” (Gehr_clin_1.0::0244) = "default"
                                        HV; “country” (Gehr_clin_1.0::0245) = "default"
content:ORGANISER_ROOT                               (gehr.org.patient-core-demographics.v1)
        concept:“patient core demographics”
        organisers:ORGANISER:
                        name: “core demographics” (Gehr_clin_1.0::0238)
                        content: DEFINITION_CONTENT          (gehr arch id = gehr.cont.identity-path.v1)
                                concept = “identity path”
                                proposition: SIMPLE_PROPOSITION
                                        name = “identity path”
                                        item: HG
                                                values: HV; “path” (Gehr_clin_1.0::0260) = "name"
                ORGANISER:
                        name: “identity” (Gehr_clin_1.0::0236)
                        content: DEFINITION_CONTENT:         (gehr arch id = gehr.cont.patient-identity.v1)
                                concept = “patient identity”
                                proposition: HIERARCHICAL_PROPOSITION
                                        name = “patient identity” (Gehr_clin_1.0::0209)
                                        item: HG
                                                values: HV; “name” (Gehr_clin_1.0::0218) = “unknown”
                                                        HV; “date of birth” (Gehr_clin_1.0::0214) = 01/01/1800
                                                        HV; “place of birth” (Gehr_clin_1.0::0215) = “unknown”
                                                        HV; “sex” (Gehr_clin_1.0::0216) = “not specified” (Gehr_clin_1.0::0254)
                ORGANISER
                        name: “contacts” (Gehr_clin_1.0::0237)
                        content:DEFINITION_CONTENT:          (gehr arch id = gehr.cont.party-contacts.v1)
                                concept = “party contacts”
                                proposition: HIERARCHICAL_PROPOSITION
                                        name = “party contacts”  (Gehr_clin_1.0::0210)
                                        item: HG
                                                groups:HG; “address” (Gehr_clin_1.0::0005)
                                                        values: HV; “street number” (Gehr_clin_1.0::0240) = “default”
                                                                HV; “street name” (Gehr_clin_1.0::0241) = "default"
                                                                HV; “locality” (Gehr_clin_1.0::0242) = "default"
                                                                HV; “county” (Gehr_clin_1.0::0243) = "default"
                                                                HV; “postcode” (Gehr_clin_1.0::0244) = "default"
                                                                HV; “country” (Gehr_clin_1.0::0245) = "default"
```

**FIGURE 8** Archetype-generated Default Content

The intention is to construct the content shown in FIGURE 9, in this case, the content of a patient demographic snapshot transaction. Note that there is content at all three levels at which GEHR archetypes can be applied, namely, EHR_CONTENT, ORGANISER_ROOT and DEFINITION_CONTENT; in each case the actual archetype name that was used is shown in red.

Stepping back slightly in time, FIGURE 8 shows us the default content, created by the relevant archetype objects, ready to have actual values populated. (In this example, the difference between the default and actual is simply that values have been inserted, but, in general the structure could also have been changed, for example certain address fields added or removed. We deal with such complexity in the following section).

Now, in order to arrive at the point where the content of FIGURE 8 exists, the relevant archetypes need to have been chosen and retrieved, and default content creating routines called.

The starting point for constructing EHR content is the factory, which performs the retrieval of archetypes, default content creation, and committal of transactions to the EHR. The factory class hierarchy is illustrated in FIGURE 10.

The important points to note here are:

- All descendants of ARCHETYPED_FACTORY have attributes for:
  - *sub_archetype_id_patterns*: a table of {attribute name, archetype pattern id} pairs. In this context, "attribute name" refers to an attribute in the GOM type being archetyped. For example, in EHR_CONTENT, there are two attributes, *content* and *context*, which both point to objects which are archetyped. Thus, EHR_CONTENT_FACTORY.*sub_archetype_id_patterns* is a table of the form {{"/content", "???\.org\.???"}, {"|context", "???\.cont\.???"}}. The patterns may be quite complex, and can include disjoint archetype names, as in "gehr\.org\.party-identity\..*|gehr\.org\.demographic\..*\..*".
  - *selected_sub_archetypes*: a set of actual archetype ids, again keyed by attribute name: this is the result of a choosing process by either the application, or by some automatic means. Each selected archetype id is stored in a table, keyed by its runtime path.
  - *sub_factories*: a set of the factory objects created once archetypes have been chosen. For example, if the archetype gehr.cont.demographic-snapshot.v1 was chosen for EHR_CONTENT.*context*, then a DEFINITION_CONTENT_FACTORY would appear in this table, keyed by "context".
- ARCHETYPED_FACTORY descendants all have a reference to an ARCHETYPE object, whose type co-varies with the factory appropriately.
- ARCHETYPE objects contain a reference to an ARCHETYPED object (once *create_default* has been called), again with dynamic types covarying. For example, an A_ORGANISER_ROOT.*target* points to a ORGANISER_ROOT object.

We will now follow the sequence of calls to factories with illustrations of the state of th revelant objects over time.

The first step is to create a EHR_CONTENT_FACTORY, and retrieve an archetype thus:

```
local
    ef:EHR_CONTENT_FACTORY
do
    create ef
    ef.retrieve_archetype (data_factory.create_term_text_from_expansion ("patient
demographics", Gehr_clinical_ts))
```

EHR_CONTENT (gehr.trans.patient-core-demographics.v1)
*concept*: “patient demographics”
*context*:DEFINITION_CONTENT: (gehr arch id = gehr.cont.demographic-snapshot.v1)
*concept* = “demographic snapshot”
*proposition*: HIERARCHICAL_PROPOSITION
*name* = “demographic snapshot” (Gehr_clin_1.0::0200)
*item*: HG
*values*: HV; “demographic server id” = **“server2.mnc.com.au”**
HV; “last changed” = **13/08/2000**
*groups*:HG; “address” (Gehr_clin_1.0::0005)
*values*: HV; “street number” (Gehr_clin_1.0::0240) = **“22”**
HV; “street name” (Gehr_clin_1.0::0241) = **"Holden St"**
HV; “locality” (Gehr_clin_1.0::0242) = **"Mooloolah"**
HV; “county” (Gehr_clin_1.0::0243) = **"Qld"**
HV; “postcode” (Gehr_clin_1.0::0244) = **"4553"**
HV; “country” (Gehr_clin_1.0::0245) = **"Australia"**
*content:*ORGANISER_ROOT (gehr.org.patient-core-demographics.v1)
*concept*:“patient core demographics”
*organisers*:ORGANISER*:*
*name*: “core demographics” (Gehr_clin_1.0::0238)
*content*: DEFINITION_CONTENT (gehr arch id = gehr.cont.identity-path.v1)
*concept* = “identity path”
*proposition*: SIMPLE_PROPOSITION
*name* = “identity path”
*item*: HG
*values*: HV; “path” (Gehr_clin_1.0::0260) = **"/"core demographics"/"identity"|"patient identity"|"name""**
ORGANISER:
*name*: “identity” (Gehr_clin_1.0::0236)
*content*: DEFINITION_CONTENT: (gehr arch id = gehr.cont.patient-identity.v1)
*concept* = “patient identity”
*proposition*: HIERARCHICAL_PROPOSITION
*name* = “patient identity” (Gehr_clin_1.0::0209)
*item*: HG
*values*: HV; “name” (Gehr_clin_1.0::0218) = **“Sarah McMahon”**
HV; “date of birth” (Gehr_clin_1.0::0214) = **03/04/1959**
HV; “place of birth” (Gehr_clin_1.0::0215) = **“Ballarat”**
HV; “sex” (Gehr_clin_1.0::0216) = **“female”** (**Gehr_clin_1.0::0251**)
ORGANISER
*name*: “contacts” (Gehr_clin_1.0::0237)
*content*:DEFINITION_CONTENT: (gehr arch id = gehr.cont.party-contacts.v1)
*concept* = “party contacts”
*proposition*: HIERARCHICAL_PROPOSITION
*name* = “party contacts” (Gehr_clin_1.0::0210)
*item*: HG
*groups*:HG; “address” (Gehr_clin_1.0::0005)
*values*: HV; “street number” (Gehr_clin_1.0::0240) = **“142”**
HV; “street name” (Gehr_clin_1.0::0241) = **"Platypus Lane"**
HV; “locality” (Gehr_clin_1.0::0242) = **"Mooloolah"**
HV; “county” (Gehr_clin_1.0::0243) = **"Qld"**
HV; “postcode” (Gehr_clin_1.0::0244) = **"4553"**
HV; “country” (Gehr_clin_1.0::0245) = **"Australia"**

**FIGURE 9** Populated Content

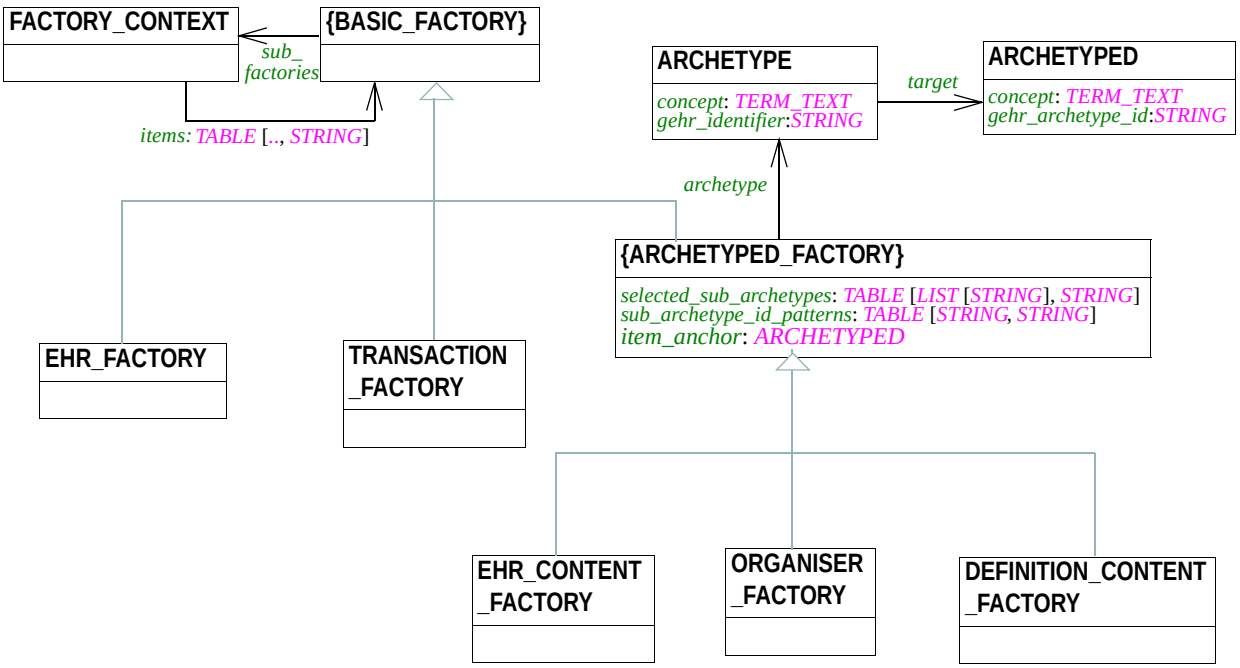


**FIGURE 10** Factory Classes

The archetype is identified as “patient demographics”; it is already known in the system that this concept name is mapped to the GEHR archetype id `"gehr.trans.patient-core-demographics.v1"`. The result of these calls is shown in FIGURE 11.

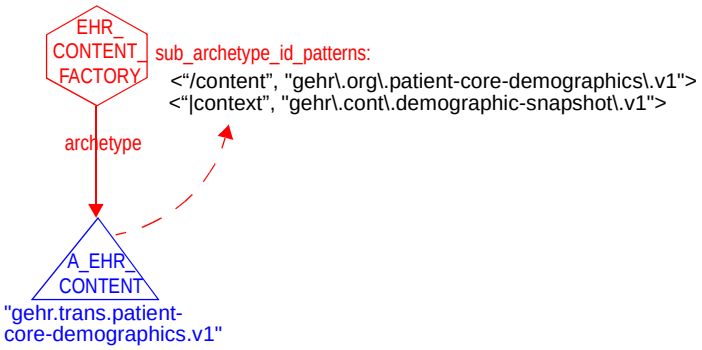


**FIGURE 11** EHR_CONTENT Factory Creation

Not only has the archetype object (`A_EHR_CONTENT`) been retrieved, its valid sub-archetype ids have been used to populate the table *sub_archetype_id_patterns*. The meaning of the first entry in this table is: valid archetypes for building the content at `EHR_CONTENT`.*content* must match the pattern `"gehr\.org\.patient-core-demographics\.v1"`. Note that this pattern has no variability - only the `ORGANISER` archetype name `"gehr.org.patient-core-demographics.v1"` will match. The pseudo-path `“/content”` is used because the actual path cannot be known until the archetype is chosen (the path can then be determined from its concept; we do however know from the GOM that it will be an `ORGANISER_ROOT`, hence the organiser-style “/”).

The next step is to select sub-archetypes for each sub-archetyped attribute, by proposing valid archetype ids, thus:

```
ef.select_sub_archetype("gehr.org.patient-core-demographics.v1", "/content", "")
ef.select_sub_archetype("gehr.cont.demographic-snapshot.v1", "|context", "")
ef.install_sub_archetypes
```

Here, archetypes have been selected for *content* and *context*. The call *install_sub_archetypes* causes the relevant factory objects to be created, and their archetypes to be retrieved, as illustrated in FIGURE 12.

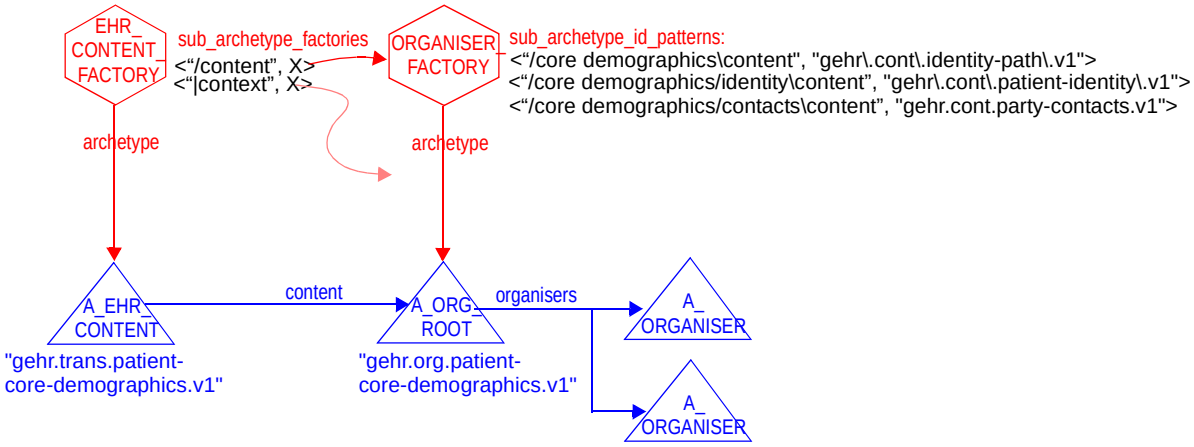


**FIGURE 12** After calling *install_sub_archetypes* on EHR_FACTORY

Once again, we need to specify and install archetypes for content, the next level down, thus:

```
local
    of:ORGANISER_FACTORY
do
    ...
    of ?= ef.sub_factories("/core demographics")
    of.select_sub_archetype("gehr.cont.identity-path.v1", "/core demographics|content",
"/%"core demographics%"")
    of.select_sub_archetype("gehr.cont.patient-identity.v1", "/core
        demographics/identity|content", "/%"core demographics%"/%"identity%"")
    of.select_sub_archetype("gehr.cont.party-contacts.v1", "/core
        demographics/contacts|content", "/%"core demographics%"/%"contacts%"")
    of.install_sub_archetypes
```

These calls lead us to the situation shown in FIGURE 13. Now we have all the factory objects we need, chained together by their *sub_archetype_factories* reference tables, and the archetype objects have all been retrieved, so at this point we are ready to start creating content. This is done with a single call to the `EHR_CONTENT_FACTORY`, as follows:

```
    ef.create_default
```

A cascade of creation is now set in train, which causes the construction of a complete set of default object structures, whose contents were shown in FIGURE 8. The object view is shown in FIGURE 14. The creation takes place recusrsively, with each factory calling *create_default* on its own archetype objects, then calling *create_default* on the sub-archetype factories, which continue the pattern. At this point, all content objects, along with associations shown in solid lines on FIGURE 14 are in place. When *create_default* returns for all sub-archetype factories, each factory calls *create_default_finalise* on its archetype, and this continues back up the chain. These calls enable the archetypes to “glue” together the targets of the archetypes of sub-archetype factories, to the target of their own archetype - the links shown as dashed lines.

The actual details of content creation of each content object, and its underlying data structures, for example `DEFINITION_CONTENT` and `HIERARCHICAL_PROPOSITION` and associated classes is dealt with in detail in the following section.

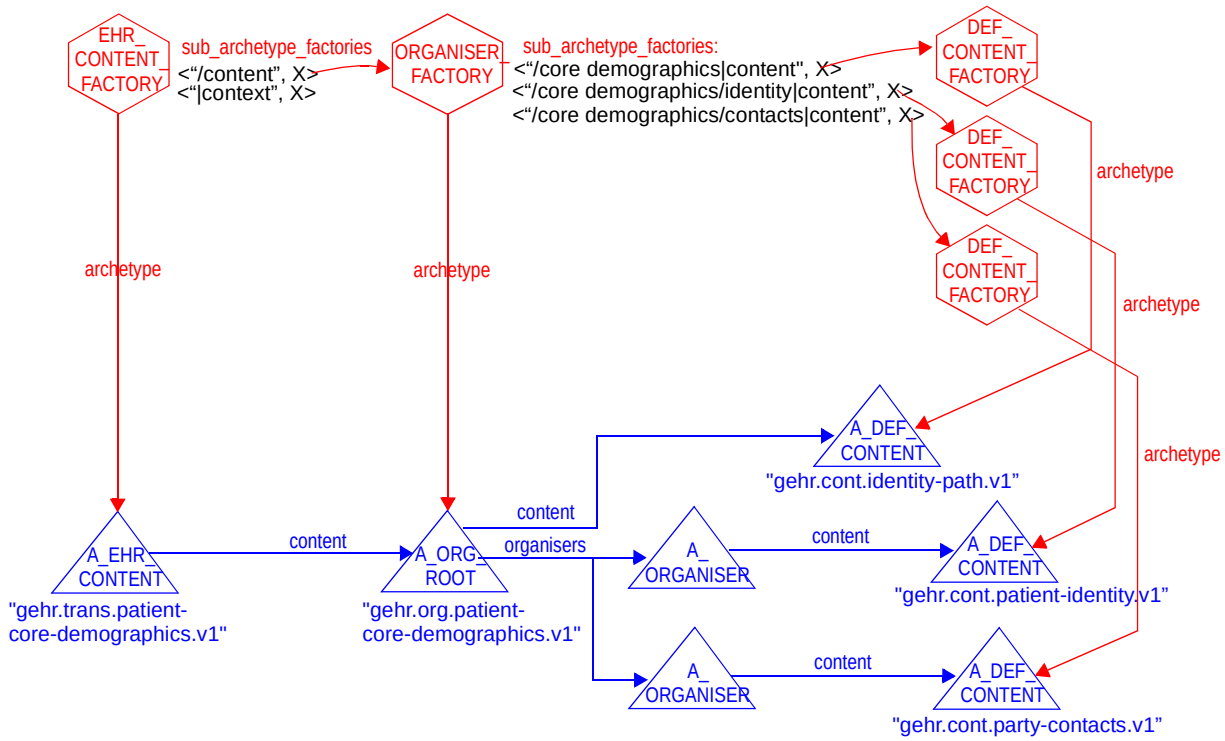


**FIGURE 13** After calling *install_sub_archetypes* on ORGANISER_FACTORY

At this point, around ten calls have enabled the creation of a default structure and content for a complete patient demographic snapshot. A number of calls are now needed to populate the structure. There are various possibilities for this, since both the ORGANISER and XXX_CONTENT classes provide an interface designed to support different flavours of application. The following set of calls is one possibility for getting from the default content to fully populated content illustrated in FIGURE 9.

```
local
    ef:EHR_CONTENT_FACTORY
    of:ORGANISER_FACTORY
    dcf:DEFINITION_CONTENT_FACTORY
    hp:HIERARCHICAL_PROPOSITION
    a_date:DATE_IMPL
do
    ...
    dcf ?= of.sub_factory("/%"core demographics%"|%"identity path%"")
    hp := dcf.item_proposition
    hp.value_go_to_name("path")
    hp.value_replace_data_value(data_factory.create_plain_text("/%"core
              demographics%"/%"identity%"|%"patient identity%"|%"name%""))

    dcf ?= of.sub_factory("/%"core demographics%"/%"identity%"|%"patient identity%"")
    hp := dcf.item_proposition

    hp.value_go_to_name("name")
    hp.value_replace_data_value(data_factory.create_plain_text("Sarah McMahon"))

    hp.value_go_to_name("date of birth")
    create a_date.make_from_string("04/03/1959")
    hp.value_replace_data_value(a_date)

    hp.value_go_to_name("place of birth")
    hp.value_replace_data_value(data_factory.create_plain_text("Melbourne"))
```

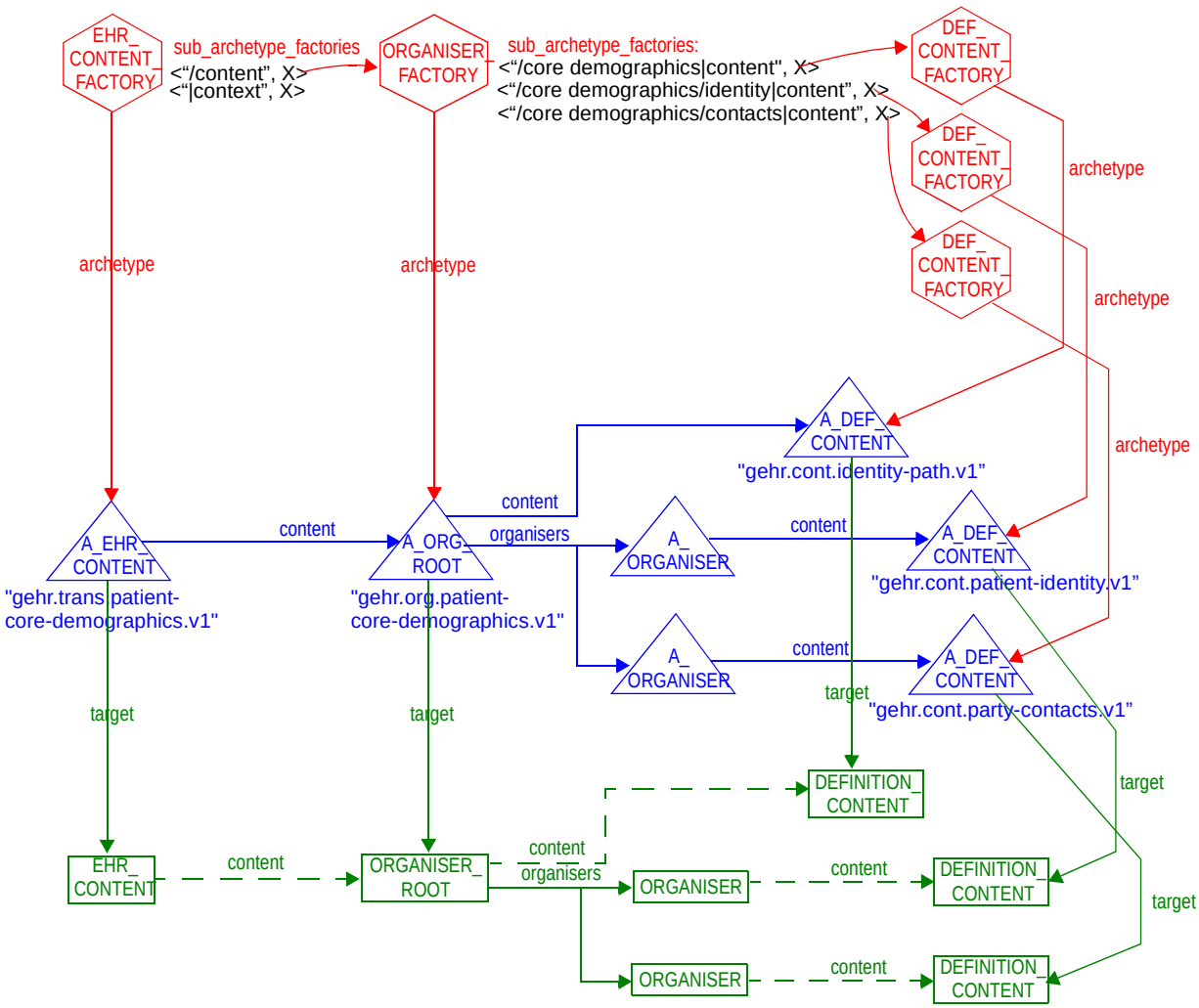


**FIGURE 14** Content Creation

```
hp.value_go_to_name("sex")
hp.value_replace_data_value(data_factory.create_term_text("female",
      Gehr_clinical_ts))
dcf ?= of.sub_factory("/%"core demographics%"/%"contacts%"|%"party contacts%"")
hp := dcf.item_proposition
hp.set_cursor_to_path("|%"party contacts%"|%"address%"")
hp.value_go_to_name("street number")
hp.value_replace_data_value(data_factory.create_plain_text("142"))
hp.value_go_to_name("street name")
hp.value_replace_data_value(data_factory.create_plain_text("Platypus Lane"))
hp.value_go_to_name("locality")
hp.value_replace_data_value(data_factory.create_plain_text("Mooloolah"))
hp.value_go_to_name("county")
hp.value_replace_data_value(data_factory.create_plain_text("Queensland"))
hp.value_go_to_name("postcode")
hp.value_replace_data_value(data_factory.create_plain_text("4553"))
hp.value_go_to_name("country")
hp.value_replace_data_value(data_factory.create_plain_text("Australia"))
```

All content construction in the GEHR kernel functions according to the general scheme just described, with more or less complexity depending upon the particular archetypes involved.

## 6.3 Issues and limitations

### 6.3.1 Chaining by Archetype Id

One limitation with the above is that archetypes are chained by virtue of sub-archetype id patterns in each archetype. It could be argued that in fact rather than ids, concept names should be used to do chaining, since the archetype equivalence table is used by the application to decide what particular archetypes to use. Chaining by ids circumvents this table. The contrary argument is that the designers of archetypes are quite clear and precise about what sub-archetypes should be used, and do not want to compromise the integrity of their design by a badly configured equivalence table. Currently the latter argument seems to hold more weight, since informational integrity and quality is probably more important than configurability.

### 6.3.2 Id Matching

In any case, archetype id matching poses another technical problem: it requires that the archetype naming system be designed such that lexical name matching be congruent to the intended clinical matches. For example, the archetype id `"nhs.cont.drug-medication-order.v5"` matches the pattern `".*\.cont\..*medication-order.*\..*"`, but equivalent archetypes from other medical systems such as homeopathy may not. The simple way of dealing with this is to use patterns with disjoint matches, using the '|' regular expression element. However it is unclear what the impact on requirements for archetype identification is of using ids to match clinically equivalent archetypes.

### 6.3.3 Default Archetype Choices

In the above example, the factory function *select_sub_archetype* was used to choose the ids of sub-archetypes. However, in the example case, clearly there was no choice intended - the pattern was fixed. Currently the factory model does not detect this specifically, but it probably should be provided, since it would be convenient to avoid having to make such fixed choices, and go straight to content creation.

# 7 Archetype-governed Content Construction

## 7.1 Software Architecture

FIGURE 15 illustrates the class model for the kernel archetype classes. The salient features are as follows.

The archetype class model is an almost complete homologue of the GOM, i.e. each GOM class has an archetype class equivalent, which is designed to express the *possible constraints* on instances of the GOM class. Archetype classes are named by prepending A_ before the class name of their corresponding GOM class. Thus the archetype classes, A_EHR_CONTENT, A_ORGANISER, and A_DEFINITION_CONTENT define the constraints for the corresponding classes in the GOM, namely EHR_CONTENT, ORGANISER and DEFINITION_CONTENT. A_DEFINITION_CONTENT is subtyped to A_PREDICATE_CONTENT, A_SUBJECTIVE_CONTENT etc, in the same way as the corresponding GOM classes. A_EHR_CONTENT and A_ORGANISER each have a feature which lists allowed archetypes of the next lower down type.

To Be Determined: this may be removed in the future in favour of exclusive use of composite archetypes

The classes A_HIERARCHICAL_XX define constraints for their namesakes in the content cluster of the GOM. Things to note in particular:

- A_HIERARCHICAL_GROUP has *values* and *groups* attributes, defining legal structure of the content under each node; the attributes *new_group* and *new_value* refer to objects of type A_HIERARCHICAL_GROUP and A_HIERARCHICAL_VALUE, repectively, which define the constraints on new HIERARCHICAL_GROUPs or HIERARCHICAL_VALUEs added to the HIERARCHICAL_GROUP represented by the current A_HIERARCHICAL_GROUP node.
- Value constraints are defined by the attribute *type_value_constraints* in A_HIERARCHICAL_VALUE, and the class A_VALUE_CONSTRAINT and descendants.
- Each created structure must know the precise name of the archetype it was created from.

## 7.2 Scenarios

### 7.2.1 Content Construction

A slightly simplified process of content construction is illustrated in FIGURE 16.

The sequence of events is as follows:

- An application class requires an item of content to be created, such as a blood pressure. To start the process, it creates a OBSERVATION_CONTENT_FACTORY (since blood pressure will normally occur as a measured value in an EHR), which provides a build context in which content under construction can be assembled, and the relevant archetype can be attached.
- It calls OBSERVATION_CONTENT_FACTORY.*create_content*, passing a clinical concept as an argument;
- The clinical concept is used as a key into the equivalent table to find an archetype identifier; using this, the archetype is retrieved from the archetype database (causing the archetype objects to be created in the kernel program space). The retreived archetype - of subtype A_OBSERVATION_CONTENT in this example - is attached to OBSERVATION_CONTENT_FACTORY.*archetype*, making it available to the build process.

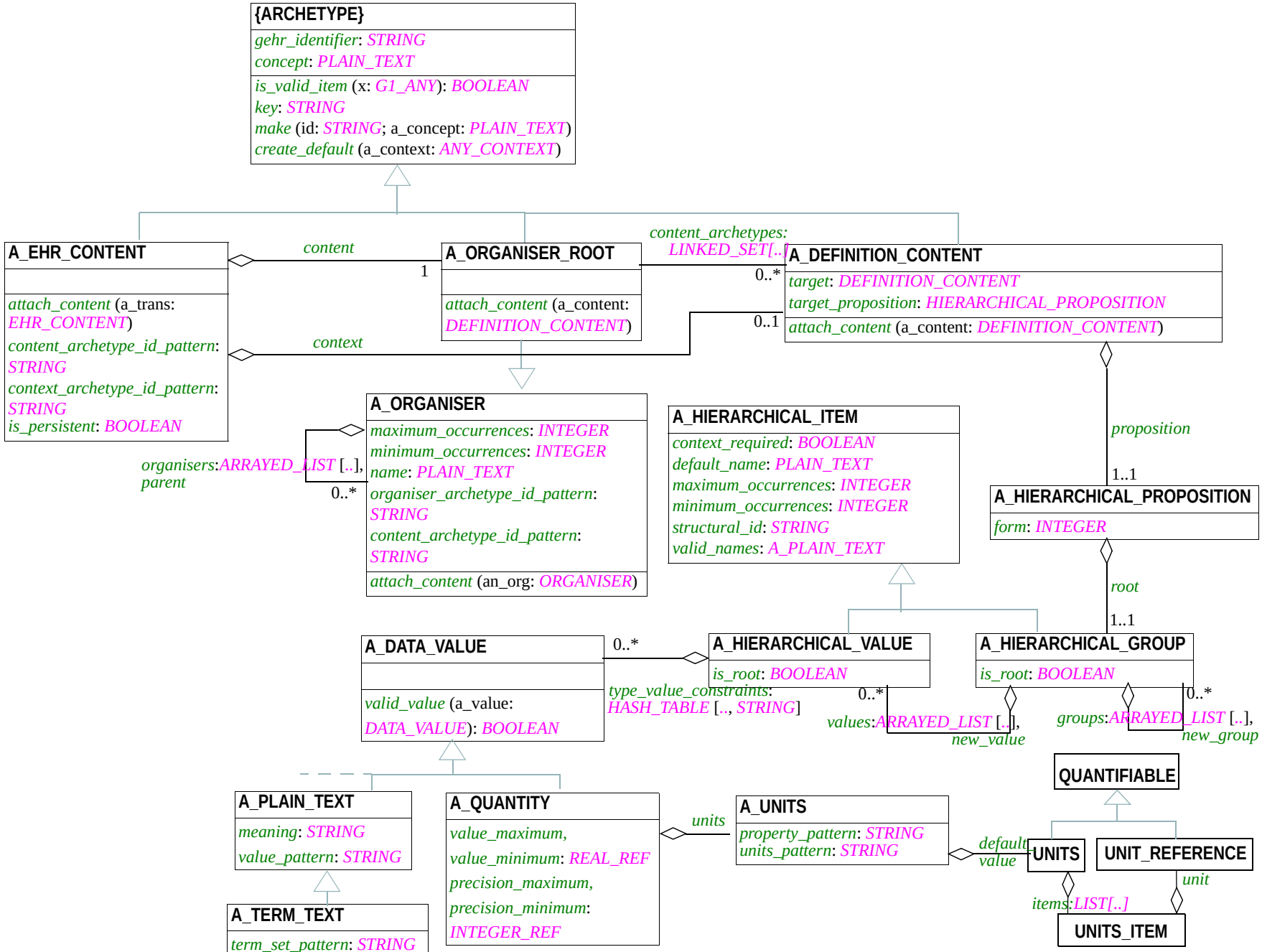


**FIGURE 15** Class Model for Archetypes

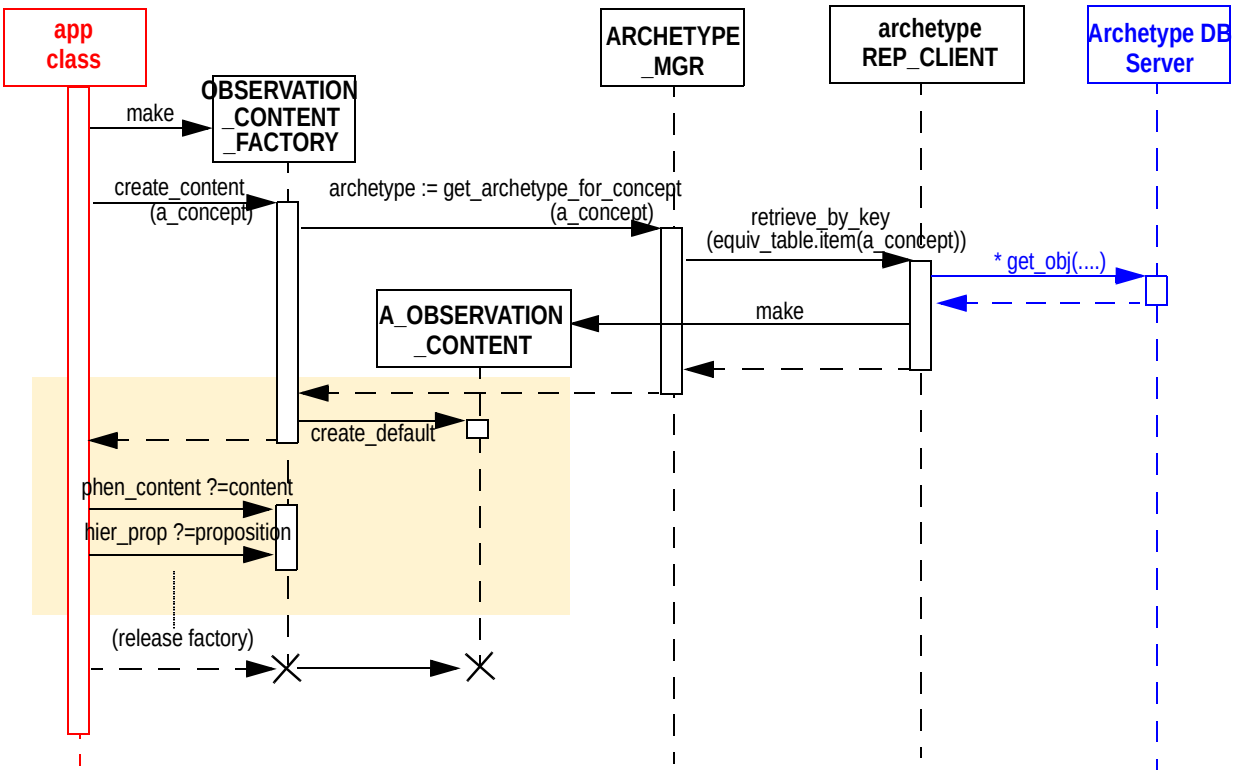


**FIGURE 16** Content construction with archetype

- Initial content is created when the content factory calls A_OBSERVATION_CONTENT.*create_default*.
- The application code can now access the variables *content*:OBSERVATION_CONTENT and *proposition*:HIERARCHICAL_PROPOSITION in the factory object. The former is the root object of the content tree being created, and the latter is its actual data which may vary, according to the different GOM forms available, as follows:
  - SIMPLE_PROPOSITION
  - LIST_PROPOSITION
  - HIERARCHICAL_PROPOSITION
  - TIME_SERIES
  - REGULAR_TIME_SERIES
  - TABLE_PROPOSITION
  - MATRIX_PROPOSITION
  - etc

These interfaces allow the application to build the content according to its requirements. All subsequent modification of content is mediated behind the scenes by archetype objects (shaded area in lower left of FIGURE 16). This is illustrated in detail in FIGURE 17.

The effect of A_DEFINITION_CONTENT.*create_default* is to recursively visit each object in the archetype structure, creating a corresponding default content object on the way, and setting its *archetype_item* reference back to the relevant archetype object.

It is via this link that further calls to HIERARCHICAL_PROPOSITION (or one of its descendants) can determine valid changes after the default creation, or after retrieval of an existing piece of content. Preconditions on content factory functions are used to determine whether each call would be valid according to the attached archetype.

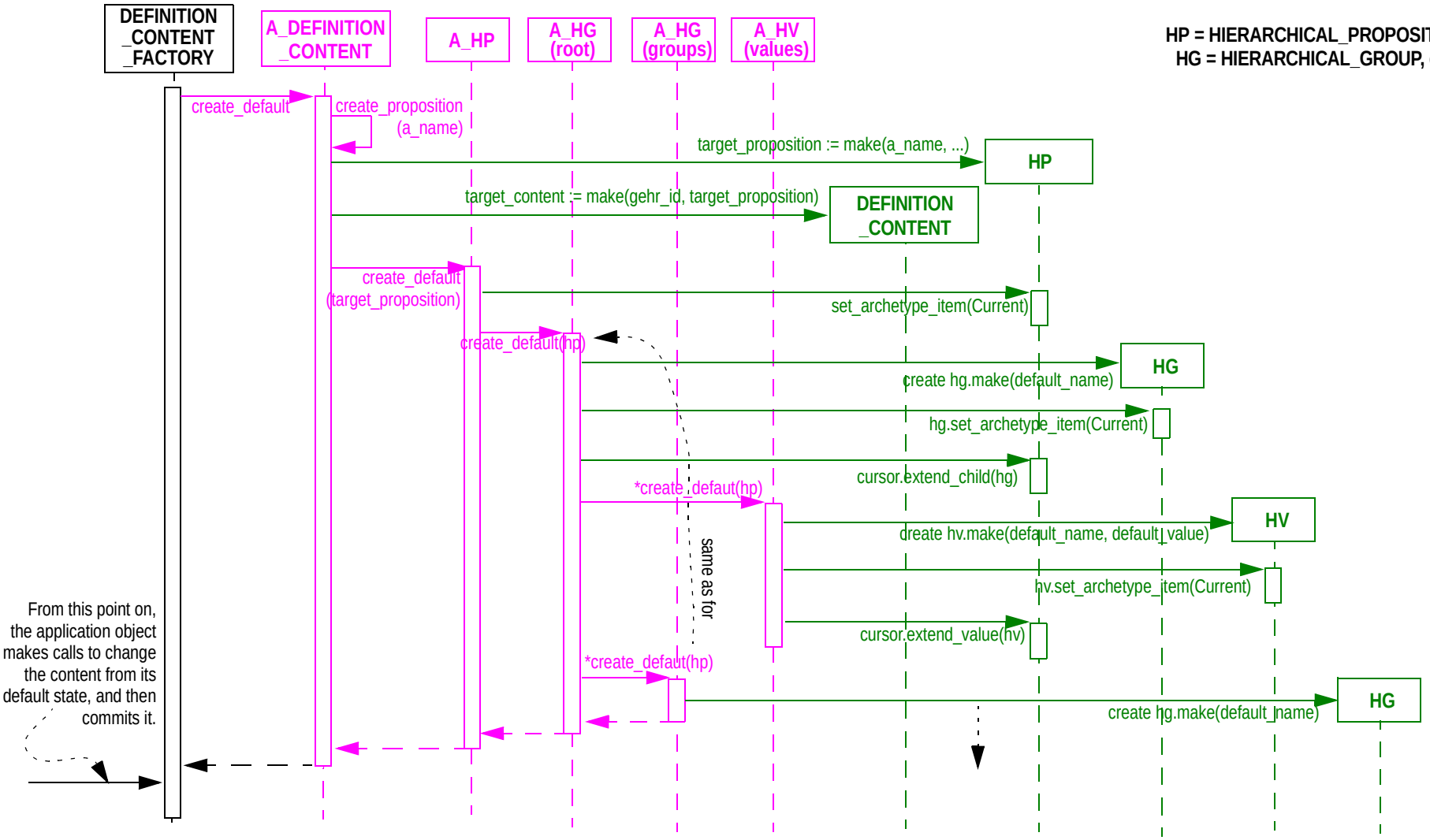


**FIGURE 17** Default Creation by Content Archetype

# 8 Archetype Parsing

FIGURE 18 illustrates the parsing of an archetype instance document by the archetype initialiser. The sequence of events is as follows:

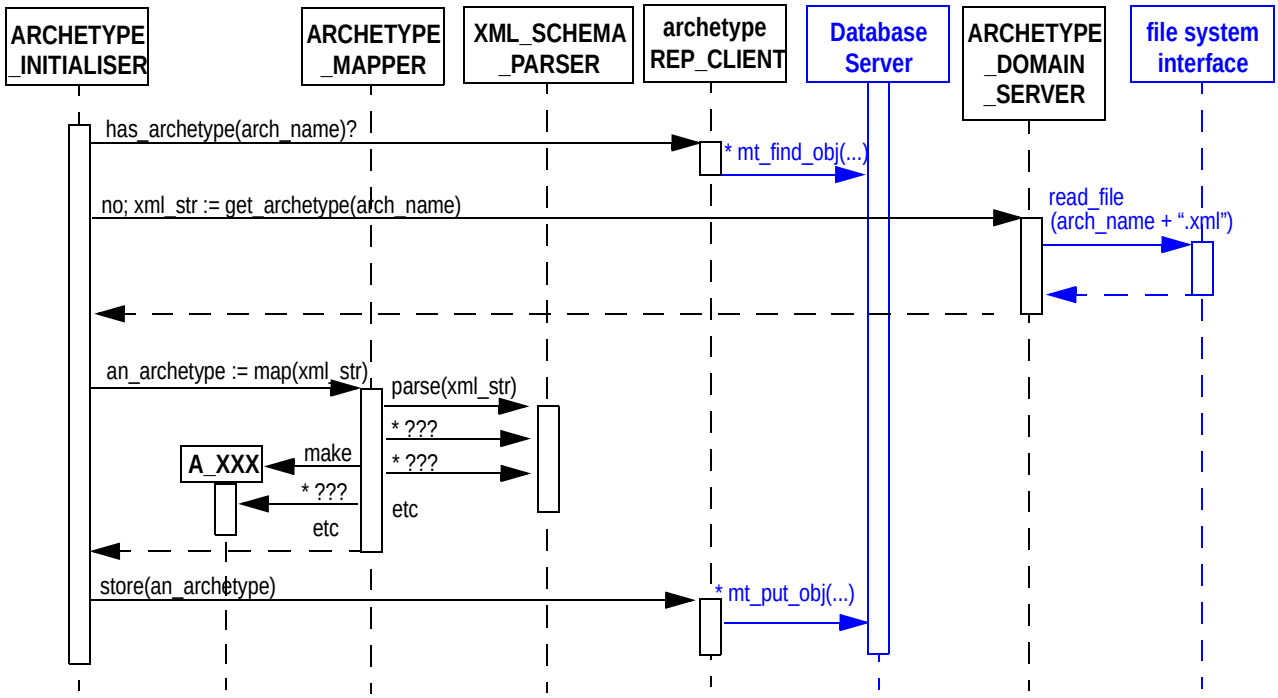


**FIGURE 18** Archetype Parsing

- The initialiser process reads an archetype name from the equivalence table.
- It checks if any archetype with the name already exists in the Archetype database.
- If not, it then communicates with the nearest archetype domain server, which retrieves an archetype document locally or obtains it from another server.
- It then parses the document and writes the resulting objects into the Archetype database.

To Be Determined: error processing here - what if no archetype document found; what if found but parsing fails; what if domain server down; what if network down (preventing contact with other domain servers)

In general, XML archetype documents will be used to generate Eiffel archetype structures, whose job it is to create EHR transaction, organiser and content structures. The Eiffel archetype structures are made persistent, allowing them to be re-used directly in their object form.

# 9 Persistence

# 10 Import/Export

## 10.1 XML

## 10.2 CORBA

# 11 User Interface

**END OF DOCUMENT**