The *open*EHR Modelling Guide

Authors:T Beale[1]

Revision: 1.1.1

Pages: 17

1. Ocean Informatics Australia

Amendment Record

Issue	Details	Who	Completed
1.1.1	CR-000041. Visually differentiate primitive types in openEHR documents. CR-000013. Rename key classes, according to CEN ENV 13606. Add explanation of qualified associations, existence, cardinality.	D Lloyd, D Kalra, T Beale	04 Oct 2003
1.1	Kestral Australia review.	G Grieve	08 Mar 2003
1.0	Adapted from openEHR EHR Reference Model document	T Beale	10 May 2002

Acknowledgements

The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

1 Introduction

Since the primary users of the formal specifications in health information standards are software developers and information systems builders, it is crucial that the models presented are comprehensible and implementable by technical people. In order to ensure implementability, all models specified by *open*EHR have been formally expressed and compiled using the Eiffel language, which is the semantically closest fit for UML among current languages.

In terms of scope, it is important to understand that the *open*EHR models is not that of a full application or component architecture; it is instead what software engineers call an *analysis model* - a formalisation of a number of inputs to the analysis and design process, namely requirements, design philosophy, and an ontological analysis of the domain. It is thus an appropriate *starting point* for constructing software, database schemas, exchange definitions and as such, an appropriate target for standardisation.

In terms of the ISO five viewpoint model for defining Open Distributed Processing systems (Open Distributed Processing Reference Model 1990), the EHR Reference Model corresponds to the Information Viewpoint. It is deliberately independent of specific engineering approaches that might be taken to implement an EHR or FHR service and provides an unambiguous and rigorous reference point to ensure the consistency (and therefore the interoperability) of different implementations.

A number of the design principles described in the document “Design Principles for the EHR” are pertinent to the kind of modelling carried out in *open*EHR.

2 Formalisms

2.1 UML

The *open*EHR models are shown in UML (Unified Modelling Language) [5] and have been formally validated using the Eiffel language, which strangely, is still one of the only reliable tool for specifying and fully validating object-oriented models. UML is an industry-standard modelling language, which has been formally defined by the OMG. The *open*EHR models make heavy used of two powerful UML semantics, namely:

- Generic classes ("template classes" in C++)
- Contracts, i.e. pre-conditions, post-conditions, invariants (defined in the OMG Object Constraint Language, OCL)

The notation used in this document follows the UML version 1.3 (see [5]). The following sections describe the major semantic constructs in the class diagrams in this document. Refer to Meyer [8] for a definitive guide to object-oriented semantics.

2.1.1 Package

A collection of related classes, grouped for convenient management of development. Packages may be nested hierarchically. Indicated graphically by a named blue rectangle containing classes.

2.1.2 Class

The primary construct in object-oriented modelling and software development. A class defines objects in terms of *behaviour* and *state*, or in more technical terms, routines and attributes. The class definition is the template for creating *objects* at runtime, which are *instances* of the class.

2.1.3 Inheritance

Inheritance is a relationship between classes in which the definition of the descendant (inheriting class) is based on the ancestor. The descendant may change the ancestor's definition in certain ways, according to the rules of the formalism. Inheritance is not normally visible at runtime as a relationship between objects. A number of meanings can be assigned to inheritance relationships, including:

- Specialisation/generalisation
- Implementation re-use
- Facility inheritance (mixin classes)
- Taxonomic classification

2.1.4 Association

Association is a relationship between classes which describes a runtime relationship between objects. Its cardinality may be single (1:1) or multiple (1:N).

A particular kind of association between classes indicates the logical part-of relationship. There are two recognised variants of this, namely composition, or containment-by-value, and aggregation meaning a logical part-of relationship. The use of these different types of association in the model is detailed below.

2.2 Other Alternatives

Numerous alternatives were considered both in the original GEHR project, and more recently, including the following:

OMG IDL: the OMG's IDL language lacks assertions and generic types, and its type model is inconsistent (basic "types" are not the same as constructed types, due to the influence of C);

Rumbaugh/Booch/etc notations: none of these notations are formal, and all lack assertions. In any case, they have been superseded by UML;

SGML/XML: SGML is overly complex, and very document-oriented.

XML-schema: not well adapted to information modelling (cf information representation) because it is purely data oriented, and missing a number of important semantics, namely constraints, generic types and multiple inheritance.

Z, Object Z, B: these are worthy of future consideration. Their use now is prevented mainly by a lack of industrial strength tools.

3 Modelling Guide

3.1 General Principles

One crucial point to understand about modelling is that the semantics of all definitions in a model constitute statements about the informational (or behavioural) entities defined by the relevant classes, and no more. Many modellers make the mistake of entering into torturous discussions about the semantics of real world objects based on the arrangement of classes or relationships in a model, when in fact the argument should be the other way around - any model is a formalisation and abstraction, potentially of real world entities, and its meaning does not extend beyond itself. Thus, any concept in a model, such as defined by the type `QUANTITY` should not be understood as being a description of quantities in the real world, but a formal, abstract model of a concept called “quantity” as agreed by the modellers.

3.2 Naming

Class names are in upper case, with underscore separators, enabling them to be easily identified and read. In almost all cases, the full english word has been used. The only exception is the use of “id” rather than identifier, since “id” is generally used as a meaningful word in its own right in IT.

Class feature names (i.e. attribute and method names) are in lower case, underscore separated.

All names have been chosen with implementors and other people in mind who will deal with technical modelling, rather than users. In almost all cases, users will never see the names used in the reference or archetype models. The exception is archetype editors which would normally show the class names of instances of the archetype model which are being created; it is assumed that users of this tool will have a basic technical understanding of the reference and archetype models.

3.3 Operators

Three classes of operator are used:

- infix operators, i.e. any binary operator which appears between the operands, e.g. “+” in the expression “X + Y”
- prefix operators, i.e. any unary operator appearing before the operand, e.g. “-” in “-5”
- postfix operators, i.e. any unary operatory appearing after the operand, e.g. “!” in “x!” (factorial).

3.4 Types

The reference model can be thought of as consisting of a number of classes which fulfill one of two purposes. The first category includes those which represent concretely-modelled concepts like “revision history entry” or “transaction”, while the second includes those whose job it is to represent generic data structures, used to express clinical data whose specific form is defined by archetypes, rather than by the reference model. The general form of the latter can best be understood as structures of name/value pairs, where all nodes in the structure have names, and leaf nodes have values as well. There are accordingly two kinds of “datatypes” used in the model: one for the *attributes* of all classes, and the other for the *values* in the clinical name/value structures. These latter are known as “data value types”, whereas the former are known as “attribute types”. Instances of data value types are the only allowable values in the generic information structures.

In addition to types defined in the model, a number of basic types are assumed in the modelling formalism, which are globally understood in the same (or compatible) ways in all implementation formalisms. These are:

- `Character` (members of a character set)
- `String` (strings of printable characters)
- `Integer` (integer numbers)
- `Real` (real numbers)
- `Double` (double precision floating point real numbers)
- `Boolean` (bivalued entities)
- `Array<T>` (physical container of items indexed by number)
- `List<T>` (implied order, non-unique membership)
- `Set<T>` (no order, unique membership)

3.4.1 Data Value Types

Data value types are characterised by being explicitly modelled and inheriting from the abstract class DATA_VALUE. The names of all of these types are prefixed with "DV_" to differentiate them from types of the same names which may occur in particular implementation technologies, thus DV_DATE rather than DATE and so on. Types which are notionally one of the standard basic types have a specific model. For example, the notional "string" type is modelled as the data value type DV_PLAIN_TEXT.

Data value types are the only types which can be used as data values, e.g. as the type of the ELEMENT.*value* attribute in the *open*EHR EHR reference model, or other similar places where the type DATA_VALUE is specified.

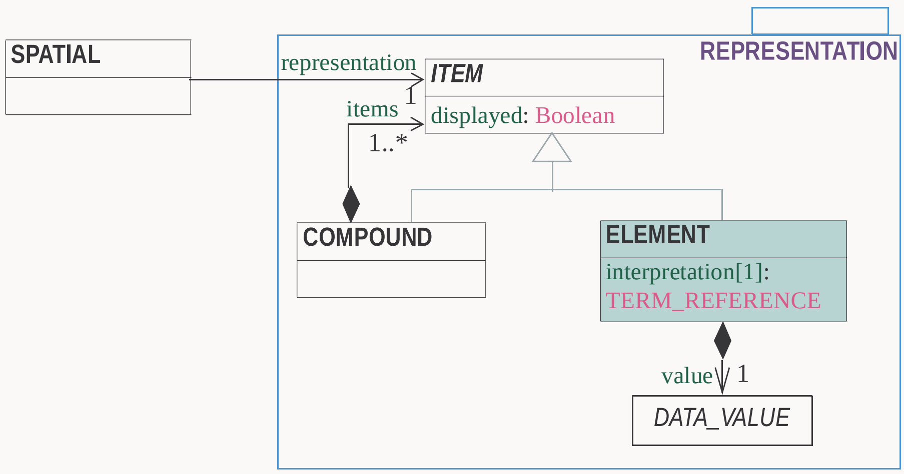

FIGURE 1 EHR RM Representation Package

3.4.2 Attribute Types

Types which can be used for other attributes in model classes include any standard basic type, or any of the data value types. For example, if a string is needed, the class STRING may be used, unless special features of DV_PLAIN_TEXT are required. If a date/time is needed however, since there is no guaranteed standard type for this, the data value type DV_DATE_TIME must be used.

3.4.3 Existence and Cardinality

Existence of attributes is indicated by brackets after the attribute name inside a class box. Possible values are: `[0..1]`, `[1]`, meaning optional and mandatory, respectively. For attributes of container types such as `List<T>`, existence of the whole container is shown the same way. Cardinality of the container is shown by including the container type explicitly.

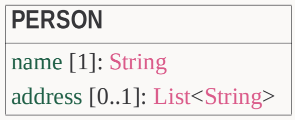

FIGURE 2 Attribute Existence and Cardinality

3.5 Relationships

Relationships between classes in the reference model are of three logical types, described below.

3.5.1 Composition

Composition indicates the part/sub-part relationship where the sub-part can have no meaningful existence outside of the whole, or, put another way, the lifetime of the part is controlled by the whole. For example, in the *open*EHR EHR RM, the class `COMPOSITION` has as a subpart `ACCESS_CONTROL`, illustrated in FIGURE 3. All objects contained within a single “business object”, i.e. `Strings`, `Integers` and other leaf types are always related to the containing object by composition.

In UML, composition is indicated by a black diamond on the class representing the whole. A “part” object can only be in a composition relationship with one “whole” object, i.e. a given instance cannot be part-of multiple wholes.

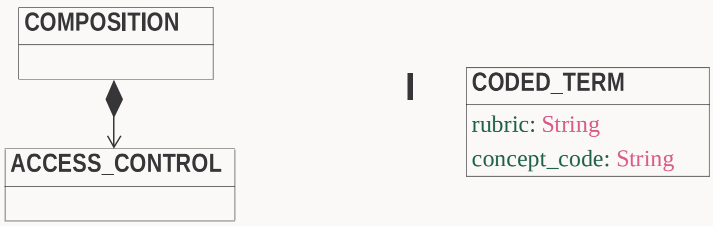

FIGURE 3 Examples of Composition

Semantically, composition corresponds to physical containment by value. Movement or deletion of the whole causes movement or deletion respectively of the part.

3.5.2 Aggregation

Aggregation indicates a logical part/sub-part relationship, where the sub-part can meaningfully exist on its own, i.e. does not need to be deleted if the parent whole is deleted. Consider by way of example the relationship between `HEALTH_CARE_FACILITY` and `HEALTH_CARE_PROFESSIONAL` illustrated in FIGURE 4. The difference in semantics with respect to composition is that aggregation parts and wholes represent business objects (e.g. `HOSPITAL` and `PERSON`), whereas the part objects of composite relationships represent fine-grained constituents inside a business object (e.g. `PERSON` and `PERSON_NAME`).

In UML, aggregation is indicated by a white diamond on the class representing the whole and a key shown on the part, meaning that the whole class contains a key referring to the part class. Movement or deletion of the whole may occur without movement or deletion of the part.

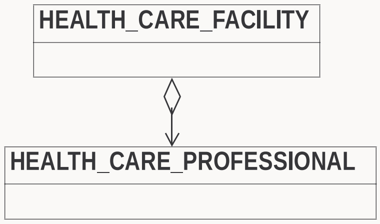

FIGURE 4 Example of Aggregation

A sensible definition of the semantics of aggregation has historically been, and remains, problematic for many modellers. Various books on UML including "UML Distilled" [5], and indeed the authors of UML themselves have noted the confusion[1], and done little to clear it up. Consequently, in some publications, the aggregation relationship has the semantics of allowing a "part" to be a part-of more than one whole. We see this as an error for a number of reasons.

- Firstly, there is no sensible understanding in natural language for the concept of something that is part of more than one whole.
- When the semantics of changing the part are investigated, it is normally found that a change to the part, seen as part-of one whole is not expected to cause a change in the same part seen as part-of another whole. If the change should indeed occur in all wholes, then the whole/part relationships are associations, and do not represent the part-of relationship at all. If changes are not meant to be global to all wholes, then distinct (possibly initially identical) instances of the part must be part-of each whole.
- In some models, aggregation is used in an attempt to represent "re-use". However, re-use is not a meaningful modelling concept, although it is a meaningful implementation concept[2]. The only reasonable modelling interpretation of "re-use" would be that a part is part-of one whole, and there are other similar wholes that have (or will have at a later point in time) a part which is identical to the existing part. In this case, the proper interpretation of aggregation is that each whole has a part, and that there are also constraints or operations (such as copy) which guarantee that the parts of certain wholes are all identical in value to each other.

Consequently, in this document aggregation semantics are defined such that a "part" object can only be in a aggregation relationship with one "whole" object, i.e. a given instance cannot be part-of multiple wholes.

3.5.3 Association

Association indicates any other kind of relationship in which instances of both classes are completely meaningful in themselves. Indicated by no diamond in UML.

3.5.4 Qualified Association

One kind of association which occurs quite commonly is the "qualified association". In contrast with normal associations which are "direct" (i.e. object to object), qualified associations are by symbolic

1. Jim Rumbaugh says of aggregation "think of it as a modelling placebo" (Rumbaugh, Jacobsen and Booch 1999) [3]. Martin Fowler calls it "one of my biggest *betes noires*". Clearly, aggregation is not well understood by the "experts" [5].

2. This is well known as the "flyweight" pattern described in [6]. In the UML diagram for this pattern, an aggregation relationship appears between a flyweight-factory and the flyweight (shared) object; associations appear between the logical "owners" and the factory-generated flyweights)

reference, where the reference is in the form of an attribute value from the target class. FIGURE 5 illutrates the qualified asscociation, and shows an equivalent single class below it. The qualified association is most commonly used when objects of the target class will each have a unique id which can be referenced from elsewhere, in the manner of a primary/foreign key in relational systems (here the foreign key is the attribute *bar_id*: `String` in the class Foo.

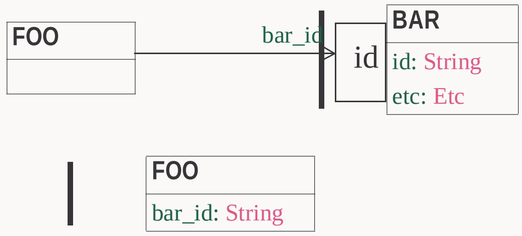

FIGURE 5 Qualified Association

3.6 Constraints and Contracts

Constraints are written in a order predicate logic based on the OMG's Object Constraint Language (OCL), with some differences due to problems in the current definition of OCL.

The keywords used are:

- ***require***: routine precondition
- ***ensure***: routine postcondition
- ***invariant***: class invariant
- ***and***, ***or***, ***and then***, ***or else***, ***implies***, ***xor***: Boolean operators

The various kinds of constraints together form the "contract" of a class, that is the conditions under which its instances interact with instances of other classes (including itself). The following sections describe the three constraint types. See Meyer [8] and Kilov [7] for an explanation of contracts.

3.6.1 Pre-conditions

Pre-conditions are introduced with the keyword ***require***, and consist of a first-order predicate logic expression evaluating to True or False. A precondition represents the truth condition which must be upheld by the caller of a routine to ensure the correct functioning of the routine, i.e. it is a condition *assumed* to be true by the routine. If a pre-condition is violated, the caller is in the wrong.

3.6.2 Post-conditions

Post-conditions are introduced with the keyword ***ensure***, and consist of a first-order predicate logic expression evaluating to True or False. A post-condition represents the truth condition which must be upheld by a routine, i.e. it is a condition *guaranteed* to be true by the routine to the caller. If a post-condition is violated, the called routine is in the wrong.

3.6.3 Invariants

Invariants consist of first order predicate logic statements which apply to the whole class. The meaning is that for every instance of the class, the condition is true at all times, apart from mid-execution of a routine. In other words, object invariants are always true at the points in time when they are accessible to other objects - including prior to calling a routine, and upon exit. If an invariant fails, there is an error in the design of the class. Invariants must be satisfied upon completion of any creation routine.

3.7 Special Types

The type `Any` is assumed as the parent type of all other types, and is the type on which basic operators of equality and assignment are defined. See the Support Reference Model for details.

3.8 Special Instances

The following special instances are indentified in constraints.

- ***Result*** - the result of any function. “Result” is treated like a normal variable whose type is the return type of the function;
- ***Current*** - the current object. Synonymous with “self” in some languages.
- ***Void*** - the empty pseudo-object; conforms to any type. Means the absence of an instance. Synonymous with “null” in many languages.

4 Class Descriptions

Classes in the *open*EHR models are formally described in tables of the form shown below. The various meanings of each section are indicated in this example table.

CLASS	CLASS_NAME	
Purpose	Description of purpose of class in model and information based on it.	
Use	Particular uses of class in the model, or instances of the class in data.	
MisUse	Potential expected misuses of the class, usually based on common misuses or misconceptions of the name of the class.	
CEN	Correspondence to CEN ENV 13606 part 1 - part 4 concepts. These standards were published by CEN in 2000, and can be found on `http://www.centc251.org`.	
Synapses	Correspondence to concepts in the Synapses, SynEx and EHCR-support Action models, produced in various EC-funded (4th framework) post-original GEHR projects. Some of this work is available at `http://www.chime.ucl.ac.uk`.	
GEHR	Correspondence to Australian GEHR models as published on `http://www.gehr.org`.	
HL7	Correspondence to concepts in HL7 version 3 models, as published in various ballots at `http://www.hl7.org`.	
Attributes	**Signature**	**Meaning**
	attr_1: `SOME_TYPE`	Description of this attribute
	attr_n: `SOME_TYPE`	Description of this attribute
Functions	**Signature**	**Meaning**
	func_1(some_args: `SOME_TYPE`): `SOME_TYPE` ***require*** precondition ***ensure*** postcondition	Description of this function
	func_n(some_args: `SOME_TYPE`): `SOME_TYPE`	Description of this function
Invariants	Class invariants. Each mandatory attribute must have an invariant of the form: ***Attr_1_exists***: attr /= Void Other invariants may be stated. All invariants have to be true before and after calls to routines (procedures or functions) made from outside an object.	

Preconditions and postconditions of functions are optional; all preconditions, postconditions and invariants are written in the first-order predicate logic used in the Eiffel language [8], [9] (mainly because this is compilable and testable in any Eiffel tool, including the `Gnu SmallEiffel compiler`).

A References

1 Beale T. *Archetypes: Constraint-based Domain Models for Future-proof Information Systems*. See http://www.deepthought.com.au/it/archetypes.html.

2 Beale T *et al*. *Design Principles for the EHR*. See http://www.deepthought.com.au/openEHR.

3 Booch G, Rumbaugh J, Jacobsen I. *The Unified Modelling Language User Guide*. Addison esley 1999.

4 Fowler M. *Analysis Patterns: Reusable Object Models*
Addison Wesley 1997

5 Fowler M, Scott K. *UML Distilled (2nd Ed.)*
Addison Wesley Longman 2000

6 Gamma E, Helm R, Johnson R, Vlissides J. *Design patterns of Reusable Object-oriented Software.* Addison-Wesley 1995

7 Kilov H, Ross J. *Information Modelling*. Prentice Hall 1994.

8 Meyer B. *Object-oriented Software Construction*, 2nd Ed. Prentice Hall 1997

9 Walden K, Nerson J. *Seamless Object-oriented Software Architecture*. Prentice Hall 1994

END OF DOCUMENT

email: info@openEHR.org web: www.openEHR.org